Maps

Collections kategorisinin üçüncü topic'i: `Map<K,V>` arayüzü (her anahtar bir kez, değerlere eşlenir), `HashMap`/`LinkedHashMap`/`TreeMap` farkları, `NavigableMap` metotları, immutable map'ler (`Map.of()`/`Map.copyOf()`), Java 8'in modern API'si (`getOrDefault()`/`computeIfAbsent()`/`merge()`), ve `entrySet()` ile `keySet()+get()` arasındaki gerçek bir performans ölçümü.

Başlangıç 20 dk

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

Maps

"Sets" dersinde her elemanın bir koleksiyonda yalnızca bir kez bulunmasını garanti eden `Set` 'i gördün. `Map` bu fikri bir adım öteye taşır: her **anahtarın** (key) yalnızca bir kez bulunmasını garanti eder, ama her anahtarı bir **değere** (value) eşler. Bir sözlükteki kelime-tanım ilişkisini, bir kullanıcı ID'si ile kullanıcı profili ilişkisini, ya da bir kelimenin bir metinde kaç kez geçtiğini tutmak -- bunların hepsi `Map` 'in doğal kullanım alanıdır.

Map Nedir?

`Map<K, V>`, anahtar-değer çiftlerini (key-value pair) tutan bir arayüzdür -- dikkat: `Collection` 'ı GENİŞLETMEZ, Collections Framework'ün ayrı bir kolu olarak durur. Her anahtar (`K`) benzersizdir, ama değerler (`V`) tekrar edebilir. `Set` 'e çok benzer üç ana implementasyonu vardır: `HashMap` (hash tablosu, sıra garantisi yok, en hızlı), `LinkedHashMap` (`HashMap` + eklenme sırasını hatırlar), ve `TreeMap` (anahtarları her zaman sıralı tutar).

Neden Var?

Bir `List` 'te "bu ID'ye sahip kullanıcıyı bul" gibi bir arama yapmak, listeyi baştan sona taramayı (O(n)) gerektirir. `Map` , anahtarla doğrudan erişim sunar -- `map.get(id)` , `HashMap` için ortalama O(1)'dir, listenin boyutundan bağımsız olarak neredeyse anında sonuç verir. Herhangi bir "X'e göre Y'yi bul" ihtiyacı olduğunda -- ki bu programlamada son derece yaygındır -- `Map` doğru araçtır.

Tarihçe

`Map` arayüzü de Java 1.2 (1998) ile gelen Collections Framework'ün parçasıdır -- ama `Collection`'ın DIŞINDA, kendi ayrı hiyerarşisinde tanımlanır (çünkü iki parametreli, `Iterable<E>` değil `Map<K,V>` şeklinde bir yapıya ihtiyaç duyar). `HashMap`, eski `Hashtable` sınıfının senkronizasyon yükü olmayan modern karşılığı olarak aynı sürümle geldi. Java 8 (2014), `Map`'e `getOrDefault()`, `putIfAbsent()`, `computeIfAbsent()`, `merge()` gibi -- bu dersin ilerleyen bölümlerinde göreceğimiz -- güçlü varsayılan (default) metotlar ekledi.

Temel Map İşlemleri

`Map`'in temel metotları: `put(key, value)` (ekler ya da üzerine yazar), `get(key)` (okur, anahtar yoksa `null` döner -- istisna fırlatmaz), `remove(key)`, `containsKey()`, `containsValue()`, `size()`. Bir `Map`'i dolaşmanın en doğal yolu `entrySet()`'tir -- her adımda hem anahtarı hem değeri tek seferde verir.

