Type Erasure and Generic Limitations

What type erasure is, what happens to generic types at runtime, why `new T()` and generic arrays aren't allowed, static members and generics, and how generics' runtime limitations actually surface in practice (raw types, unchecked warnings, heap pollution). The 6th and final lesson in the Generics series.

Intermediate 25 min

Last updated: November 7, 2026
Read online: https://www.learnforgex.com/en/topics/type-erasure-and-generic-limitations

---

Every lesson in this series so far has focused on what generics let you DO. This closing lesson explains WHY a handful of things you might reasonably expect to work simply don't — `new T()`, `new T[10]`, a static field of type `T`, an `instanceof` check against `List<String>`. All of these trace back to a single design decision made when generics were added to Java: type erasure.

What Is Type Erasure?

Type erasure is the way the Java compiler implements generics: type parameters and type arguments are used to check your code at COMPILE time, and then thrown away — ERASED — before the code becomes runtime bytecode. `List<String>` and `List<Integer>` are both compiled down to the exact same raw `List` bytecode; the compiler inserts casts where needed and verifies everything is consistent beforehand, but none of that type information survives into the running program.

Why Does It Exist?

When generics were introduced in Java 5 (2004), an enormous amount of existing Java code and already-compiled `.class` files used raw types like `List`. Erasure was the design choice that let generic code interoperate with all of that pre-existing, non-generic code and bytecode without breaking it — a new `List<String>` compiles down to something a pre-Java-5 JVM (and pre-Java-5 code calling into it) could still run. The trade-off is exactly what this lesson covers: several things that feel like they should work at runtime don't, because the information they'd need was erased.

What Happens to Generic Types at Runtime

Since the type argument doesn't survive compilation, two collections built with different type arguments are, at runtime, indistinguishable.

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

public class TypeErasureRuntimeInspectionExample {

    public static void main(String[] args) {
        List<String> strings = new ArrayList<>();
        List<Integer> integers = new ArrayList<>();

        // At runtime, the type argument is GONE -- both lists share the
        // exact same runtime class, ArrayList, with no trace of String or
        // Integer left anywhere in that Class object.
        System.out.println(strings.getClass());
        System.out.println(integers.getClass());
        System.out.println("same runtime class? " + (strings.getClass() == integers.
getClass()));

        Object value = strings;

        // if (value instanceof List<String>) { } // would NOT compile --
        //     there is no such runtime information as "a List of String"
        //     to check against; the type argument was erased.
        if (value instanceof List<?>) { // only the RAW type can be checked
            System.out.println("it's some kind of List, element type unknown at runtime");
        }
    }
}
```

Expected Output

```
class java.util.ArrayList
class java.util.ArrayList
same runtime class? true
it's some kind of List, element type unknown at runtime
```

`strings.getClass()` and `integers.getClass()` return the exact same `Class` object — there is no runtime trace of `String` or `Integer` left anywhere to tell them apart. `instanceof List<String>` doesn't even compile, for the same reason: there's no such runtime information as "a `List` of `String`" to check against — only the raw `instanceof List<?>` is legal.

Why `new T()` Is Not Allowed

Creating an instance requires the JVM to know a real, concrete class to call a constructor on. Because of erasure, at runtime the JVM has no idea what `T` actually is — so `new T()` has no real class to instantiate.

```
import java.util.function.Supplier;

public class GenericMethodConstructionWorkaroundExample {

    // static <T> T createDefault() {
    //     return new T(); // would NOT compile -- because of erasure, at
    //         runtime the JVM has no idea what T actually is, so "new T()"
    //         has no real class to call a constructor on.
    // }

    // The common workaround: have the CALLER supply a way to create a T,
    // since only the caller actually knows what T is at that point.
    static <T> T createDefault(Supplier<T> factory) {
        return factory.get();
    }

    public static void main(String[] args) {
        String s = createDefault(String::new);
        StringBuilder sb = createDefault(StringBuilder::new);

        System.out.println("'" + s + "'");
        System.out.println("'" + sb + "'");
    }
}
```

Expected Output

```
''
''
```

The standard workaround: since only the CALLER of a generic method actually knows what `T` is at that point, have the caller supply a way to create one — here, a `Supplier<T>` (often a constructor reference like `String::new`) plays that role instead of the method trying to `new T()` itself.

Generic Arrays: Why You Can't Create Them Directly

Unlike a `List`, a Java array remembers its element type at RUNTIME — but erasure means there's no real `T` to give an array at runtime either, so `new T[10]` doesn't compile.

