Maps

The third topic in the Collections category: the `Map<K,V>` interface (each key once, mapped to a value), the differences between `HashMap` / `LinkedHashMap` / `TreeMap`, `NavigableMap` methods, immutable maps (`Map.of()` / `Map.copyOf()`), Java 8's modern API (`getOrDefault()` / `computeIfAbsent()` / `merge()`), and a real performance measurement comparing `entrySet()` and `keySet()+get()`.

Beginner 20 min

Last updated: August 17, 2026
Read online: https://www.learnforgex.com/en/topics/maps

Maps

In the "Sets" lesson you saw `Set`, which guarantees that each element appears only once in a collection. `Map` takes this idea a step further: it guarantees that each **key** appears only once, but maps each key to a **value**. Storing a word-to-definition relationship in a dictionary, a user ID to a user profile, or how many times a word appears in a text -- these are all natural use cases for `Map`.

What Is a Map?

`Map<K, V>` is an interface that holds key-value pairs -- note: it does NOT extend `Collection`, it's a separate branch of the Collections Framework. Every key (`K`) is unique, but values (`V`) can repeat. It has three main implementations, very similar to `Set`'s: `HashMap` (hash table, no ordering guarantee, the fastest), `LinkedHashMap` (`HashMap` plus remembering insertion order), and `TreeMap` (always keeps keys sorted).

Why Does It Exist?

Doing a lookup like "find the user with this ID" in a `List` requires scanning it from start to end (O(n)). `Map` offers direct access by key -- `map.get(id)` is O(1) on average for `HashMap`, returning almost instantly regardless of how large the map is. Whenever you need "find Y given X" -- an extremely common need in programming -- `Map` is the right tool.

History

Like `List` and `Set`, the `Map` interface is part of the Collections Framework that arrived in Java 1.2 (1998) -- but it lives OUTSIDE `Collection`, in its own separate hierarchy (because it needs a two-parameter shape, `Map<K,V>`, rather than `Iterable<E>`). `HashMap` came in the same release as the modern counterpart to the old `Hashtable` class, without its synchronization overhead. Java 8 (2014) added powerful default methods to `Map` -- `getOrDefault()`, `putIfAbsent()`, `computeIfAbsent()`, `merge()` -- which we'll see later in this lesson.

Basic Map Operations

`Map`'s basic methods are: `put(key, value)` (inserts or overwrites), `get(key)` (reads, returns `null` if the key is missing -- doesn't throw), `remove(key)`, `containsKey()`, `containsValue()`, `size()`. The most natural way to iterate a `Map` is `entrySet()` -- it gives you both the key and the value in a single step per entry.

```
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());
    }
}
```

> Classes used as `Map` keys must have a consistent `equals()` / `hashCode()` -- exactly the same rule described for `HashSet` in the "Sets" lesson's "The equals() and hashCode() Contract" section. If a class doesn't override these methods correctly, `HashMap` may treat two keys that look "the same by value" as DIFFERENT, unexpectedly creating two separate entries.

LinkedHashMap: Preserving Insertion Order

While `HashMap`'s iteration order is unpredictable, `LinkedHashMap` preserves all of `HashMap`'s behavior and adds a thin linked list on top that remembers insertion order.

```
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).
    }
}
```

> A lesser-known use of `LinkedHashMap` is building a simple LRU (least-recently-used) cache -- when constructed with `accessOrder=true` and `removeEldestEntry()` overridden, `LinkedHashMap` starts tracking most-recently-accessed order and can automatically evict the oldest entry.

TreeMap: A Sorted Map

`TreeMap` is `TreeSet`'s counterpart for `Map` -- it always keeps its keys sorted regardless of insertion order and implements the `NavigableMap` interface: `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 Maps: Map.of(), Map.entry(), Collections.unmodifiableMap()

Just like `List` / `Set`, `Map` has immutable variants: `Map.of(...)` offers a short syntax for up to 10 pairs; for more pairs or when building entries dynamically, use `Map.ofEntries(Map.entry(...), ...)`; `Collections.unmodifiableMap()` returns a read-only VIEW of an existing map; `Map.copyOf()` creates an independent COPY.

```
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()

These methods, added in Java 8, collapse extremely common "map patterns" into a single line. `getOrDefault()` returns a default value instead of `null` when the key is missing.

`putIfAbsent()` only inserts if the key isn't already present. `merge()` is the classic way to implement a counting/accumulating pattern (like counting words) -- it uses a starting value if the key is missing, or combines it with the given function if it exists. `computeIfAbsent()` is the classic way to implement a grouping pattern (producing a `Map<K, List<V>>`) -- it creates a fresh container if the key is missing.

