

.NET CORE Reflection(Runtime'da Kodu Okumak ve Değiştirmek: Reflection'ın Gücü)

7 min read · Apr 14, 2025



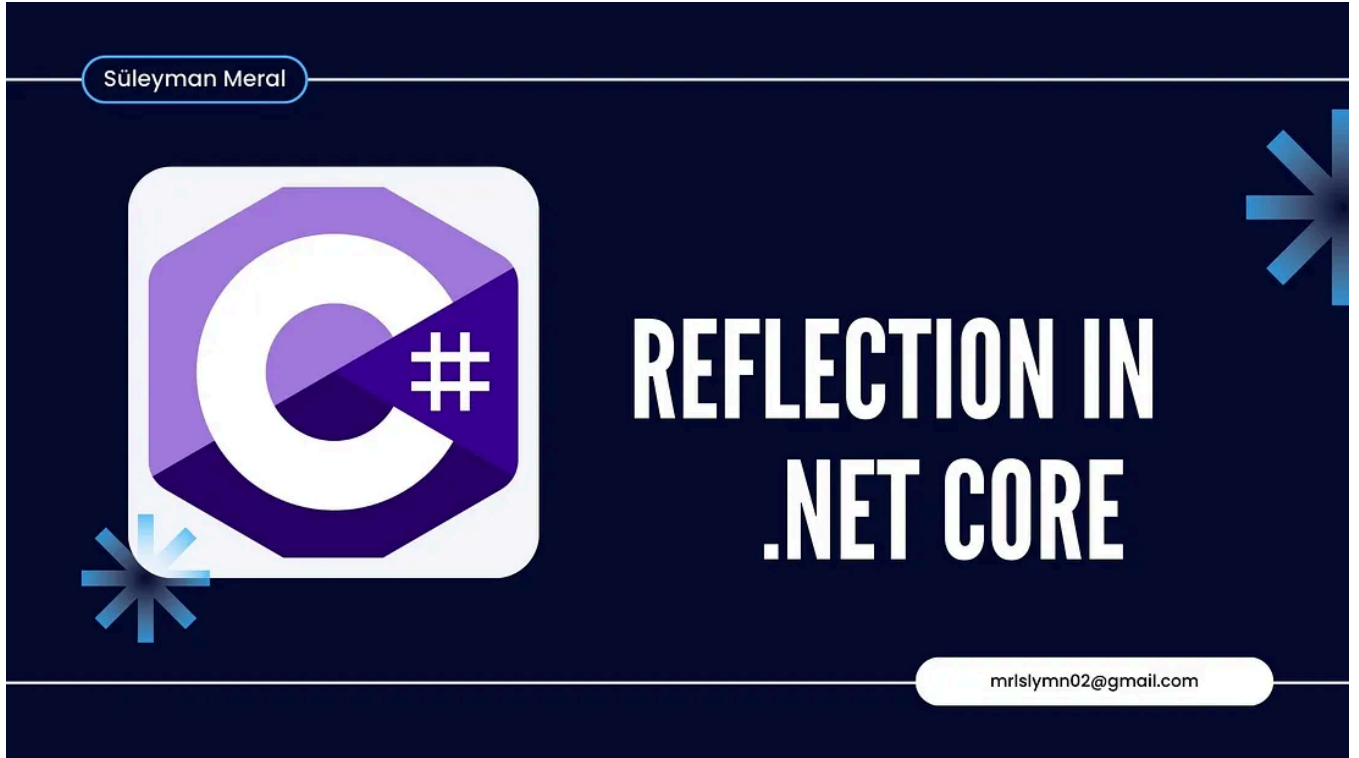
Süleyman Meral



Share



More



Herkese merhaba! Bu yazımda sizlere Reflection kavramından bahsedeceğim. Reflection Nedir? Niçin Kullanılır? Avantajları Ve Dezavantajları Nelerdir ? Bu sorular üzerinde duracağız ve bir örnek de yapacağız.

Reflection Nedir?

C# dilinde **Reflection**, bir programın çalışma zamanında (runtime) kendi yapısını (tiplerini, metodlarını, özelliklerini, vb.) incelemesine ve hatta bazı durumlarda değiştirmesine olanak sağlayan bir özelliktir.

Reflection, canlıda kodu analiz edip müdahale etmemizi sağlayan bir “ayna” gibidir.

