Type Erasure ve Generics'in Kısıtları

Type erasure'ın (tür silme) ne olduğu, generic türlere çalışma zamanında ne olduğu, `new T()` ve generic array'lerin neden izin verilmediği, statik üyeler ve generics, ve generics'in çalışma zamanı kısıtlarının pratikte (raw type/unchecked uyarı/heap pollution) nasıl karşımıza çıktığı. Generics serisinin 6. ve son dersi.

Orta 25 dk

Son güncelleme: 7 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/type-erasure-and-generic-limitations

---

Bu serideki her ders şimdiye kadar generics'in ne YAPMANA izin verdiğine odaklandı. Bu kapanış dersi, makul biçimde çalışmasını bekleyebileceğin bir avuç şeyin neden basitçe ÇALIŞMADIĞINI açıklıyor — `new T()`, `new T[10]`, `T` türünde statik bir alan, `List<String>`'e karşı bir `instanceof` kontrolü. Bunların hepsi, generics Java'ya eklendiğinde alınan tek bir tasarım kararına dayanır: type erasure (tür silme).

Type Erasure Nedir?

Type erasure, Java derleyicisinin generics'i uygulama şeklidir: tür parametreleri ve tür argümanları kodunu DERLEME zamanında kontrol etmek için kullanılır, sonra çalışma zamanı bytecode'u olmadan önce ATILIR — SİLİNİR. `List<String>` ve `List<Integer>`, ikisi de tam olarak aynı raw `List` bytecode'una derlenir; derleyici gerektiğinde cast'ler ekler ve önceden her şeyin tutarlı olduğunu doğrular, ama bu tür bilgisinin hiçbiri çalışan programa kadar hayatta kalmaz.

Neden Var?

Generics Java 5'e (2004) eklendiğinde, mevcut Java kodunun ve zaten derlenmiş `.class` dosyalarının muazzam bir kısmı `List` gibi raw type'lar kullanıyordu. Erasure, generic kodun bu önceden var olan, generic olmayan kod ve bytecode ile onu bozmadan birlikte çalışmasına izin veren tasarım kararıydı — yeni bir `List<String>`, Java 5 öncesi bir JVM'in (ve ona çağrı yapan Java 5 öncesi kodun) hâlâ çalıştırabileceği bir şeye derlenir. Ödünleşim tam olarak bu dersin işlediği şey: çalışma zamanında çalışması gerektiği hissedilen ama ihtiyaç duydukları bilginin silinmiş olması nedeniyle çalışmayan birkaç şey.

Generic Türlere Çalışma Zamanında Ne Olur?

Tür argümanı derlemeyi atlatamadığı için, farklı tür argümanlarıyla inşa edilmiş iki koleksiyon, çalışma zamanında birbirinden AYIRT EDİLEMEZ.

```
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");
        }
    }
}
```

Beklenen Çıktı

```
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()` ve `integers.getClass()`, tam olarak aynı `Class` nesnesini döndürür — `String` ya da `Integer`'ı birbirinden ayırt edecek hiçbir çalışma zamanı izi hiçbir yerde kalmaz. `instanceof List<String>` aynı nedenle derlenmez bile: karşılaştırılacak "bir `String List`i" gibi bir çalışma zamanı bilgisi yoktur — yalnızca raw `instanceof List<?>` geçerlidir.

new T() Neden İzin Verilmez?

Bir instance oluşturmak, JVM'in bir constructor'ı çağıracağı gerçek, somut bir sınıfı bilmesini gerektirir. Erasure yüzünden, çalışma zamanında JVM'in `T`'nin gerçekte ne olduğu hakkında hiçbir fikri yoktur — bu yüzden `new T()`'nin instance oluşturacak gerçek bir sınıfı yoktur.

