

Sets

The second topic in the Collections category: the `Set` interface and its no-duplicates guarantee, the differences between `HashSet`/`LinkedHashSet`/`TreeSet`, `NavigableSet` methods, why the `equals()`/`hashCode()` contract is critical for `HashSet` (with a real bug example), set operations (union/intersection/difference), and a real performance measurement comparing `List`/`HashSet`/`TreeSet`.

Beginner 20 min

Generated on 2 October 2026

Read online: <https://www.learnforgex.com/en/topics/sets>

Sets

The `List` interface you saw in the "Lists" lesson allowed duplicate elements and preserved insertion order. Sometimes you want the opposite: a guarantee that an element appears **only once** in the collection, and order usually doesn't matter at all -- think unique user IDs in a system, the distinct words in a piece of text, or a mathematical set. That's what Java's `Set` interface is for.

What Is a Set?

`Set<E>` is an interface that extends `java.util.Collection` and makes one guarantee: **no element can appear more than once**. Unlike `List`, it doesn't offer index-based access (`get(index)`) -- an element can only be reached via `contains()` or by iterating. It has three main implementations: `HashSet` (hash table, no ordering guarantee, the fastest), `LinkedHashSet` (`HashSet` plus a linked list that remembers insertion order), and `TreeSet` (a red-black tree that always keeps elements sorted).

Why Does It Exist?

Manually preventing duplicates in a `List` requires checking with `contains()` before every `add()` -- easy to forget, and slow on large lists because `List.contains()` does a linear ($O(n)$) scan. `Set` embeds the "is this element already here" check directly inside `add()` and does it much faster (depending on the implementation) -- it also directly signals to the reader that "duplicates don't matter here, uniqueness does."

History

Like `List`, the `Set` interface is part of the Collections Framework that arrived in Java 1.2 (1998). `HashSet` is implemented internally using a `HashMap` (storing only the keys).

`LinkedHashSet` and `TreeSet` arrived in the same initial release; `TreeSet` is built on top of `TreeMap`, a sorted structure -- just as `HashSet` is built on top of `HashMap`.

Basic Set Operations

`Set`'s basic methods look a lot like `List`'s -- `add()`, `remove()`, `contains()`, `size()` -- but with two important differences: there's no index-based access, and `add()` silently returns `false` (rather than throwing) if you try to add an element that's already present.

```
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 's iteration order is **unrelated to insertion order** -- it's determined by the elements' positions in the internal hash table, and that order can vary across JDK versions or even between runs. Never rely on HashSet 's iteration order in code where order matters.

LinkedHashSet: Preserving Insertion Order

While HashSet 's iteration order is unpredictable, sometimes you want "remove duplicates, but also keep insertion order." LinkedHashSet does exactly that: it preserves all of HashSet 's behavior while adding a thin doubly-linked list on top to remember insertion order -- at a small memory and performance cost that's negligible in most applications.

```
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: A Sorted Set

TreeSet always keeps its elements **sorted**, regardless of insertion order -- by natural ordering (Comparable) by default, or by a Comparator passed to the constructor. It also implements NavigableSet , offering ordering-specific methods like first() / last() , higher() / lower() (strictly greater/less), ceiling() / floor() (greater-or-equal/less-or-equal), and headSet() / tailSet() (the sub-set before/after a given point).

```

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's `add()` / `contains()` / `remove()` operations are slower than HashSet's ($O(\log n)$ vs. $O(1)$) -- if you don't actually need sorting, HashSet (when order doesn't matter) or LinkedHashMap (when insertion order matters) is a better default.

The equals() and hashCode() Contract

HashSet's "is this element already here" check relies on the elements' `hashCode()` and `equals()` methods. If a class you write doesn't override these, `Object`'s default is used -- which means "equality" collapses to just **same reference** (`==`). The result: two different objects with seemingly identical values are treated as DIFFERENT by HashSet.

```

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.");
}
}

```

If you're going to use a class in a `HashSet` (or as a `HashMap` key), you MUST override `hashCode()` whenever you override `equals()` -- the two must stay consistent (two objects for which `equals()` returns `true` must also have the same `hashCode()`). Overriding only one leads to exactly the kind of silent, hard-to-spot bugs shown in the example above.

Set Operations: Union, Intersection, Difference

`Set` supports the mathematical set operations through three methods: `addAll()` computes the **union**, `retainAll()` computes the **intersection** (keeping only elements present in both sets), and `removeAll()` computes the **difference** (removing elements present in the other set).