Kodun kendisini **çalışma zamanında okuyabilen**, anlayan ve değiştiren sistemler yapmak istiyorsak, **reflection** vazgeçilmezdir.

Reflection Niçin Kullanılır?

- Çalışma zamanında nesnelerin yapısını öğrenmek
- Metot çağırmak (invoke) (Oyun Programlamada Çokça Kullanılır)
- Özelleştirilmiş davranışlar yazmak (örneğin plugin sistemi)
- Attribute'lara göre özel iş kuralları
- Test frameworkleri, ORM araçları, web frameworkleri

Avantajları Ve Dezavantajları Nelerdir?

Avantajları

- Kodun çalışma zamanında analiz edilmesini ve çalıştırılmasını sağlar. Bu sayede dinamiklik sağlar.
- Sınıfın, metodun, property'nin isimleri, türleri gibi bilgiler runtime'da erişilebilir.
- Özelleştirilmiş davranışlar attribute'larla işaretlenip kullanılabilir.

Dezavantajları

- Performansı düşüktür. Doğrudan kod çalıştırmaya göre çok yavaştır.
- Hatalar ancak uygulama çalıştırıldığında (runtime'da) ortaya çıkar.
- Kodun okunabilirliğini düşürebilir.
- Güvenlik riskleri doğurabilir

Şimdi ise bir reflection örneği yapalım.

Person adında bir class oluşturalım ve içine Name isimli bir property ve Hello isimli bir method ekleyelim.

```
public class Person
{
    public string? Name { get; set; }
```

```
public void Hello()
{
    Console.WriteLine($"Hello,{Name}");
}
}
```

Şimdi ise Reflection kullanarak runtime'da yani kod çalışıyor iken Person classının Methodlarına erişmeye çalışalım.

```
Type type = typeof(Person);
```

`typeof(Person)` => Person sınıfının tipini (Type objesi) alınır.

Type sınıfı, C#'ta bir nesnenin veya sınıfın yapısını temsil eder.

Person sınıfının yansımasını (reflection) kullanmak üzere bu tip bilgisi alınıyor.

```
MethodInfo[] methods = type.GetMethods();
foreach (var method in methods)
{
    Console.WriteLine("Method: " + method.Name);
}
```

MethodInfo türünde methods isimli bir dizi oluşturduk ve bu dizinin elemanları type objesinin methodları olarak belirlendi.

Daha sonra foreach kullanarak bu methodları yazdırdık.

MethodInfo[] dizisi, her metodun bilgilerini içerir (isim, parametreler vs.).

```
Method: get_Name
Method: set_Name
Method: Hello
Method: GetType
Method: ToString
Method: Equals
Method: GetHashCode
```

Kodumuzu run ettiğimizde böyle bir çıktı elde edeceğiz. Görüldüğü üzere methodlar teker teker listelendi. Bazı methodlar C# da classlara default olan methodlardır. Örneğin daha iyi anlaşılabilmesi açısından classımıza bir property ve bir method daha ekleyelim ve çıktıya bakalım.

```
public class Person
{
    public string? Name { get; set; }
    public decimal Salary { get; set; }

    public void Hello()
    {
        Console.WriteLine($"Hello,{Name}");
    }
    public void Calculate()
    {
        Console.WriteLine($"Hello,{Salary}");
    }
}
```

```
Method: get_Name
Method: set_Name
Method: get_Salary
Method: set_Salary
Method: Hello
Method: Calculate
Method: GetType
Method: ToString
Method: Equals
Method: GetHashCode
```

Eklediğimiz method ve property ile ait get ve set methodları da eklendi.

Şimdi ise runtime da Person sınıfından bir nesne oluşturup değer ataması yapalım.

```
object personInstance = Activator.CreateInstance(type);
```

`new Person();` ama reflection kullanılarak yapılır.

Runtime da Person classından personInstance isimli bir instance oluşturduk.

```
PropertyInfo prop = type.GetProperty("Name");  
prop.SetValue(personInstance, "Süleyman");
```

PropertyInfo türünde prop isimli bir değişken oluşturduk ve Name propertyisine atadık. Daha sonra SetValue ile Propertyimize bir değer atadık.

`GetProperty("Name")` ⇒ Person sınıfındaki Name özelliğini bulur.

