Generic Methods

Generic methods that declare their own type parameter independent of whether their class is generic -- method type parameters, multiple type parameters, type inference, and explicit type witnesses. The 2nd lesson in the Generics series.

Intermediate 20 min

Last updated: November 7, 2026
Read online: https://www.learnforgex.com/en/topics/generic-methods

---

"Introduction to Generics" attached type parameters to a whole CLASS — every method in a `Box<T>` shares the same `T`. But plenty of useful generic code doesn't belong to any particular class at all: a standalone utility method that works with any type, independent of whatever class happens to contain it. This lesson covers giving a single METHOD its own type parameter.

What Is a Generic Method?

A generic method is a method that declares its own type parameter, written in angle brackets right before the return type — `static <T> T firstElement(...)`. That type parameter belongs to the method alone: it has nothing to do with whether the surrounding class is generic, and it gets a fresh value on every single call.

Why Do They Exist?

Not every useful piece of generic behavior naturally belongs to a generic class. A utility method like "give me the first element of any list" isn't really about some `Utils` class being parameterized by a type — it's about THIS ONE METHOD needing to work for any type, called from an ordinary, non-generic class. Generic methods let that flexibility live at the method level, exactly where it's actually needed.

Declaring a Generic Method

The type parameter appears once, right before the return type, and can then be used anywhere in that method's parameter list, body, or return type.

```
import java.util.List;

public class GenericMethodBasicsExample {

    // A generic method: the <T> right before the return type is the
    // method's OWN type parameter -- declared here because this method
    // lives in an ordinary, non-generic class. Nothing about
    // GenericMethodBasicsExample itself is generic; only this one method is.
    static <T> T firstElement(List<T> list) {
        return list.get(0);
    }

    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob");
        String first = firstElement(names); // T is deduced to be String here

        List<Integer> numbers = List.of(10, 20, 30);
        int firstNumber = firstElement(numbers); // same method, T is Integer this time

        System.out.println(first);
        System.out.println(firstNumber);
    }
}
```

Expected Output

```
Alice
10
```

`firstElement(...)` lives in `GenericMethodBasicsExample`, an entirely ordinary, non-generic class — yet the method itself is fully generic. Calling it with a `List<String>` deduces `T` as `String`; calling it with a `List<Integer>` deduces `T` as `Integer`, on the very same method.

Multiple Type Parameters

A method can declare more than one type parameter, comma-separated, exactly the way a generic class can.

```
public class MultipleTypeParametersMethodExample {

    // A method can declare as many type parameters as it needs, separated
    // by commas, exactly like a generic class can -- K and V here are
    // completely independent of each other and get deduced separately.
    static <K, V> String describeEntry(K key, V value) {
        return key + " -> " + value;
    }

    public static void main(String[] args) {
        System.out.println(describeEntry("age", 30));         // K=String, V=Integer
        System.out.println(describeEntry(101, "order-created")); // K=Integer, V=String
        System.out.println(describeEntry(true, 3.14));         // K=Boolean, V=Double
    }
}
```

Expected Output

```
age -> 30
101 -> order-created
true -> 3.14
```

`describeEntry(K key, V value)` deduces `K` and `V` independently on every call — `describeEntry("age", 30)` and `describeEntry(101, "order-created")` are both valid, unrelated uses of the same method, each with its own pair of inferred types.

Type Inference

In nearly every call you've seen so far, the compiler figured out the type parameter entirely on its own, from the arguments you passed — this is TYPE INFERENCE. You almost never have to spell out what the type parameter is.

```
import java.util.List;

public class TypeInferenceExample {

    static <T> T firstElement(List<T> list) {
        return list.get(0);
    }

    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob");

        // Normally, the compiler infers T entirely from the argument --
        // no need to spell it out.
        String inferred = firstElement(names);

        // The same call, but with an explicit TYPE WITNESS: the compiler
        // is told exactly what T is instead of figuring it out. This is
        // rarely necessary -- it exists for the (uncommon) cases where
        // there isn't enough information at the call site for inference
        // to work out T on its own.
        String explicit = TypeInferenceExample.<String>firstElement(names);

        System.out.println(inferred);
        System.out.println(explicit);
    }
}
```

Expected Output

```
Alice
Alice
```

The explicit form, `TypeInferenceExample.<String>firstElement(names)`, is called a TYPE WITNESS — it tells the compiler exactly what `T` should be instead of letting it infer one.

Both calls in this example produce the identical result; the witness form exists for the rarer situations where the compiler doesn't have enough information at the call site to infer the type on its own.

> In everyday code, never write a type witness unless the compiler actually complains without one. `firstElement(names)` is idiomatic; `TypeInferenceExample.<String>firstElement(names)` is verbose noise the compiler doesn't need in the overwhelming majority of cases.

