Introduction to Generics

Why generics exist (the pre-generics raw type/cast/ClassCastException problem), how type safety is enforced at compile time, generic classes and interfaces, and the type parameter naming convention. The 1st lesson in the Generics series.

Intermediate 25 min

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

---

Java classes and interfaces you've used throughout this course — `List`, `Optional`, your own classes from "Interface" and "Abstract Class" — are written once and reused everywhere. Generics extend that same idea one level further: instead of writing a class once per TYPE, you write it once for ANY type, while the compiler still checks every use of it as strictly as if you'd written a dedicated version by hand. This lesson opens a new series on exactly that mechanism.

What Are Generics?

Generics let a class, interface, or method be parameterized by a TYPE, the same way a method is parameterized by ordinary values. `List<String>` and `List<Integer>` are both built from the exact same `List` class — the part in angle brackets, the TYPE ARGUMENT, tells the compiler which specific type this particular use is meant to hold, without needing a separate `StringList` and `IntegerList` class.

Why Do They Exist?

Before generics (introduced in Java 5, 2004), a general-purpose container like a `List` had no way to remember what type of element it held — it could only store `Object`, the common ancestor of everything. Reading an element back required an explicit cast, and nothing stopped you from putting the wrong type in to begin with; the mistake surfaced later as a `ClassCastException`, often far from where the bad element was actually added.

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

public class PreGenericsCastingProblemExample {

    public static void main(String[] args) {
        // A raw (non-generic) List: the compiler has no idea what type of
        // element it's supposed to hold, so it accepts anything...
        List names = new ArrayList();
        names.add("Alice");
        names.add("Bob");
        names.add(42); // ...including something that clearly doesn't belong here.

        // Reading requires an explicit, unchecked cast -- the compiler
        // trusts you that the element really is a String.
        for (Object item : names) {
            String name = (String) item; // throws at runtime on the int 42
            System.out.println(name.toUpperCase());
        }
    }
}
```

This raw (non-generic) `List` accepts a `String`, then an `Integer`, without complaint — the failure only shows up later, at the cast, when the loop reaches the misplaced element. Generics exist to catch exactly this class of mistake at COMPILE time, before the program ever runs.

Type Parameters

The letter inside the angle brackets — `T` in `Box<T>`, `K` / `V` in `Pair<K, V>` — is called a TYPE PARAMETER: a placeholder name for a type that gets filled in later, when the class is actually used. By convention, Java code uses short, single-uppercase-letter names: `T` for a generic type in general, `E` for a collection element, `K` and `V` for a map's key and value, `N` for a number. Nothing in the language enforces these letters specifically, but every Java codebase you'll read expects them.

Generic Classes

A class becomes generic by declaring one or more type parameters right after its name, and using that parameter anywhere a concrete type would normally appear — field types, method parameters, return types.

```
public class GenericBoxClassExample {

    // A generic class: T is a type parameter, a placeholder for whatever
    // real type is supplied when Box is used. The class is written ONCE,
    // but works correctly for any type without any casting or duplication.
    static class Box<T> {
        private T content;

        void set(T content) {
            this.content = content;
        }

        T get() {
            return content;
        }
    }

    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.set("hello");
        String value = stringBox.get(); // no cast needed -- the compiler already knows
it's a String

        Box<Integer> intBox = new Box<>();
        intBox.set(42);
        int number = intBox.get(); // same class, different type argument, still no cast

        System.out.println(value.toUpperCase());
        System.out.println(number * 2);
    }
}
```

Expected Output

```
HELLO
84
```

`Box<T>` is written exactly once, yet `Box<String>` and `Box<Integer>` behave like two completely separate, fully type-safe classes — `stringBox.get()` returns a `String` with no cast required, and the compiler would reject trying to `set(...)` an `Integer` into a `Box<String>`.

A class isn't limited to a single type parameter — as many as the design needs can be declared, separated by commas.

```
public class GenericPairClassExample {

    // A generic class with TWO type parameters -- K and V here are just
    // names (by convention, K for "key", V for "value"); nothing forces
    // this specific naming beyond readability.
    static class Pair<K, V> {
        private final K key;
        private final V value;

        Pair(K key, V value) {
            this.key = key;
            this.value = value;
        }

        K getKey() {
            return key;
        }

        V getValue() {
            return value;
        }

        @Override
        public String toString() {
            return key + " = " + value;
        }
    }

    public static void main(String[] args) {
        Pair<String, Integer> ageEntry = new Pair<>("Alice", 30);
        Pair<Integer, String> idEntry = new Pair<>(101, "order-created");

        System.out.println(ageEntry);
        System.out.println(idEntry);
    }
}
```

Expected Output

```
Alice = 30
101 = order-created
```

`Pair<K, V>` uses two independent type parameters — `Pair<String, Integer>` and `Pair<Integer, String>` are both valid, unrelated uses of the exact same class, each fully checked by the compiler for its own two types.

Generic Interfaces

Interfaces can declare type parameters exactly the same way classes do — the type parameter then flows through every method the interface declares.