PropertyInfo , o özelliğin bilgilerini içerir.

Dinamik oluşturulan personInstance nesnesinin Name özelliğine "Süleyman" değerini atar.

`person.Name = "Süleyman";` Bu anlama gelir fakat reflection kullanarak yapılmıştır.

```
MethodInfo HelloMethod = type.GetMethod("Hello");  
HelloMethod.Invoke(personInstance, null);
```

`GetMethod("SayHello")` ⇒ SayHello metodunu bulur.

`Invoke(...)` ⇒ bu metodu çağırır. (Bu yapı unity de çokça kullanılır)

`null` ⇒ metodun parametresi olmadığı için boş gönderildi.

```
Method: get_Name
Method: set_Name
Method: get_Salary
Method: set_Salary
Method: Hello
Method: Calculate
Method: GetType
Method: ToString
Method: Equals
Method: GetHashCode
Hello, Süleyman
```

Kodumuzun çıktısı bu şekilde görüntülenmektedir. Aynı şekilde Salary prop'una runtime da değer atayıp Calculate fonksiyonunu çağırabiliriz.

```
using ReflectionOrnek;
using System.Reflection;

Type type = typeof(Person);

MethodInfo[] methods = type.GetMethods();
foreach (var method in methods)
{
    Console.WriteLine("Method: " + method.Name);
}

object personInstance = Activator.CreateInstance(type);

PropertyInfo prop = type.GetProperty("Name");
prop.SetValue(personInstance, "Süleyman");

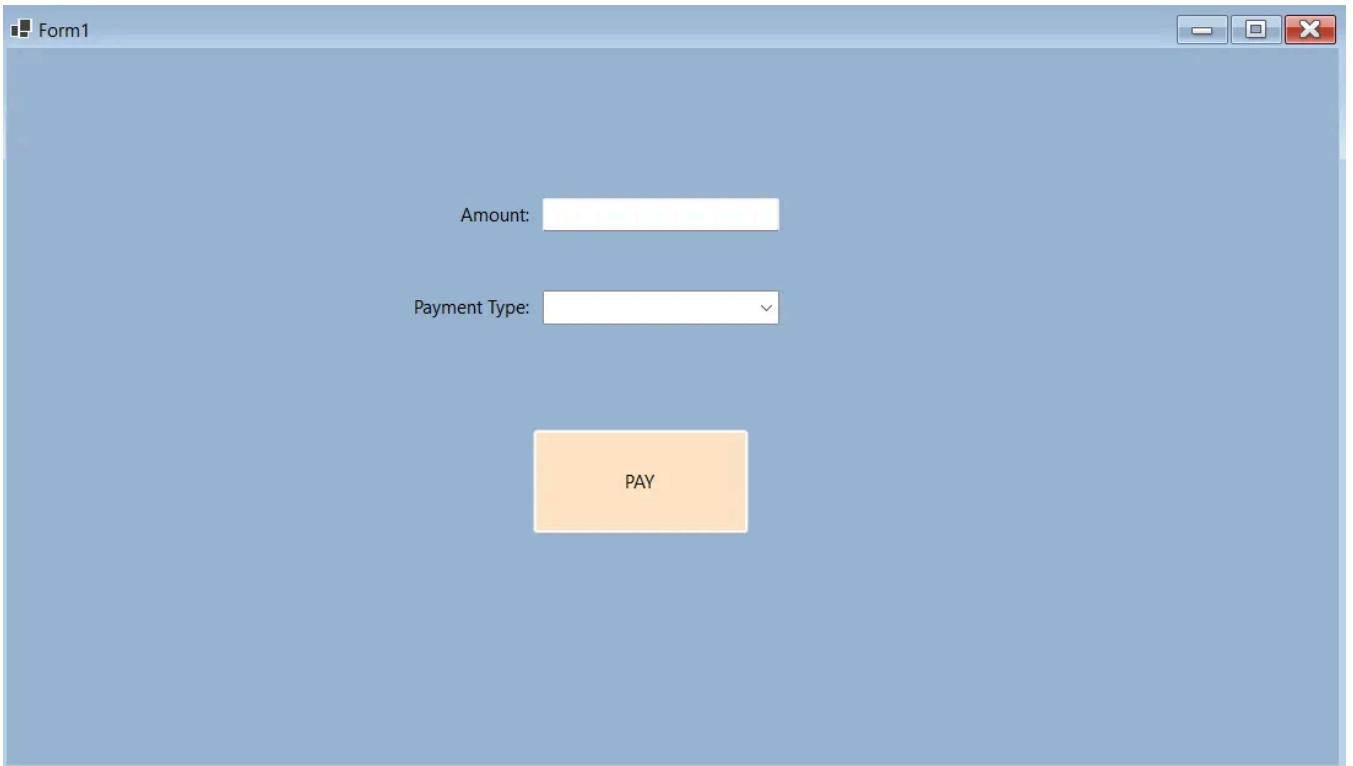
MethodInfo HelloMethod = type.GetMethod("Hello");
HelloMethod.Invoke(personInstance, null);
PropertyInfo salary = type.GetProperty("Salary");
salary.SetValue(personInstance, (decimal)80000.00);

MethodInfo calculate = type.GetMethod("Calculate");
calculate.Invoke(personInstance, null);
```

```
Method: get_Name  
Method: set_Name  
Method: get_Salary  
Method: set_Salary  
Method: Hello  
Method: Calculate  
Method: GetType  
Method: ToString  
Method: Equals  
Method: GetHashCode  
Hello, Süleyman  
Hello, 80000  
  
C:\Users\suley\source\repos\ReflectionOrnek\ReflectionOrnek\bin\Debug\net8.0\ReflectionOrnek.exe (process 4680) exited with code 0 (0x0).  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .|
```