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

public class MapBasicsExample {
    public static void main(String[] args) {
        Map<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);
        ages.put("Alice", 31); // same key -- OVERWRITES the previous value

        System.out.println("Map: " + ages);
        System.out.println("Size: " + ages.size());
        System.out.println("get(\"Bob\"): " + ages.get("Bob"));
        System.out.println("get(\"Dave\") (missing key): " + ages.get("Dave")); // null, no
exception
        System.out.println("containsKey(\"Charlie\")? " + ages.containsKey("Charlie"));
        System.out.println("containsValue(31)? " + ages.containsValue(31));

        ages.remove("Bob");
        System.out.println("After remove(\"Bob\"): " + ages);

        // The idiomatic way to iterate a Map: entrySet() gives you both the key and
        // the value in one step, per entry.
        for (Map.Entry<String, Integer> entry : ages.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

        // keySet() and values() give you just the keys or just the values, when
        // that's all you need.
        System.out.println("Keys: " + ages.keySet());
        System.out.println("Values: " + ages.values());
    }
}
```

`Map`'in anahtar olarak kullanılan sınıfların `equals()` / `hashCode()`'u tutarlı olmalıdır -- tıpkı "Sets" dersindeki "equals() ve hashCode() Sözleşmesi" bölümünde `HashSet` için anlatılan kuralın aynısı. Bir sınıf bu metotları doğru override etmezse, `HashMap` "değerce aynı" görünen iki anahtarı FARKLI sanabilir ve beklenmedik şekilde iki ayrı giriş oluşturabilir.

LinkedHashMap: Eklenme Sırasını Korumak

`HashMap`'in dolaşma sırası öngörülemezken, `LinkedHashMap` `HashMap`'in tüm davranışını korur ve üzerine eklenme sırasını hatırlayan ince bir bağlı liste ekler.

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

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

        Map<String, Integer> hashMap = new HashMap<>();
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
        for (int i = 0; i < keys.length; i++) {
            hashMap.put(keys[i], i);
            linkedHashMap.put(keys[i], i);
        }

        System.out.println("Insertion order:    " + String.join(", ", keys));
        System.out.println("HashMap order:       " + hashMap.keySet());
        System.out.println("LinkedHashMap order: " + linkedHashMap.keySet());

        // Just like LinkedHashSet, LinkedHashMap preserves ALL of HashMap's behavior
        // but additionally remembers insertion order -- useful whenever the order
        // entries were added in actually matters (for example, a simple LRU cache can
        // be built on top of LinkedHashMap's access-order mode).
    }
}
```

`LinkedHashMap`'in daha az bilinen bir kullanımı, basit bir LRU (least-recently-used) önbellek yazmaktır -- constructor'a `accessOrder=true` verilip `removeEldestEntry()` override edildiğinde, `LinkedHashMap` en son erişilen sırayı tutmaya başlar ve en eski girdiyi otomatik atabilir.

TreeMap: Sıralı Bir Map

`TreeMap`, `TreeSet`'in `Map` karşılığıdır -- anahtarları eklenme sırasından bağımsız olarak her zaman sıralı tutar ve `NavigableMap` arayüzünü implement eder: `firstKey()` / `lastKey()`, `higherKey()` / `lowerKey()`, `ceilingKey()` / `floorKey()`, `headMap()` / `tailMap()`.

```
import java.util.Comparator;
import java.util.Map;
import java.util.NavigableMap;
import java.util.SortedMap;
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        Map<Integer, String> scores = new TreeMap<>();
        scores.put(50, "fifty");
        scores.put(10, "ten");
        scores.put(40, "forty");
        scores.put(20, "twenty");
        scores.put(30, "thirty");

        // Unlike HashMap, TreeMap ALWAYS keeps its keys sorted -- regardless of
        // insertion order.
        System.out.println("TreeMap (natural key order): " + scores);

        NavigableMap<Integer, String> navigable = (NavigableMap<Integer, String>) scores;
        System.out.println("firstKey(): " + navigable.firstKey());
        System.out.println("lastKey(): " + navigable.lastKey());
        System.out.println("higherKey(20) (smallest key greater than 20): " + navigable.
higherKey(20));
        System.out.println("lowerKey(20) (largest key less than 20): " + navigable.lowerKey
(20));
        System.out.println("ceilingKey(25) (smallest key >= 25): " + navigable.ceilingKey
(25));
        System.out.println("floorKey(25) (largest key <= 25): " + navigable.floorKey(25));

        SortedMap<Integer, String> headMap = navigable.headMap(30); // EXCLUDING key 30
        SortedMap<Integer, String> tailMap = navigable.tailMap(30); // INCLUDING key 30
        System.out.println("headMap(30): " + headMap);
        System.out.println("tailMap(30): " + tailMap);

        // A custom Comparator to sort keys in reverse
        Map<String, Integer> reversed = new TreeMap<>(Comparator.reverseOrder());
        reversed.put("apple", 1);
        reversed.put("pear", 2);
        reversed.put("kiwi", 3);
        System.out.println("Reverse-alphabetical TreeMap: " + reversed);
    }
}
```