```
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 + "'");
    }
}
```

Beklenen Çıktı

```
''
''
```

Standart geçici çözüm: generic bir metodun yalnızca ÇAĞIRANI o noktada `T`'nin ne olduğunu gerçekten bildiği için, bir tane oluşturmanın bir yolunu çağırana sağlat — burada, metot kendisi `new T()` yapmaya çalışmak yerine bu rolü bir `Supplier<T>` (genelde `String::new` gibi bir constructor referansı) üstleniyor.

Generic Array'ler: Neden Doğrudan Oluşturulamazlar

Bir `List`'ten farklı olarak, bir Java array'i eleman türünü ÇALIŞMA ZAMANINDA hatırlar — ama erasure, bir array'e çalışma zamanında verilecek gerçek bir `T` de olmadığı anlamına gelir, bu yüzden `new T[10]` derlenmez.

```
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());
    }
}
```

Beklenen Çıktı

```
second
first
```

Generic bir sınıfın içindeki yaygın geçici çözüm: düz bir `Object[]` inşa et, sonra onu `T[]`'e cast et. Bu, cast'in gerçekten doğrulanamaması nedeniyle bir "unchecked" derleyici uyarısı üretir — bu yalnızca array'in sınıf dışına asla gerçek bir `T[]` olarak açığa çıkmadığı, yalnızca tekil `T` değerlerini geri veren metotlar aracılığıyla erişildiği için güvenlidir.

> `SimpleStack`'tekine benzer bir `@SuppressWarnings("unchecked")` cast'i, derleyicinin senin için doğruladığı bir şey değil, SENİN derleyiciye verdiğin bir sözdür. Bunu yalnızca, `SimpleStack`'in array'ini private tutmasının gösterdiği gibi, altında yatan işlemin neden güvenli olduğunu gerçekten akıl yürütebildiğinde kullan.

Statik Üyeler ve Generics

`static` bir alan ya da metot, her instance arasında paylaşılan SINIFIN kendisine aittir — ama bir sınıfın tür parametresi yalnızca INSTANCE BAŞINA bilinir (`Container<String>` ve `Container<Integer>` bir arada var olabilir), bu yüzden statik bir üyenin başvurabileceği tek, tutarlı bir `T` yoktur.

```
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));
    }
}
```

> **Beklenen Çıktı**
>
> ```
> [hello]
> [42]
> ```

Ne `static T sharedDefault` bir alan ne de `static void printDefault(T value)` bir metot, `Container`'ın `T`'sine başvurabilir — statik bir bağlamın kastedebileceği tek bir `Container` instance'ının `T`'si yoktur. Statik bir metodun YAPABİLECEĞİ şey, tam olarak "Generic Metotlar"da işlendiği gibi, kendi, tamamen bağımsız tür parametresini bildirmektir — `singletonList`'in `U`'sunun `Container`'ın `T`'siyle hiçbir ilgisi yoktur.

Çalışma Zamanı Kısıtları Pratikte

Şimdiye kadar işlenen kısıtlar yalnızca teorik değil — bir raw type (hiç tür argümanı olmadan kullanılan bir generic tür), derleme-zamanı kontrolünü tamamen atlatabilir, tam olarak generics-öncesi kodun her zaman yaptığı gibi.

```
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(...)`, raw bir `List` alır, bu yüzden derleyici bu serinin geri kalanının dayandığı tür kontrolünün hiçbirini uygulamaz — gerçekte bir `List<Integer>` olan bir şeye bir `String` eklemek sorunsuz derlenir. Ama hata eklemede gerçekleşmez; daha sonra, okumada, derleyicinin eklediği `Integer`'a cast sonunda çalışıp bir `ClassCastException` fırlattığında gerçekleşir — gerçek hatanın yapıldığı yerden çok uzakta. Bu, tam olarak "Generics'e Giriş" in bu serinin tamamını açtığı durum, bir raw type kullanıldığında bugün hâlâ erişilebilir.

Best Practices

- Yazdığın kodda asla bir raw type kullanma — kullandığın anda, tam olarak `pollute(...)` 'un gösterdiği gibi, bu serinin işlediği her derleme-zamanı garantisini kaybedersin.