```
public class GenericArrayWorkaroundExample {

    static class SimpleStack<T> {
        // private T[] elements = new T[10]; // would NOT compile -- an
        //     array remembers its element type at RUNTIME (unlike a List),
        //     but erasure means there is no real T to give it at runtime.

        @SuppressWarnings("unchecked")
        private final T[] elements = (T[]) new Object[10]; // the standard
        // workaround: build an Object[], then cast it to T[] -- this
        // produces an "unchecked" compiler warning because the cast can't
        // truly be verified, but it's safe AS LONG AS this array is never
        // exposed outside the class as a T[] (only ever accessed through
        // methods like get/push below, which return individual T values).

        private int size = 0;

        void push(T value) {
            elements[size++] = value;
        }

        T pop() {
            return elements[--size];
        }
    }

    public static void main(String[] args) {
        SimpleStack<String> stack = new SimpleStack<>();
        stack.push("first");
        stack.push("second");

        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

Expected Output

```
second
first
```

The common workaround inside a generic class: build a plain `Object[]`, then cast it to `T[]`. This produces an "unchecked" compiler warning, since the cast can't truly be verified — it's safe here only because the array is never exposed outside the class as a real `T[]`, only ever accessed through methods that hand back individual `T` values.

> An `@SuppressWarnings("unchecked")` cast like the one in `SimpleStack` is a promise YOU are making to the compiler, not something the compiler has verified for you. Only use it when you can actually reason through why the underlying operation is safe, exactly as `SimpleStack` keeping its array private demonstrates.

Static Members and Generics

A `static` field or method belongs to the CLASS itself, shared across every instance — but a class's type parameter is only known PER INSTANCE (`Container<String>` and `Container<Integer>` can coexist), so there's no single, consistent `T` a static member could refer to.

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

public class StaticMembersAndGenericsExample {

    static class Container<T> {
        // private static T sharedDefault; // would NOT compile -- a static
        //     field belongs to the CLASS, shared across every instance,
        //     but T is only known per INSTANCE (Container<String> vs
        //     Container<Integer> could coexist) -- there is no single T
        //     a static field could consistently hold.

        // static void printDefault(T value) { } // would NOT compile for
        //     the same reason -- a static method has no particular
        //     instance, so it has no T to refer to either.

        // A static method CAN declare its own, independent type
        // parameter, exactly as covered in "Generic Methods" -- this one
        // has nothing to do with Container's T.
        static <U> List<U> singletonList(U value) {
            List<U> list = new ArrayList<>();
            list.add(value);
            return list;
        }
    }

    public static void main(String[] args) {
        System.out.println(Container.<String>singletonList("hello"));
        System.out.println(Container.<Integer>singletonList(42));
    }
}
```

Expected Output