Şimdi ise konuyu daha iyi pekiştirmek için günlük hayata dair bir örnek yapalım.

Bir windows form uygulaması oluşturalım. Daha sonra basit bir ödeme ekranı tasarlayalım.

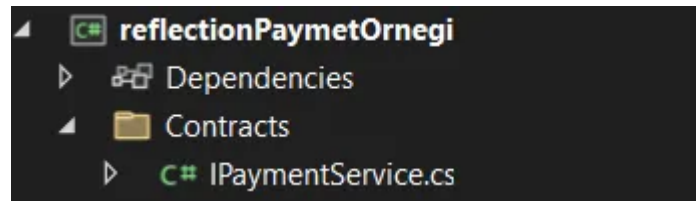


Basit bir senaryomuz olsun. Kullanıcı alışveriş tutarını girsin Ödeme tipini seçsin. Daha sonra butona bassın ve ödeme işlemini gerçekleştirsin.

Ödeme yöntemlerimiz Kredi Kartı, EFT, Havale gibi işlemler olabilir. Peki bu yapıyı nasıl oluşturmamız? Veya reflection bu senaryoda ne işimize yarayacak inceleyelim.

Ödeme yöntemlerimizin hepsinin ortak bir yapısı bulunmakta. Hepsi ödeme işlemini gerçekleştiriyor. Bu yüzden bunları ortak bir interfaceden türetebiliriz. Bu

şekilde her ödeme yöntemi classımıza ayrı bir metod yazmamıza gerek kalmaz, aynı zamanda yeni bir ödeme yöntemi eklemek çok kolay olur.



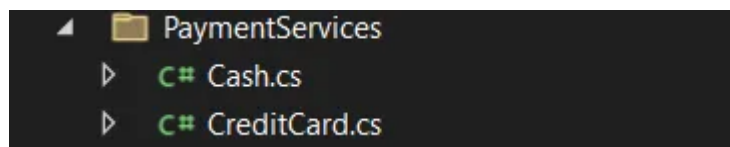
Contracts isimli bir klasör oluşturalım. Bu klasör bizim interfacerimizi barındırsın. IPaymentService adında bir interface oluşturuyoruz.

```
public interface IPaymentService
{
    string Pay(decimal amount);
}

// sadece govdesiz method barındırır
```

Geri dönüş tipi string olan ve decimal türünde amount adında bir parametre alan Pay methodunu oluşturuyoruz. Bildiğimiz üzere interface içindeki metodlar sadece imza içerir. Gövdeleri bu interfaceden türetilen ilgili classlarda override edilmelidir.

Pay metodumuz tüm ödeme yöntemleri için ortak olacak. Bu yüzden başlangıç için bu interfaceden(arayüz) türeyen 2 adet ödeme yöntemi classı oluşturalım.