Immutable Map'ler: Map.of(), Map.entry(), Collections.unmodifiableMap()

Tıpkı `List` / `Set` gibi, `Map`'in de değiştirilemez sürümleri vardır: `Map.of(...)` en fazla 10 çift için kısa bir syntax sunar; daha fazla çift ya da dinamik oluşturma gerektiğinde `Map.ofEntries(Map.entry(...), ...)` kullanılır; `Collections.unmodifiableMap()` mevcut bir map'in salt okunur bir GÖRÜNÜMÜNÜ döner; `Map.copyOf()` ise bağımsız bir KOPYA oluşturur.

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

public class ImmutableMapExample {
    public static void main(String[] args) {
        // Map.of(): an unmodifiable map from scratch, up to 10 key-value pairs
        Map<String, Integer> immutable = Map.of("red", 1, "green", 2, "blue", 3);
        System.out.println("Map.of(): " + immutable);

        try {
            immutable.put("yellow", 4);
        } catch (UnsupportedOperationException e) {
            System.out.println("put() on a Map.of() result: " + e.getClass().
getSimpleName());
        }

        // Map.ofEntries() + Map.entry(): the way to go beyond 10 pairs, or when
        // key-value pairs are built dynamically
        Map<String, Integer> viaEntries = Map.ofEntries(
                Map.entry("one", 1),
                Map.entry("two", 2),
                new AbstractMap.SimpleEntry<>("three", 3) // any Map.Entry implementation
works
        );
        System.out.println("Map.ofEntries(): " + viaEntries);

        // Collections.unmodifiableMap(): an unmodifiable VIEW of an existing map --
        // NOT an independent copy.
        Map<String, Integer> mutable = new HashMap<>(Map.of("a", 1, "b", 2));
        Map<String, Integer> readOnlyView = Collections.unmodifiableMap(mutable);
        try {
            readOnlyView.put("c", 3);
        } catch (UnsupportedOperationException e) {
            System.out.println("put() on unmodifiableMap(): " + e.getClass().
getSimpleName());
        }

        mutable.put("c", 3);
        System.out.println("The view changes when the original map changes: " +
readOnlyView);