```
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);
    }
}
```

> The old approach that predates `merge()` / `computeIfAbsent()` -- `if (!map.containsKey(key)) map.put(key, ...)` followed by `map.put(key, map.get(key) + 1)` -- is both longer and does TWO separate dictionary lookups (`containsKey` + `get`) for the same key. The modern methods do the job with a single lookup.

Iteration Performance: entrySet() vs. keySet() + get()

If you need both the key and the value while iterating a `Map`, it might be tempting to iterate over `keySet()` and additionally call `get(key)` at each step -- but this performs an UNNECESSARY second lookup per element. `entrySet()` gives you the key and the value in a single step, with a single lookup.

```
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");
    }
}
```

Real measurement: summing all the values in a 200,000-entry `HashMap` 50 times takes roughly 120-145 ms with `entrySet()`, versus roughly 140-170 ms with `keySet() + get()` -- `entrySet()` is consistently faster because it doesn't perform an unnecessary second lookup per element.

Best Practices

- **Use a `Map` whenever you need fast lookup by key** -- it's almost always faster and more readable than manually scanning a `List`.
- **Iterate with `entrySet()` when you need both the key and the value**, not the `keySet()` + `get()` combination -- this avoids an unnecessary second lookup.
- **Use `merge()` for counting/accumulating patterns, and `computeIfAbsent()` for grouping patterns** -- both are shorter and less error-prone than a hand-written `containsKey()` / `get()` / `put()` sequence.
- **Always override `equals()` / `hashCode()` together for any custom class you'll use as a `Map` key** -- otherwise `HashMap`'s behavior becomes unpredictable.

Common Mistakes

- **Forgetting that `get()` can return `null` and using the result directly.** If the key is missing, `get()` returns `null` (it doesn't throw) -- use `getOrDefault()` or check for `null`.
- **Iterating `keySet()` and additionally calling `get()` at each step.** This performs an unnecessary second lookup per element -- use `entrySet()` instead.
- **Using a class that doesn't override `equals()` / `hashCode()` as a `HashMap` key.** The result: keys that look "the same by value" are treated as different, producing unexpected duplicate entries.
- **Hand-writing the `containsKey()` + `get()` + `put()` sequence for a counting pattern.** `merge()` does the same job in one line with a single lookup.

Summary, Cheat Sheet, and Glossary

`Map<K, V>` is an interface that maps unique keys to values (it doesn't extend `Collection`). `HashMap` is the fastest but unordered, `LinkedHashMap` preserves insertion order, and `TreeMap` always keeps keys sorted. `Map.of()` / `Map.copyOf()` create immutable maps. `getOrDefault()` / `putIfAbsent()` / `computeIfAbsent()` / `merge()` collapse common map patterns into one line. While iterating, `entrySet()` is faster than `keySet()` + `get()`.

Quick reference:

```
Map<String, Integer> hash = new HashMap<>();          // fastest, no ordering guarantee
Map<String, Integer> linked = new LinkedHashMap<>();   // preserves insertion order
Map<String, Integer> tree = new TreeMap<>();            // always sorted by key
map.getOrDefault(key, 0);                                 // read with a default value
map.putIfAbsent(key, value);                                // insert only if absent
map.merge(key, 1, Integer::sum);                              // counting/accumulating
pattern
map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);    // grouping pattern
for (Map.Entry<String, Integer> e : map.entrySet()) { ... }      // the correct way to
iterate
```

Glossary

Map — A separate Collections Framework interface, not extending `Collection`, that maps unique keys to values.

HashMap — The `Map` implementation backed by a hash table; the fastest (O(1)) but with no ordering guarantee.

LinkedHashMap — A `HashMap` variant that additionally remembers insertion order.

TreeMap — A `Map` implementation that always keeps its keys sorted, implementing the `NavigableMap` interface.

entrySet() — Returns all of a `Map`'s key-value pairs as `Map.Entry<K,V>` objects; the most efficient way to iterate.