Generics with Collections

Generic collections like List<T>, Set<T>, and Map<K,V>, type safety with collections, why List<Object> is NOT List<String> (the full treatment of generics invariance), and type inference with the diamond operator and var. The 5th lesson in the Generics series.

Intermediate 25 min

Last updated: November 7, 2026
Read online: https://www.learnforgex.com/en/topics/generics-with-collections

---

Every mechanic in this series so far — generic classes, generic methods, bounds, wildcards — has mostly used small, invented types like `Box<T>` to keep the focus on the mechanism itself. In practice, though, the single most common place you'll actually use generics is Java's own collections framework. This lesson looks at `List<T>`, `Set<T>`, and `Map<K, V>` specifically, and closes a question "Wildcards" only motivated in passing: why exactly `List<String>` and `List<Object>` are unrelated types.

Generic Collections: List, Set, and Map

`List<T>`, `Set<T>`, and `Map<K, V>` are themselves ordinary generic types, built with exactly the mechanism covered in "Introduction to Generics" — `List` has one type parameter for its elements, `Map` has two, one for its keys and one for its values.

```
import java.util.List;
import java.util.Map;
import java.util.Set;

public class GenericCollectionApisExample {

    public static void main(String[] args) {
        // List<T>: an ordered, index-accessible sequence of one element type.
        List<String> names = List.of("Alice", "Bob", "Charlie");

        // Set<T>: an unordered collection of one element type, with no duplicates.
        Set<String> uniqueNames = Set.copyOf(List.of("Alice", "Bob", "Alice")); // the
duplicate collapses

        // Map<K, V>: an association from one key type to one value type.
        Map<String, Integer> ages = Map.of("Alice", 30, "Bob", 25);

        System.out.println("List: " + names);
        System.out.println("Set size: " + uniqueNames.size());
        System.out.println("Map: " + ages);

        // The exact same three interfaces -- List, Set, Map -- are generic
        // over ANY type; nothing here is specific to String or Integer.
        List<Integer> scores = List.of(90, 85, 78);
        Set<Boolean> flags = Set.of(true, false);
        System.out.println("Scores: " + scores);
        System.out.println("Flags: " + flags);
    }
}
```

The same three interfaces work identically regardless of what they hold — `List<String>` and `List<Integer>` are the same `List`, `Set<String>` and `Set<Boolean>` are the same `Set`. Nothing about the collection APIs themselves changes; only the type argument does.

Type Safety with Collections

The compile-time checking covered generally in "Introduction to Generics" applies to every collection operation — `add`, `put`, `get` — not just to construction.

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

public class CollectionTypeSafetyExample {

    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        // names.add(42); // would NOT compile -- List<String> only accepts String

        String first = names.get(0); // no cast needed -- the compiler already knows it's a
String

        Map<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 30);
        // ages.put("Alice", "thirty"); // would NOT compile -- the value must be an Integer
        // ages.put(42, 30);            // would NOT compile -- the key must be a String

        int age = ages.get("Alice"); // no cast needed here either

        System.out.println(first + " is " + age);
    }
}
```

Expected Output

```
Alice is 30
```

Both `names.add(42)` on a `List<String>` and `ages.put("Alice", "thirty")` on a `Map<String, Integer>` are rejected at compile time, before either mistake could ever reach a running program. Reading back with `names.get(0)` or `ages.get("Alice")` requires no cast, for the same reason — the compiler already knows exactly what type is stored.

Why List<Object> Is Not List<String>

"Wildcards" introduced this rule briefly as motivation; here's the fuller picture. Generics are INVARIANT: even though `String` IS-A `Object`, `List<String>` and `List<Object>` are treated as two completely unrelated types, with no substitutability between them in either direction.

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

public class ListInvarianceExample {

    // If List<Object> WERE a List<String> (it isn't), this method would
    // let you sneak an Integer into a list that was really created as a
    // List<String> -- a broken promise the compiler will never allow.
    static void addNumber(List<Object> list) {
        list.add(42);
    }

    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");

        // addNumber(names); // would NOT compile -- List<String> is not a
        //     List<Object>, even though String IS an Object. If this line
        //     were allowed, addNumber(...) could insert an Integer into
        //     what the caller believes is purely a list of Strings --
        //     exactly the unsafety generics exist to prevent.

        List<Object> anything = new ArrayList<>();
        addNumber(anything); // fine -- this really is a List<Object>

        System.out.println(names);
        System.out.println(anything);
    }
}
```

Expected Output

```
[Alice]
[42]
```

If `List<String>` WERE allowed to be passed where a `List<Object>` is expected, `addNumber(...)` could insert an `Integer` into what its caller believes is purely a list of `String`s — a broken promise the type system would have no way to catch later. Invariance is precisely what prevents that: `List<String>` can only be passed where a `List<String>` (or, as "Wildcards" covered, a `List<? extends Object>` — which every `List` already satisfies) is expected.

> **When you actually need a method to accept a `List` of an unknown or related element type, reach for the wildcard forms from "Wildcards" (`List<?>`, `List<? extends T>`, `List<? super T>`) — that's exactly the tool this invariance rule makes necessary.**

Type Inference with Collections

Constructing a collection doesn't require repeating the type argument twice — Java infers it from context in two common ways.