```
public class GenericInterfaceExample {

    // A generic interface: the type parameter T flows through the method
    // signatures exactly like it would in a generic class.
    interface Repository<T> {
        void save(T item);
        T findLatest();
    }

    // A concrete class implementing the interface supplies a real type
    // argument -- every method in the implementation now deals with
    // Order specifically, no casting required anywhere.
    static class InMemoryOrderRepository implements Repository<Order> {
        private Order latest;

        @Override
        public void save(Order item) {
            latest = item;
        }

        @Override
        public Order findLatest() {
            return latest;
        }
    }

    record Order(int id, double total) {
    }

    public static void main(String[] args) {
        Repository<Order> repository = new InMemoryOrderRepository();
        repository.save(new Order(1, 49.99));

        Order latest = repository.findLatest(); // already an Order, no cast
        System.out.println(latest);
    }
}
```

`Repository<T>` declares `save(T item)` and `T findLatest()` in terms of `T`. `InMemoryOrderRepository implements Repository<Order>` supplies the real type argument — every method in that implementation now deals with `Order` specifically, and `findLatest()` returns an `Order` with no cast needed anywhere in `main`.

Type Safety in Compile-Time Action

The concrete benefit of everything above is that invalid usage is rejected before the program ever runs, not discovered later as a runtime crash.

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

public class TypeSafetyCompileTimeCheckExample {

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

        // names.add(42); // does NOT compile -- int cannot be used where
        //                   the compiler expects a String. This line is
        //                   commented out only because it wouldn't compile
        //                   otherwise; uncomment it to see the real error.

        // Reading back requires no cast at all -- the compiler already
        // knows every element is a String, because that's the only thing
        // this particular List<String> was ever allowed to accept.
        for (String name : names) {
            System.out.println(name.toUpperCase(Locale.ROOT));
        }
    }
}
```

Expected Output

```
ALICE
BOB
```

Trying to `add(42)` to a `List<String>` simply does not compile — there's no cast to forget, no `ClassCastException` waiting to happen later. This is the core promise generics make: the kind of mistake shown in the very first example becomes impossible to write in the first place.

> Whenever you see a raw type being used (a generic class referenced without its angle brackets, like plain `List` instead of `List<String>`), treat it as a warning sign — the compiler falls back to pre-generics behavior for that specific usage, silently losing all of the type-safety benefits covered in this lesson.

Best Practices

- Always supply a type argument when using a generic class or interface — avoid raw types entirely in code you write.
- Follow the standard single-letter naming convention (`T`, `E`, `K`, `V`, `N`) for type parameters, so other Java developers recognize their role immediately.
- Reach for a generic class when you find yourself writing nearly identical code for different types — that duplication is exactly what generics remove.

- Keep the number of type parameters small; a class with too many quickly becomes harder to read than the duplication it was meant to avoid.

Common Mistakes

- Using a raw type (`List` instead of `List<String>`) and being surprised later by a `ClassCastException` that generics were supposed to prevent.
- Assuming `Box<Object>` can hold anything the way pre-generics code used to — it can only hold what's declared, and (as later lessons cover) it isn't interchangeable with `Box<String>`.
- Inventing unconventional type parameter names that make code harder for other developers to read at a glance.
- Confusing a type PARAMETER (`T`, the placeholder in the class declaration) with a type ARGUMENT (`String`, the real type supplied when the class is used) — the two terms describe different sides of the same relationship.

Summary, Cheat Sheet, and Glossary

Summary

- Generics let a class, interface, or method be parameterized by a type, checked fully by the compiler.
- Before generics, general-purpose containers stored `Object` and required unchecked casts, deferring type mistakes to runtime.
- A type parameter (`T`, `K`, `V`, ...) is a placeholder filled in with a real type argument when the class is used.
- A class or interface can declare one or more type parameters, used throughout its fields, method parameters, and return types.
- The core benefit is catching type mistakes at compile time instead of discovering them later as a `ClassCastException`.

Cheat Sheet

```
// Generic class, one type parameter
class Box<T> {
    private T content;
    void set(T content) { this.content = content; }
    T get() { return content; }
}
Box<String> box = new Box<>();

// Generic class, two type parameters
class Pair<K, V> {
    K getKey() { ... }
    V getValue() { ... }
}
Pair<String, Integer> entry = new Pair<>("Alice", 30);

// Generic interface
interface Repository<T> {
    void save(T item);
    T findLatest();
}
class OrderRepository implements Repository<Order> { ... }
```

Glossary

- **Generics**: the mechanism letting a class, interface, or method be parameterized by a type.
- **Type parameter**: a placeholder name (`T`, `E`, `K`, `V`, `N`) declared on a generic class, interface, or method.
- **Type argument**: the real, concrete type supplied for a type parameter when a generic class is actually used.
- **Raw type**: a generic class or interface used without any type argument, falling back to pre-generics behavior.
- **Type safety**: the compiler rejecting incompatible types at compile time, before the program can run with a mistaken value in place.