A Method's Type Parameter vs. Its Class's

When a generic method lives inside a generic class, it's worth being precise about which type parameter is which — the method can declare its own, entirely separate from the class's.

```
public class GenericMethodInGenericClassExample {

    // Container<T> is a generic CLASS -- T belongs to the class itself,
    // fixed once for the whole instance (Container<String> means every T
    // in this instance is a String).
    static class Container<T> {
        private final T value;

        Container(T value) {
            this.value = value;
        }

        T getValue() {
            return value;
        }

        // combineWith declares its OWN type parameter, U -- completely
        // separate from the class's T. A single Container<String> instance
        // can call combineWith with an Integer, a Boolean, anything --
        // U is decided fresh on every call, independent of what T is.
        <U> String combineWith(U other) {
            return value + " + " + other;
        }
    }

    public static void main(String[] args) {
        Container<String> box = new Container<>("hello");

        System.out.println(box.combineWith(42));     // U = Integer
        System.out.println(box.combineWith(true));    // U = Boolean
        System.out.println(box.combineWith("world")); // U = String, unrelated to T being
String too
    }
}
```

Expected Output

```
hello + 42
hello + true
hello + world
```

`Container<T>` fixes `T` once, for the whole instance — a `Container<String>` always holds a `String`. But `combineWith`'s `U` is decided fresh on every single call, completely independent of `T` — the same `Container<String>` instance calls `combineWith` with an `Integer`, then a `Boolean`, then a `String`, and each call gets its own `U`.

A Practical Generic Method

Generic methods are common in everyday utility code — anywhere the exact same logic needs to apply to an array or collection of any type.

```
public class PracticalArraySwapExample {

    // A single, practical generic method that works on an array of ANY
    // reference type -- no casting, no duplicating this method once per
    // element type.
    static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String[] args) {
        String[] names = {"Alice", "Bob", "Charlie"};
        swap(names, 0, 2);
        System.out.println(String.join(", ", names));

        Integer[] numbers = {1, 2, 3};
        swap(numbers, 0, 1); // the exact same method, now working on Integer[]
        for (int n : numbers) {
            System.out.print(n + " ");
        }
    }
}
```

Expected Output

```
Charlie, Bob, Alice
2 1 3
```

`swap(...)` works identically on a `String[]` and an `Integer[]` — one method, written once, with no casting and no risk of accidentally swapping elements of mismatched types.

Best Practices

- Prefer a generic method over a generic class when the generic behavior belongs to a single operation, not to a whole family of state a class would hold.
- Let type inference do its job — only reach for an explicit type witness when the compiler genuinely can't infer the type on its own.
- Give a method-level type parameter a name that doesn't shadow a same-named type parameter from its enclosing class, even though the language allows it — it reads as confusingly as reusing a variable name in a nested scope.
- Keep a generic method's type parameter list as small as the operation actually requires.

Common Mistakes

- Forgetting the `<T>` declaration before the return type and writing `static T firstElement(...)` — this doesn't compile, since `T` would be an undeclared type.
- Adding a type witness to every generic method call out of habit, when inference already resolves the type correctly on its own.
- Assuming a generic method's type parameter is somehow tied to its enclosing class's type parameter, when the two are entirely independent, as shown by `combineWith`'s `U`.
- Making an entire class generic when only one of its methods actually needs a type parameter — a generic method on an otherwise ordinary class is often the simpler, more accurate design.

Summary, Cheat Sheet, and Glossary

Summary

- A generic method declares its own type parameter, in angle brackets right before the return type, independent of whether its class is generic.
- Generic methods let type-parameterized behavior live at the method level, for logic that doesn't belong to a whole generic class.
- A method can declare multiple type parameters, comma-separated, each inferred independently per call.
- Type inference resolves a generic method's type parameter from its arguments in almost every case; an explicit type witness is rarely needed.
- A method's own type parameter (like `U` in `combineWith`) is completely separate from its enclosing class's type parameter (like `T` in `Container<T>`).

Cheat Sheet

```
// Generic method in an ordinary class
class Utils {
    static <T> T firstElement(List<T> list) { return list.get(0); }
}
String first = Utils.firstElement(names); // T inferred as String

// Multiple type parameters
static <K, V> String describeEntry(K key, V value) { return key + " -> " + value; }

// Explicit type witness (rarely needed)
String first = Utils.<String>firstElement(names);

// Method type parameter, independent of the class's
class Container<T> {
    <U> String combineWith(U other) { ... } // U != T
}
```

Glossary

- **Generic method**: a method that declares its own type parameter, independent of whether its enclosing class is generic.
- **Type inference**: the compiler deducing a generic method's type parameter from the arguments passed at the call site.
- **Type witness**: an explicit type argument supplied at a generic method's call site, overriding inference.
- **Method-level type parameter**: a type parameter declared on a method itself, distinct from any type parameter its enclosing class declares.