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

public class DiamondOperatorInferenceExample {

    public static void main(String[] args) {
        // The diamond operator, <>, lets the constructor's type argument be
        // INFERRED from the variable's declared type -- no need to repeat
        // "String" on both sides.
        List<String> names = new ArrayList<>(); // infers ArrayList<String>
        names.add("Alice");

        Map<String, Integer> ages = new HashMap<>(); // infers HashMap<String, Integer>
        ages.put("Alice", 30);

        // "var" infers the variable's type from the right-hand side instead
        // -- here the compiler works out that scores is a List<Integer>,
        // purely from what List.of(...) returns.
        var scores = List.of(90, 85, 78);

        System.out.println(names);
        System.out.println(ages);
        System.out.println(scores);
    }
}
```

The diamond operator, `<>`, infers a constructor's type argument from the variable it's being assigned to — `new ArrayList<>()` assigned to a `List<String>` variable becomes an `ArrayList<String>` without writing `String` again. `var` infers the

variable's own type instead, from whatever's on the right-hand side — `var scores = List.of(90, 85, 78)` gives `scores` the type `List<Integer>`, deduced entirely from `List.of(...)`'s arguments.

A Practical Example

Generic collections are the backbone of everyday Java code — counting, grouping, and looking things up almost always goes through a `Map` or a `List`.

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

public class PracticalWordFrequencyExample {

    // A practical, everyday use of Map<K, V>: counting how many times
    // each word appears -- the kind of code that shows up constantly in
    // real applications.
    static Map<String, Integer> countWords(String[] words) {
        Map<String, Integer> frequency = new HashMap<>();
        for (String word : words) {
            frequency.merge(word, 1, Integer::sum);
        }
        return frequency;
    }

    public static void main(String[] args) {
        String[] words = {"apple", "banana", "apple", "cherry", "banana", "apple"};

        Map<String, Integer> counts = countWords(words);
        for (Map.Entry<String, Integer> entry : counts.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

`countWords(...)` builds a `Map<String, Integer>` from an array of words, using `merge(...)` to increment each word's count — ordinary, practical code that leans entirely on the type safety and inference covered in this lesson, with no casting anywhere.

Best Practices

- **Prefer the collection interfaces (`List`, `Set`, `Map`) as variable and parameter types over concrete implementations (`ArrayList`, `HashMap`) — this mirrors "Interface"'s general guidance and applies just as much to generic collection types.**
- **Use the diamond operator by default when constructing a collection with an explicitly typed variable — there's rarely a reason to repeat the type argument on both sides.**
- **Reach for `var` when a collection's type is already obvious from its initializer, but keep the explicit type when it improves readability for a non-obvious case.**
- **When a method needs to accept a `List` of a related but not identical element type, use a wildcard (from "Wildcards") instead of trying to work around invariance some other way.**

Common Mistakes

- **Trying to pass a `List<String>` where a `List<Object>` is expected and being surprised the compiler rejects it — this is invariance working exactly as designed, not a compiler limitation.**
- **Forgetting that a `Map<K, V>`'s type safety covers keys and values independently — `put(...)` and `get(...)` are both checked against their own respective type parameter.**
- **Writing out the full generic type on the right-hand side of a declaration (`List<String> names = new ArrayList<String>();`) instead of using the diamond operator, adding pure repetition.**
- **Assuming `var` makes a variable's type "less strict" or removes type safety — it only removes the need to WRITE the type; the compiler still enforces it exactly as if it had been spelled out.**

Summary, Cheat Sheet, and Glossary

Summary

- `List<T>`, `Set<T>`, and `Map<K, V>` are ordinary generic types, built with the same mechanism as any custom generic class.
- Collection operations (`add`, `put`, `get`) are all checked at compile time, against the collection's declared type arguments.
- `List<String>` and `List<Object>` are unrelated types because generics are invariant — this is what makes the wildcard forms from "Wildcards" necessary in the first place.
- The diamond operator (`<>`) infers a constructor's type argument from context; `var` infers a variable's entire type from its initializer.
- Generic collections are the most common place generics are actually used in everyday Java code.

Cheat Sheet

```
// The three core generic collection types
List<String> names = List.of("Alice", "Bob");
Set<String> unique = Set.of("Alice", "Bob");
Map<String, Integer> ages = Map.of("Alice", 30);

// Type safety, checked at compile time
List<String> list = new ArrayList<>();
list.add("ok");
// list.add(42); // rejected

// Diamond operator vs var
List<String> a = new ArrayList<>();       // diamond infers ArrayList<String>
var b = List.of(1, 2, 3);                  // var infers List<Integer>

// Invariance
// List<Object> o = names; // rejected -- List<String> is not List<Object>
```

Glossary

- Generic collection: a collection type (`List`, `Set`, `Map`) parameterized by the type(s) it holds.

- **Invariance: the rule that `List<A>` and `List<B>` are unrelated types even when `A` and `B` are related, unless `A` and `B` are the same type.**
- **Diamond operator: `<>`, letting a constructor's type argument be inferred from the variable it's assigned to.**
- **var: a keyword that infers a local variable's entire declared type from its initializer, without removing any compile-time type checking.**