Payment services klasörü içinde Cash ve CreditCard isimli 2 adet class oluşturduk. Bu 2 class IPaymentService arayüzünden türetildi.

```
public class Cash : IPaymentService
{
    public string Pay(decimal amount)
    {
        return "Payment with cash is successfull. Amount:" + amount;
    }
}
```



```
}  
}
```

```
public class CreditCard : IPaymentService  
{  
    public string Pay(decimal amount)  
    {  
        return "Payment with credit card is successfull. Amount:"+amount;  
    }  
}
```

IPaymentService arayüzünde bulunan Pay metodunun gövdelerini oluşturduk.

Şimdi ise ödeme yöntemlerimizi arayüzümüzde bulunan combobox componentimize entegre edelim.

Bunun için formun yüklenme fonksiyonunu kullanacağız. Form yüklendiği anda ödeme yöntemlerinin hepsi combo box componentine gelecek. Peki bu işlem nasıl gerçekleşecek. Reflection tam da bu kısımda devreye giriyor.

İşlemimizi öncelikle reflection kullanmadan gerçekleştirelim.

```
cmbPaymentType.Items.Add("CreditCard");  
cmbPaymentType.Items.Add("Cash");  
  
IPaymentService paymentService;  
  
if (cmbPaymentType.SelectedItem.ToString() == "CreditCard")  
{  
    paymentService = new CreditCard();  
}  
else if (cmbPaymentType.SelectedItem.ToString() == "Cash")  
{  
    paymentService = new Cash();  
}  
  
else  
{  
    throw new Exception("Invalid payment type");  
}
```

Veriler statik olarak combo boxa her seferinde eklenmek zorunda. Ve hangi sınıftan örnek oluşturulacağı if else bloklarıyla tüm ödeme yöntemleri için kontrol edilmek zorunda. Aynı zamanda yeni bir ödeme yöntemi eklersek birçok yerde değişiklik yapmamız gerekecek. Buda kodumuzdaki karmaşıklığı arttıracak. Okunabilirliğini azaltacak. Ayrıca bu yöntem SOLID'in open/closed ilkesine aykırı bir yöntem olmaktadır. Yani yeni bir özellik , yeni bir ödeme yöntemi eklemek için kodumuzu değiştirmememiz gerekmektedir. Bu yöntem ile OCP yi ihlal ediyoruz.

Şimdi daha düzenli ve esnek ayrıca OCP ye uyumlu bir kod yazmak için reflection'ı kullanalım.

```
var paymentMethods = AppDomain.CurrentDomain.GetAssemblies()
    .SelectMany(x => x.GetTypes())
    .Where(t => typeof(IPaymentService).IsAssignableFrom(t) && !t.IsInterface) &
    .ToList();

foreach( var type in paymentMethods )
{
    cmbPaymentType.Items.Add(type);
}

cmbPaymentType.DisplayMember = "Name";
```