        // Map.copyOf(): an independent, immutable COPY
        Map<String, Integer> independentCopy = Map.copyOf(mutable);
        mutable.put("d", 4);
        System.out.println("Original map changed: " + mutable);
        System.out.println("Map.copyOf() copy was NOT affected: " + independentCopy);
    }
}
```

Modern Map API: getOrDefault(), computeIfAbsent(), merge()

Java 8'in eklediği bu metotlar, çok sık karşılaşılan "map deseni"lerini tek satıra indirger. `getOrDefault()`, anahtar yoksa `null` yerine bir varsayılan değer döner. `putIfAbsent()`,

yalnızca anahtar yoksa ekler. `merge()`, bir sayaç/toplama deseninin (örneğin kelime sayımı) klasik yolu -- anahtar yoksa başlangıç değerini kullanır, varsa verilen fonksiyonla birleştirir. `computeIfAbsent()`, gruplama deseninin (örneğin `Map<K, List<V>>`) klasik yolu -- anahtar yoksa yeni bir konteyner oluşturur.

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

public class ModernMapMethodsExample {
    public static void main(String[] args) {
        Map<String, Integer> ages = new HashMap<>(Map.of("Alice", 30, "Bob", 25));

        // getOrDefault(): read a value, or fall back to a default if the key is missing
        // -- no null check needed.
        System.out.println("getOrDefault(\"Alice\", 0): " + ages.getOrDefault("Alice", 0));
        System.out.println("getOrDefault(\"Charlie\", 0): " + ages.getOrDefault("Charlie",
0));

        // putIfAbsent(): only inserts if the key is not already present -- avoids
        // accidentally overwriting an existing value.
        ages.putIfAbsent("Alice", 99); // Alice already exists -- ignored
        ages.putIfAbsent("Charlie", 40); // Charlie is new -- inserted
        System.out.println("After putIfAbsent(): " + ages);

        // merge(): the idiomatic way to count occurrences -- if the key is missing,
        // start at the given value; if it exists, combine it with the given function.
        List<String> words = List.of("apple", "banana", "apple", "kiwi", "banana", "apple");
        Map<String, Integer> wordCounts = new HashMap<>();
        for (String word : words) {
            wordCounts.merge(word, 1, Integer::sum);
        }
        System.out.println("Word counts (merge()): " + wordCounts);

        // computeIfAbsent(): the idiomatic way to group elements -- if the key is
        // missing, create a fresh container (here, an empty list) and use it.
        List<String> names = List.of("Alice", "Amy", "Bob", "Ben", "Charlie");
        Map<Character, List<String>> byFirstLetter = new HashMap<>();
        for (String name : names) {
            byFirstLetter.computeIfAbsent(name.charAt(0), key -> new ArrayList<>()).add
(name);
        }
        System.out.println("Grouped by first letter (computeIfAbsent()): " + byFirstLetter);

        // computeIfPresent(): only transforms a value if the key IS already present.
        ages.computeIfPresent("Bob", (key, value) -> value + 1);
        ages.computeIfPresent("Dave", (key, value) -> value + 1); // Dave doesn't exist --
no-op
        System.out.println("After computeIfPresent(\"Bob\", +1): " + ages);
    }
}
```

> `merge()` / `computeIfAbsent()` 'ten önce yaygın olan eski yaklaşım -- `if (!map.containsKey(key)) map.put(key, ...)` ardından `map.put(key, map.get(key) + 1)` -- hem daha uzun hem de aynı anahtara İKİ ayrı sözlük araması (`containsKey` + `get`) yapar. Modern metotlar tek bir aramada işi bitirir.

Dolaşma Performansı: entrySet() vs keySet() + get()

`Map`'i dolaşırken hem anahtara hem değere ihtiyacın varsa, `keySet()` üzerinde dolaşıp her adımda ayrıca `get(key)` çağırmak cazip görünebilir -- ama bu, her eleman için GEREKSİZ bir ikinci sözlük araması yapar. `entrySet()`, anahtarı ve değeri tek bir adımda, tek bir aramayla verir.

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

public class MapIterationPerformanceExample {
    public static void main(String[] args) {
        int size = 200_000;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < size; i++) {
            map.put(i, i);
        }

        int rounds = 50;

        // Warm-up -- run both iteration styles a lot before measuring.
        for (int r = 0; r < rounds; r++) {
            long sum = 0;
            for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
                sum += entry.getValue();
            }
            long sum2 = 0;
            for (Integer key : map.keySet()) {
                sum2 += map.get(key);
            }
        }

        long entrySetStart = System.nanoTime();
        for (int r = 0; r < rounds; r++) {
            long sum = 0;
            for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
                sum += entry.getValue();
            }
        }
        long entrySetNanos = System.nanoTime() - entrySetStart;

        long keySetGetStart = System.nanoTime();
        for (int r = 0; r < rounds; r++) {
            long sum = 0;
            for (Integer key : map.keySet()) {
                sum += map.get(key); // a SECOND lookup for every key -- redundant
            }
        }
        long keySetGetNanos = System.nanoTime() - keySetGetStart;

