C# Generic Yapılar Örnekli Anlatım

3 min read · Mar 29, 2025

Süleyman Meral

Herkese merhaba! Bu yazımızda C# üzerinde kullanılan generic yapılardan bahsedeceğiz. Bu yapılara gerçek bir proje üzerinden örnek vereceğiz.

Generic Listeler

C# dilinde **Generic Listeler**, veri türüne bağlı olmaksızın farklı türdeki verileri tutabilen, tür güvenliğini sağlayan ve esnek bir koleksiyon yapısı sunan bir özelliktir. C#'ta `List<T>` sınıfı, koleksiyonların en yaygın kullanılan ve temel yapı taşlarından biridir. `T` burada herhangi bir veri türünü temsil eder ve **generic** yapısı sayesinde, veri türünü çalışma zamanında değil, derleme zamanında belirlemenize olanak tanır.

Generic liste, belirli bir türdeki verileri depolamak için kullanılan ve **tür güvenliği** sağlayan dinamik bir koleksiyon yapısıdır. Örneğin, `List<int>`, sadece `int` türündeki verileri tutarken, `List<string>` sadece `string` türündeki verileri tutar. Bu sayede yanlış türde veri eklenmesi engellenir ve çalışma zamanı hataları önlenir.

`ArrayList` gibi eski koleksiyonlar **`object` tabanlı** çalıştığı için **değer tipleri (`int`, `double`, `bool` vb.) eklenirken boxing, geri alınırken unboxing olur.**

```
ArrayList list = new ArrayList();

        int number = 42;
        list.Add(number); // Boxing (int -> object)

        int extractedNumber = (int)list[0]; // Unboxing (object -> int)
```

```
        Console.WriteLine(extractedNumber);
```

`int` değerini `object` olarak sakladık (**heap belleğe taşındı**).

`object` tekrar `int`'e çevrildi (**cast işlemi yapıldı**).

Generic `List<T>` **tipi baştan belirlediği** için **boxing-unboxing işlemi olmaz** ve **daha performanslı çalışır.**

Heap yerine stack kullanıldığı için bellek kullanımı daha verimlidir.

Tip güvenliği sağlar ve `InvalidCastException` gibi hataların önüne geçer.

```
List<int> numbers = new List<int>();
        numbers.Add(42); // Boxing yok!

        int extractedNumber = numbers[0]; // Unboxing yok!

        Console.WriteLine(extractedNumber);
```

İnt ve String türünde 2 adet liste oluşturalım. Ve bu liste elemanlarını ekrana yazdıralım.

```
List<int> intNumbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
List<string> isimler = new List<string> { "Süleyman", "Beyza"};

foreach(var item in intNumbers)
{
    Console.WriteLine(item);
}

foreach (var item in isimler)
{
    Console.WriteLine(item);
}
```

Yukarıda görüldüğü gibi 2 listemizin elemanlarını da yazdırdık. Fakat görüldüğü gibi 2 adet döngü kullandık. Eğer elimizde onlarca veya yüzlerce liste olsaydı hepsi için ayrı ayrı döngü yazmak zorunda kalacaktık. Ve bu listelerin elemanlarının çok fazla olacağını düşünürsek programın üzerine çokça yük binecekti, çok fazla gereksiz kod yazacaktık. Peki bunun önüne nasıl geçebiliri? Generic method yardımı ile farklı türlerde olan listeler ile işlem yapabiliyoruz.

```
public void Print<T>(List<T> list)
{
    foreach (var item in list)
    {
        Console.WriteLine(item);
    }
}
```

Print methodumuz T türünde bir liste istiyor. Yani tür ayırt etmeksizin tüm türlerle çalışabilir. Bizde bu fonksiyonumuzu kullanarak elemanlarımızı yazdırabiliriz.

```
List<int> intNumbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
List<string> isimler = new List<string> { "Süleyman", "Beyza";

GenericClass gnr = new GenericClass();

gnr.Print(intNumbers);
gnr.Print(isimler);
```

Görüldüğü gibi tek bir method ile çözüm ürettik.

Generic Interfaceler

Generic interface'ler, generic sınıflarla çalışabilen ve birden fazla türde implementasyon yapmanıza olanak sağlayan interface yapılarıdır.

```
public interface IGenericService<T>
  {
      void TAdd(T t);
      void TDelete(T t);
      void TUpdate(T t);
```

```
        List<T> TGetList();
        T TGetBYID(int id);

        List<T> TGetListbyFilter();

    }
```

Yukarıda bir generic interface yapısı görmekteyiz. İçinde generic türden methodlar da bulunduruyor. Kişisel bilgilerimizin olduğu bunlara veri ekleyip silebildiğimiz,güncelleyebildiğimiz bir web veya mobil uygulamamız olduğunu düşünelim. Backend kısmı için veri tabanında varlıklar oluşturduk, bu varlıkların hepsi birer classı temsil eder.

Örnek verecek olursak yetenek işlemlerimiz için skill adında bir class oluşturmuş olalım ve bunun gibi bir sürü classımız olduğunu varsayalım.

```
 public class Skill
   {
       [Key]
       public int SkillID { get; set; }
       public string? Title { get; set; }
       public int Value{ get; set; }
       public string? imageUrl{ get; set; }
   }
```

Bu elemanları listelemek güncellemek silmek için ayrı ayrı bir sürü fonksiyon yazmamız gerekirdi. Fakat generic yapılar ile bunun önüne geçeceğiz.

```
public interface IGenericService<T>
   {
       void TAdd(T t);
       void TDelete(T t);
       void TUpdate(T t);
       List<T> TGetList();
       T TGetBYID(int id);

       List<T> TGetListbyFilter();

   }
```

Bir generic service interfaci oluşturduk ve işlemlerimizi buraya entegre ettik. Bu işlemler skill gibi bütün varlıklarımızda yapılacak. Bunları varlıklar için ayrı ayrı oluşturmak yerine her türden classı kabul eden bir interface yazdık.

```
public interface ISkillService:IGenericService<Skill>
    {
    }
```

İlgili interface 'imize bu yapıdan miras aldırdık. Bu sayede tipi ne olursa olsun IGenericService interface'i tüm entity işlemlerine uygulanabilir oldu.

Ayrıca Generic yapılarımıza kısıtlama ekleyebiliyoruz.

```
  public interface IGenericDAL<T> where T : class
   {
       void Insert(T t);
       void Delete(T t);
       void Update(T t);
       List<T>GetList();
       T GetById(int id);

       List<T> GetbyFilter(Expression<Func<T, bool>> filter);
   }
```

Burada IGenericDAL interface'imizin sadece class tipinden referans türleri ile çalışması gerektiğini belirledik.

Generic Classlar

- Generic sınıflar, belirli bir tür belirtilmeden kullanılabilen sınıflardır. Tür parametreleri ile çalışarak esneklik sağlarlar.

```
public class GenericClass<T>
{
    public void save(T obj)
    {
        Console.WriteLine( " database kaydedildi."+obj.GetType());
```

```
    }
}
```

```
GenericClass<Person> person = new GenericClass<Person>();
person.save(new Person { ad = "Süleyman", soyad = "Meral", departman = "IT" });


GenericClass<Car> car = new GenericClass<Car>();
car.save(new Car { marka="Cupra" });
```

Görüldüğü gibi tek bir Class yapısı ile 2 türden nesne üretebildik.

Yazılım

Edit profile

Written by Süleyman Meral

6 followers · 6 following

Backend Developer(.NET Core)

No responses yet

Süleyman Meral

What are your thoughts?