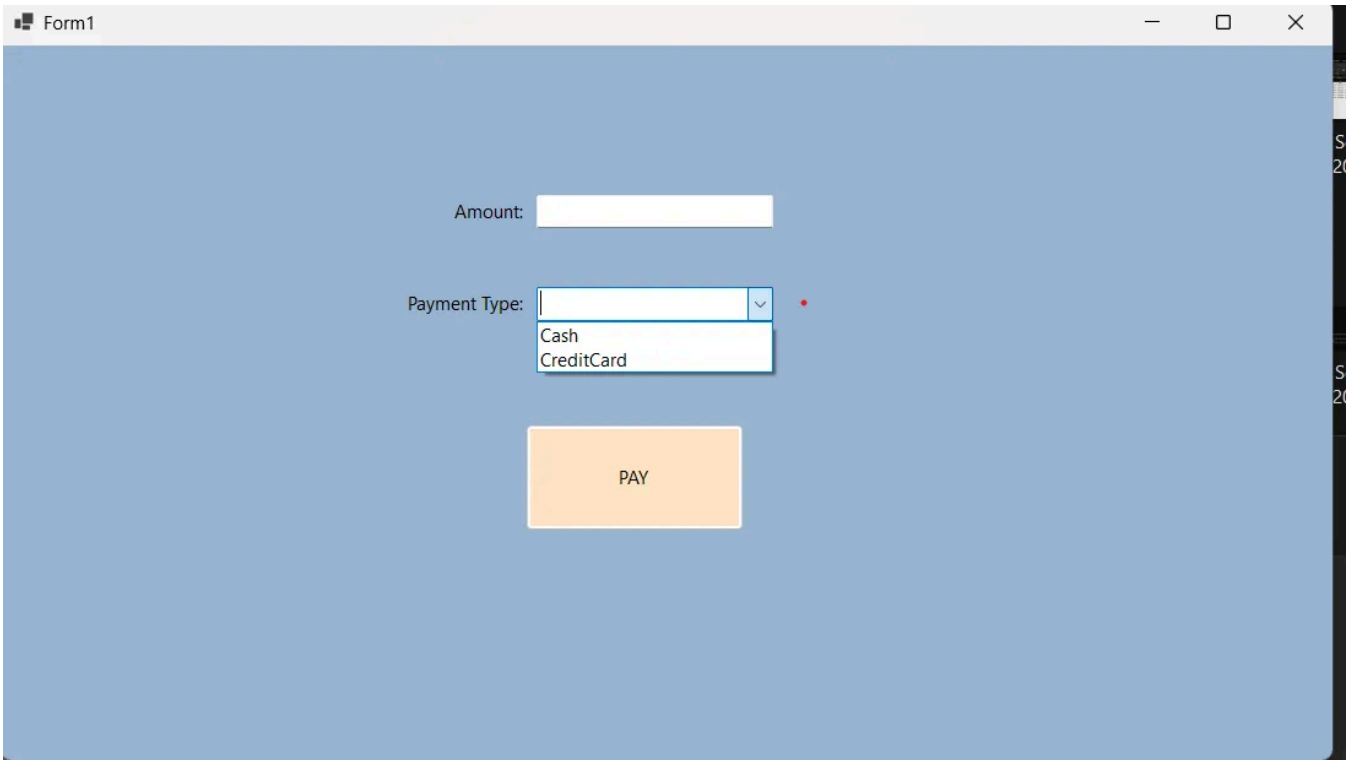
Form yüklendiğinde projemizdeki IPaymentService arayüzünü implement eden veya bu arayüzden türeyen tüm classları paymentMethods değişkenine atıyoruz.

`typeof(IPaymentService).IsAssignableFrom(t)` → “t tipi, IPaymentService’den türemiş mi veya onu implement ediyor mu?

Daha sonra foreach döngüsü ile bu classları combo boxa ekliyoruz.

```
cmbPaymentType.DisplayMember = "Name";
```

Satırı ile type nesnesinin sadece Name içeriğini alıyoruz.



Uygulamamızı başlattığımızda IPaymentService arayüzünden türeyen classlarımızın ismi combo box' a eklendi.