```

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 u B (union, addAll): " + union);

        // Intersection: retainAll()
        Set<Integer> intersection = new TreeSet<>(a);
        intersection.retainAll(b);
        System.out.println("A n 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));
    }
}

```

All three of these methods modify the set they're called on **in place** -- if you want to keep the original intact, you need to make a copy first (as in the example above) and perform the operation on the copy.

Performance: List, HashSet, and TreeSet Compared

Let's confirm `Set`'s reason for existing (the "Why Does It Exist?" section) with a real measurement: calling `contains()` thousands of times on the same 20,000-element collection takes milliseconds on `List`, while on `HashSet` / `TreeSet` it's too fast to measure. At this scale, the difference between `HashSet` ($O(1)$) and `TreeSet` ($O(\log n)$) doesn't show up either -- to actually see it, you need a much larger collection and far more repetitions, which is what the second measurement shows.

```

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");
    }
}

```

Real results: calling `contains()` 2,000 times on a 20,000-element collection takes roughly 70-90 ms on `List`, versus 0 ms (too fast to measure) on `HashSet` / `TreeSet`. Scaling up to 200,000 elements and 200,000 repetitions reveals the difference: `HashSet` takes about 9-10 ms, `TreeSet` about 15-21 ms -- both are incomparably faster than `List`, but the theoretical $O(1)$ vs. $O(\log n)$ gap becomes genuinely measurable at scale.

Best Practices

- **If you know a collection shouldn't have duplicates, use a `Set` from the start** -- a `List` plus a manual `contains()` check is both slower and more error-prone.
- **Prefer `HashSet` when order doesn't matter** -- it's the fastest option. Use `LinkedHashSet` when insertion order matters, and `TreeSet` when you need to iterate in sorted order.
- **Always override `equals()` and `hashCode()` together for any class you'll use in a `HashSet` / as a `HashMap` key** -- using your IDE's auto-generation feature is safer than writing them by hand.
- **Make a copy before a set operation (`addAll` / `retainAll` / `removeAll`) if you need to preserve the original** -- these methods mutate in place.

Common Mistakes

- **Assuming `HashSet` 's iteration order matches insertion order.** This isn't guaranteed and can vary by JDK version -- use `LinkedHashSet` if order matters.

- **Putting a custom class into a `HashSet` and forgetting to override `equals()` / `hashCode()`** . The result: objects that look equal by value get added as duplicates, because the `Set` considers them different.
- **Overriding only `equals()` (or only `hashCode()`)** . When the two are inconsistent, `HashSet` 's behavior becomes unpredictable.
- **Using `TreeSet` when you don't need sorting** . It's slower than `HashSet` ($O(\log n)$ vs. $O(1)$) -- reach for it only when you genuinely need sorted iteration.

Summary, Cheat Sheet, and Glossary

`Set<E>` is a collection interface that doesn't allow duplicate elements. `HashSet` is the fastest but unordered, `LinkedHashSet` preserves insertion order, and `TreeSet` always keeps elements sorted (via `NavigableSet` methods). `HashSet` working correctly depends on elements having a consistent `equals()` / `hashCode()` . `addAll()` / `retainAll()` / `removeAll()` compute the union/intersection/difference respectively.

Quick reference:

```
Set<String> hash = new HashSet<>();           // fastest, no ordering guarantee
Set<String> linked = new LinkedHashSet<>();   // preserves insertion order
Set<String> tree = new TreeSet<>();           // always sorted (natural or Comparator)
set.add(x);                                  // returns false if already present, no
exception
Set<String> union = new HashSet<>(a); union.addAll(b);           // union
Set<String> intersection = new HashSet<>(a); intersection.retainAll(b); // intersection
Set<String> difference = new HashSet<>(a); difference.removeAll(b); // difference
```

Glossary

Set — A `Collection` sub-interface that does not allow duplicate elements.

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

LinkedHashSet — A `HashSet` variant that additionally remembers insertion order.

TreeSet — A `Set` implementation that always keeps elements sorted, implementing the `NavigableSet` interface.

hashCode()/equals() contract — The rule stating that two objects for which `equals()` returns `true` must also have equal `hashCode()` ; `HashSet` / `HashMap` correctness depends on it.