- Generic bir metodun gerçekten bir `T` inşa etmesi gerektiğinde, `new T()` yapmaya çalışmak yerine çağırandan bir factory (`Supplier<T>` gibi) kabul et.
- İçeride generic bir array inşa etmen gerekirse, altta yatan `Object[]`'i tamamen private tut, ve onun aracılığıyla yalnızca tekil `T` elemanlarını açığa çıkar, asla raw array'in kendisini değil.
- Sınıf seviyesi durumun gerçekten gerekmediği durumlarda kendi tür parametresine sahip statik bir generic metoda başvur — bu, statik/generics kısıtını tamamen atlar.

Yaygın Hatalar

- `new T()` ya da `new T[size]` yazıp derleyici hatasını erasure'ın doğrudan bir sonucu olarak tanımak yerine kafa karıştırıcı bulmak.
- `instanceof List<String>` denemek ve çalışmasını beklemek, tek geçerli form olan `instanceof List<?>` yerine.
- Bir sınıfın kendi tür parametresine başvuran statik bir alan ya da metot bildirip, statik bağlamın başvurabileceği belirli bir instance'ın `T`'si olmadığını fark etmemek.
- Bir "unchecked" derleyici uyarısını, `UncheckedWarningHeapPollutionExample`'ın gösterdiği türden erasure'a bağlı güvensizliği işaret ederken, sıradan bir gürültü olarak görmezden gelmek.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Type erasure, derleme-zamanı kontrolünden sonra tür argümanlarını kaldırır, bu yüzden generic tür bilgisi çalışma zamanında var olmaz.
- Erasure, generic kodun Java 5'ten önce var olan generics-öncesi kod ve bytecode ile birlikte çalışmasına izin vermek için var.
- Farklı tür argümanlarına sahip iki koleksiyon aynı çalışma zamanı sınıfını paylaşır; yalnızca raw bir `instanceof` kontrolü geçerlidir.
- `new T()` ve `new T[]` derlenmez, çünkü erasure, JVM'in çalışma zamanında instance oluşturacağı gerçek bir sınıf bırakmaz.
- Bir sınıfın statik üyeleri onun tür parametresine başvuramaz, çünkü statik bir bağlamın kullanacağı belirli bir instance'ın tür argümanı yoktur.

Cheat Sheet

```
// Çalışma zamanı erasure'ı
List<String> a = new ArrayList<>();
List<Integer> b = new ArrayList<>();
a.getClass() == b.getClass(); // true

// instanceof: yalnızca raw form geçerli
if (obj instanceof List<?>) { ... }

// new T() geçici çözümü: çağıran bir factory sağlar
static <T> T createDefault(Supplier<T> factory) { return factory.get(); }

// Generic array geçici çözümü (yalnızca sınıf içinde)
@SuppressWarnings("unchecked")
T[] elements = (T[]) new Object[10];

// Statik üyeler sınıfın T'sini kullanamaz, ama KENDİ tür parametresini bildirebilir
static <U> List<U> singletonList(U value) { ... }
```

Terimler Sözlüğü

- **Type erasure (tür silme)**: derleyicinin generics'i uygulama stratejisi -- tür argümanlarını derleme zamanında kontrol edip, çalışma zamanından önce atmak.
- **Raw type**: hiç tür argümanı olmadan kullanılan, generics'in derleme-zamanı kontrolünün hiçbirini almayan bir generic tür.
- **Heap pollution**: parametrelenmiş bir türden bir değişkenin, gerçekte o parametrelenmiş türden olmayan bir nesneye başvurduğu, genelde bir raw type ya da unchecked bir cast aracılığıyla ortaya çıkan durum.
- **Unchecked uyarısı**: derleyicinin erasure yüzünden tür-güvenli olduğunu tam olarak doğrulayamadığı bir cast ya da işlemi işaretleyen derleyici uyarısı.