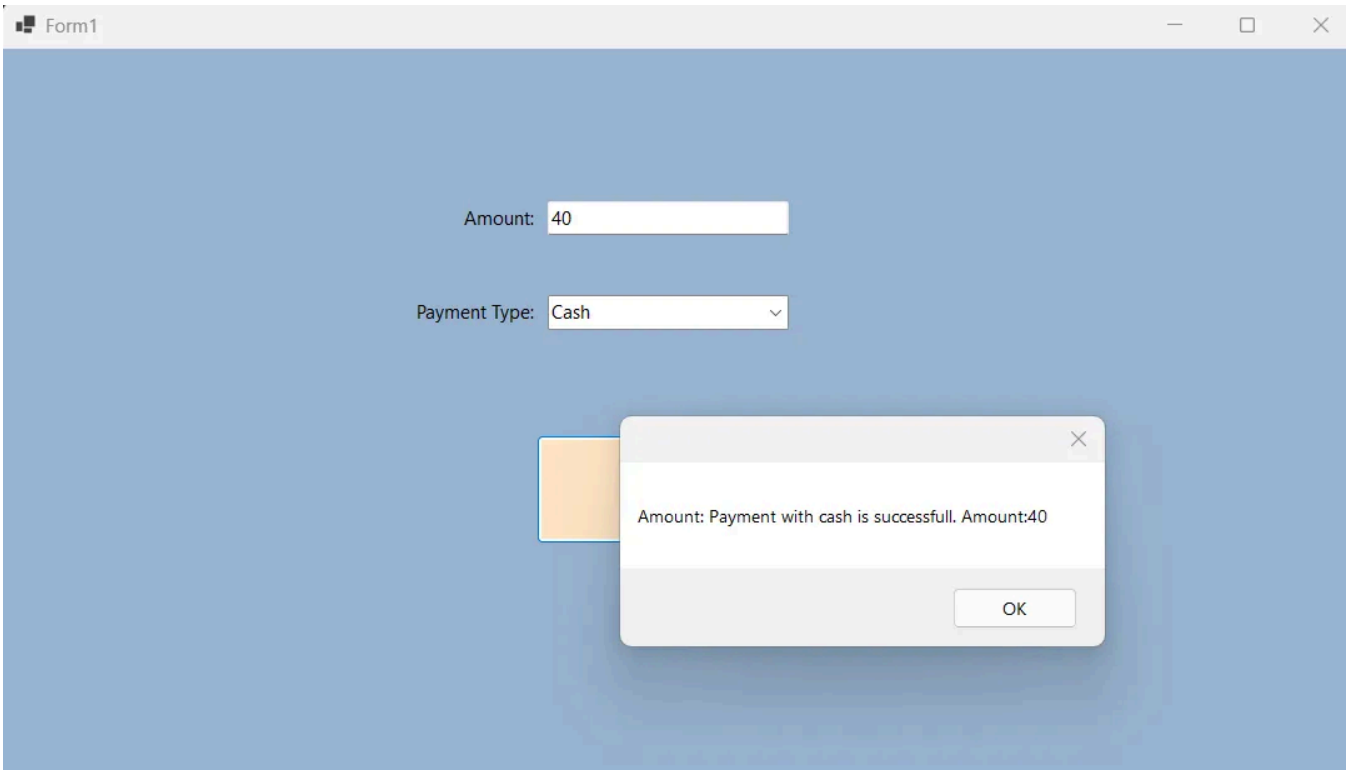
Hiçbir if else kontrolü yapmadık ve statik olarak veri girmedik.

Şimdi ödeme butonumuza ait kodları yazalım.

```
private void btnPay_Click(object sender, EventArgs e)
{
    int amount = int.Parse(txtAmount.Text);

    if(cmbPaymentType.SelectedItem is Type selectedType)
    {
        IPaymentService paymentMethod = (IPaymentService)Activator.CreateInstance
        MessageBox.Show($"Amount: {paymentMethod?.Pay(amount)} ");
    }
}
```

Combo boxdan seçilen değer selectedType değişkenine atandı. Ve ilgili classdan CreateInstance metodu ile bir örnek oluşturuldu. Her ödeme yöntemi için newleme işlemi yapmak zorunda kalmadık.