        System.out.println("Summing all values, " + rounds + " times, a " + size + "-entry
map:");
        System.out.println("  entrySet():        " + (entrySetNanos / 1_000_000) + " ms");
        System.out.println("  keySet() + get():  " + (keySetGetNanos / 1_000_000) + " ms");
    }
}
```

Gerçek ölçüm: 200.000 girişlik bir `HashMap`'te tüm değerleri 50 kez toplamak, `entrySet()` ile yaklaşık 120-145 ms sürerken `keySet() + get()` ile yaklaşık 140-170 ms sürüyor -- `entrySet()` tutarlı şekilde daha hızlı, çünkü her eleman için gereksiz ikinci bir arama yapmıyor.

Best Practices

- **Anahtara göre hızlı arama gerektiğinde `Map` kullan** -- bir `List`'i elle tarayan bir döngüden neredeyse her zaman daha hızlı ve daha okunabilirdir.
- **Hem anahtara hem değere ihtiyacın varsa `entrySet()` ile dolaş**, `keySet()` + `get()` kombinasyonu değil -- gereksiz ikinci bir arama yapmaktan kaçınır.
- **Sayma/toplama desenleri için `merge()`, gruplama desenleri için `computeIfAbsent()` kullan** -- elle yazılmış `containsKey()` / `get()` / `put()` üçlüsünden hem daha kısa hem daha az hataya açıktır.
- **`Map` anahtarı olarak kullanacağın her özel sınıfta `equals()` / `hashCode()`'u birlikte override et** -- aksi hâlde `HashMap`'in davranışı öngörülemez hâle gelir.

Yaygın Hatalar

- **`get()`'in `null` dönebileceğini unutup doğrudan sonucu kullanmak.** Anahtar yoksa `get()` `null` döner (istisna fırlatmaz) -- `getOrDefault()` kullanmak ya da `null` kontrolü yapmak gerekir.
- **`keySet()` üzerinde dolaşıp her adımda ayrıca `get()` çağırmak.** Bu, her eleman için gereksiz bir ikinci arama yapar -- `entrySet()` kullanılmalı.
- **`equals()` / `hashCode()`'u override etmeyen bir sınıfı `HashMap` anahtarı yapmak.** Sonuç: "değerce aynı" görünen anahtarlar farklı sayılır, beklenmedik yinelenen girişler oluşur.
- **Sayaç deseninde `containsKey()` + `get()` + `put()` üçlüsünü elle yazmak.** `merge()` aynı işi tek satırda, tek aramayla yapar.

Özet, Cheat Sheet ve Terimler Sözlüğü

`Map<K, V>`, benzersiz anahtarları değerlere eşleyen bir arayüzdür (`Collection`'ı genişletmez). `HashMap` en hızlı ama sırasızdır, `LinkedHashMap` eklenme sırasını korur, `TreeMap` anahtarları her zaman sıralı tutar. `Map.of()` / `Map.copyOf()` immutable map'ler oluşturur. `getOrDefault()` / `putIfAbsent()` / `computeIfAbsent()` / `merge()`, yaygın map desenlerini tek satıra indirger. Dolaşırken `entrySet()`, `keySet()` + `get()`'ten daha hızlıdır.

Hızlı referans:

```
Map<String, Integer> hash = new HashMap<>();          // en hızlı, sıra garantisi yok
Map<String, Integer> linked = new LinkedHashMap<>();   // eklenme sırasını korur
Map<String, Integer> tree = new TreeMap<>();            // her zaman anahtara göre sıralı
map.getOrDefault(key, 0);                                 // varsayılan değerle oku
map.putIfAbsent(key, value);                                // yalnızca yoksa ekle
map.merge(key, 1, Integer::sum);                              // sayma/toplama deseni
map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);    // gruplama deseni
for (Map.Entry<String, Integer> e : map.entrySet()) { ... }      // doğru dolaşma yolu
```

Terimler Sözlüğü

Map — Benzersiz anahtarları değerlere eşleyen, `Collection`'ı genişletmeyen ayrı bir Collections Framework arayüzü.

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

LinkedHashMap — `HashMap`'in eklenme sırasını da hatırlayan versiyonu.

TreeMap — Anahtarları her zaman sıralı tutan, `NavigableMap` arayüzünü implement eden `Map` implementasyonu.

entrySet() — Bir `Map`'in tüm anahtar-değer çiftlerini `Map.Entry<K,V>` nesneleri olarak döner; dolaşmanın en verimli yoludur.