```
[hello]
[42]
```

Neither a `static T sharedDefault` field nor a `static void printDefault(T value)` method can refer to `Container`'s `T` — there is no one `Container` instance's `T` a static context could mean. What a static method CAN do, exactly as covered in "Generic Methods," is declare its own, entirely independent type parameter — `singletonList`'s `U` has nothing to do with `Container`'s `T` at all.

Runtime Limitations in Practice

The limitations covered so far aren't just theoretical — a raw type (a generic type used with no type argument at all) can still slip past compile-time checking entirely, exactly the way pre-generics code always did.

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

public class UncheckedWarningHeapPollutionExample {

    @SuppressWarnings({"unchecked", "rawtypes"})
    static void pollute(List list) { // a RAW type parameter -- no type
        // argument at all, so the compiler applies none of the checking
        // covered throughout this series to this method.
        list.add("not an Integer");
    }

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);

        pollute(numbers); // compiles, because raw List accepts anything --
        //     this is exactly the pre-generics behavior "Introduction to
        //     Generics" opened with, still reachable today through a raw type.

        try {
            for (int n : numbers) { // fails here, not where the bad value
                //     was actually inserted -- the runtime cast happens at
                //     this read, far from the real mistake.
                System.out.println(n);
            }
        } catch (ClassCastException e) {
            System.out.println("blew up at read time: " + e.getMessage());
        }
    }
}
```

`pollute(...)` takes a raw `List`, so the compiler applies none of the type checking the rest of this series relies on — inserting a `String` into what's really a `List<Integer>` compiles fine. The failure doesn't happen at the insertion, though; it happens later, at the read, when the compiler-inserted cast to `Integer` finally runs and throws a `ClassCastException` — far from where the actual mistake was made. This is precisely the situation "Introduction to Generics" opened this whole series with, still reachable today whenever a raw type is used.

Best Practices

- Never use a raw type in code you write — the moment you do, you lose every compile-time guarantee this series has covered, exactly as `pollute(...)` shows.

- When a generic method genuinely needs to construct a `T`, accept a factory (like `Supplier<T>`) from the caller instead of trying to `new T()`.
- If you must build a generic array internally, keep the underlying `Object[]` completely private, and only ever expose individual `T` elements through it, never the raw array itself.
- Reach for a `static` generic method with its own type parameter when class-level state genuinely isn't needed — it sidesteps the static/generics limitation entirely.

Common Mistakes

- Writing `new T()` or `new T[size]` and being confused by the compiler error instead of recognizing it as a direct consequence of erasure.
- Trying `instanceof List<String>` and expecting it to work, instead of the only legal form, `instanceof List<?>`.
- Declaring a `static` field or method that references a class's own type parameter, not realizing static context has no particular instance's `T` to draw on.
- Ignoring an "unchecked" compiler warning as boilerplate noise, when it's often flagging exactly the kind of erasure-related unsafety `UncheckedWarningHeapPollutionExample` demonstrates.

Summary, Cheat Sheet, and Glossary

Summary

- Type erasure removes type arguments after compile-time checking, so generic type information doesn't exist at runtime.
- Erasure exists to let generic code interoperate with the pre-generics code and bytecode that existed before Java 5.
- Two collections with different type arguments share the same runtime class; only a raw `instanceof` check is legal.
- `new T()` and `new T[]` don't compile, because erasure leaves no real class for the JVM to instantiate at runtime.
- A class's static members can't reference its type parameter, since a static context has no particular instance's type argument to use.

Cheat Sheet

```
// Runtime erasure
List<String> a = new ArrayList<>();
List<Integer> b = new ArrayList<>();
a.getClass() == b.getClass(); // true

// instanceof: only the raw form is legal
if (obj instanceof List<?>) { ... }

// new T() workaround: caller supplies a factory
static <T> T createDefault(Supplier<T> factory) { return factory.get(); }

// Generic array workaround (inside the class only)
@SuppressWarnings("unchecked")
T[] elements = (T[]) new Object[10];

// Static members can't use the class's T, but CAN declare their own
static <U> List<U> singletonList(U value) { ... }
```

Glossary

- **Type erasure**: the compiler's implementation strategy for generics -- checking type arguments at compile time, then discarding them before runtime.
- **Raw type**: a generic type used with no type argument, receiving none of generics' compile-time checking.
- **Heap pollution**: a situation where a variable of a parameterized type refers to an object that isn't actually of that parameterized type, usually via a raw type or an unchecked cast.
- **Unchecked warning**: a compiler warning marking a cast or operation the compiler can't fully verify is type-safe, due to erasure.