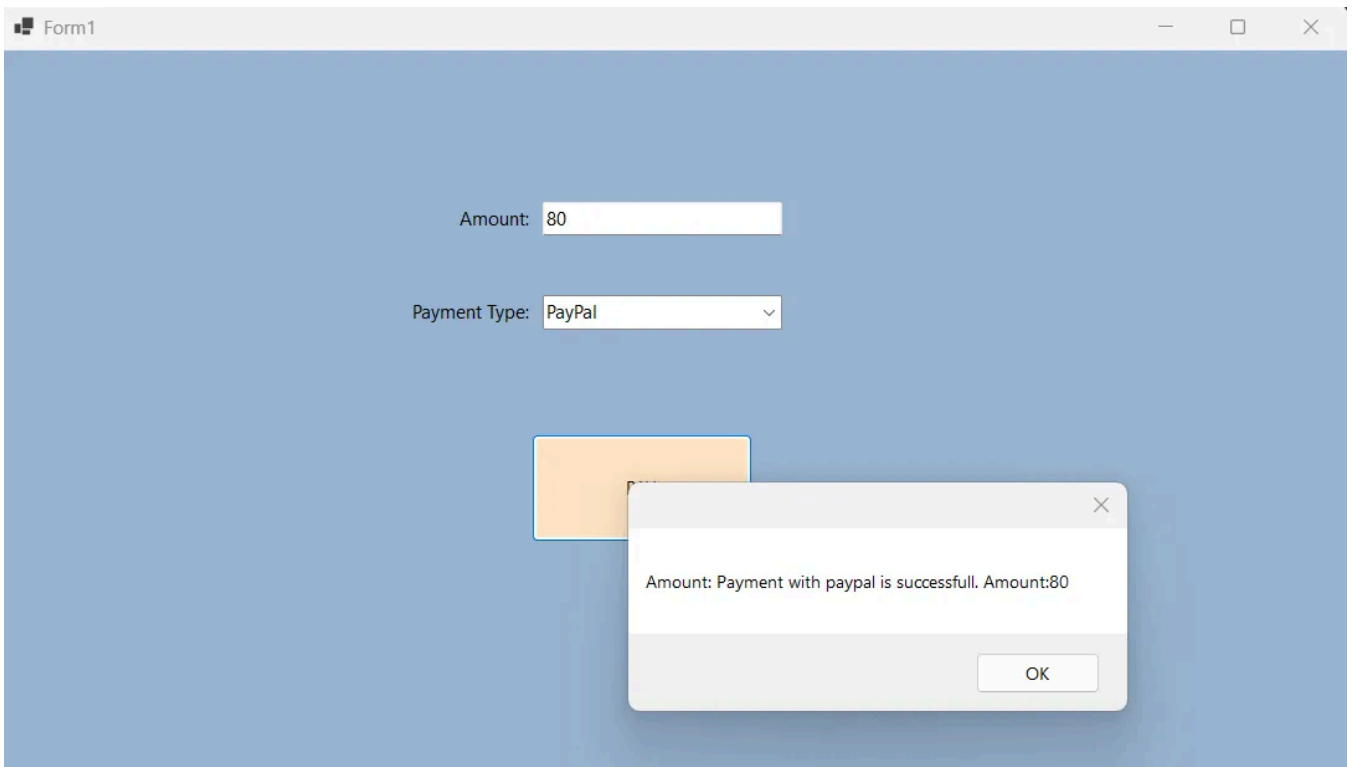
Ödeme işlemi bu şekilde gerçekleşmiş oldu. Şimdi ise senaryomuza bir ekleme daha yapalım. Ekip liderimiz bizden projeye yeni bir ödeme yöntemi olan “PayPal” ile ödeme yöntemini eklememizi istemiş olsun.

Reflection kullandığımız için bu işlemi ödeme kodlarını değiştirmeden kolay bir şekilde gerçekleştireceğiz.

Services kalsörüne PayPal adında IPaymentService arayünden türeyen bir class ekliyoruz. Arayüzü implement ediyoruz ve gövdesini oluşturuyoruz.

```
public class PayPal : IPaymentService
{
    public string Pay(decimal amount)
    {
        return "Payment with paypal is successfull. Amount:" + amount;
    }
}
```

Ödeme işlemini yapan kodlarımızı hiç değiştirmeden uygulamamızı başlatalım.



PayPal otomatik olarak combo box'a eklendi ve ödeme işlemi gerçekleştirildi. Bu sayede SOLID'in OCP (Open/Closed Principle) prensibini çiğnememiş olduk.

- Yeni bir özellik eklemek için mevcut kodu **değiştirmedik**.
- Bunun yerine yeni sınıf ekleyerek **genişlettik**.

Eğer refleciton kullanmasaydık aşağıdaki gibi kodu değiştirip solidi ezmiş olacaktık.

```
cmbPaymentType.Items.Add("CreditCard");
cmbPaymentType.Items.Add("Cash");
cmbPaymentType.Items.Add("PayPal");
IPaymentService paymentService;

if (cmbPaymentType.SelectedItem.ToString() == "CreditCard")
{
    paymentService = new CreditCard();
}
else if (cmbPaymentType.SelectedItem.ToString() == "Cash")
{
    paymentService = new Cash();
}
else if (cmbPaymentType.SelectedItem.ToString() == "PayPal")
{
    paymentService = new PayPal();
}

else
```

```
{  
    throw new Exception("Invalid payment type");  
}
```

Reflection kavramını bu şekilde ele aldık. Daha detaylı bir inceleme için YouTube kanalımda 2 adet video yükledim . Aşağıdaki linkten kanalıma ulaşabilirsiniz.
Herkes Kolay Gelsin !



<https://www.youtube.com/@suleymanmeral53/>

Yazılım Gelistirme

Yazılım

Kodlama



Edit profile

Written by Süleyman Meral

6 followers · 6 following

Backend Developer(.NET Core)

No responses yet



Süleyman Meral

What are your thoughts?