

Görsel Programlama

Ders 7

Dr. Öğr. Üyesi Mehmet Dinçer Erbaş
Bolu Abant İzzet Baysal Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü

Kapsülleme

- Kapsülleme, “bir veya daha fazla öğeyi fiziksel veya mantıksal bir paket içine alma işlemi” olarak tanımlanır.
 - Nesne yönelimli programlama metodolojisinde kapsülleme, uygulama ayrıntılarına erişimi engeller.
- Soyutlama ve kapsülleme, nesne yönelimli programlamadaki ilgili özelliklerdir.
 - Soyutlama, ilgili bilgilerin görünür hale getirilmesine izin verir.
 - Kapsülleme, bir programcının istenen soyutlama düzeyini uygulamasına olarak tanır.

Kapsülleme

- Kapsülleme, erişim belirteçleri kullanılarak gerçekleştirilir.
 - Erişim belirteci, bir sınıf üyesinin kapsamını ve görünürlüğünü tanımlar.
 - C # aşağıdaki erişim belirteçlerini destekler
 - Public
 - Private
 - Protected
 - Internal
 - Protected internal

Kapsülleme

- Public Erişim Belirteci
- Genel erişim belirteci, bir sınıfın üye özelliklerini ve üye fonksiyonlarını diğer fonksiyonlara ve nesnelere göstermesine izin verir.
 - Herkese açık herhangi bir üyeye sınıf dışından erişilebilir.
 - Örneğimizi inceleyelim: Örnek1
 - Örnekte, uzunluk ve genişlik üye değişkenleri public olarak bildirilmiştir, bu nedenle bunlara r adlı Rectangle sınıfının bir nesnesi kullanılarak Main () fonksiyonundan erişilebilir.
 - Fonksiyonlar Display () ve GetArea (), sınıfın herhangi bir nesnesini kullanmadan bu değişkenlere doğrudan erişebilir.
 - Display () fonksiyonu da public olarak bildirilmiştir, bu nedenle Main () 'den r adında Rectangle sınıfının bir nesnesi kullanılarak da erişilebilir.

Kapsülleme

- Private Erişim Belirteci
- Private (özel) erişim belirteci, bir sınıfın üye özelliklerinin ve üye fonksiyonlarının diğer fonksiyonlardan ve nesnelerden gizlemesine izin verir.
 - Yalnızca aynı sınıfın fonksiyonları, özel üyelerine erişebilir.
 - Bir sınıfın bir örneği bile özel üyelerine erişemez.
- Örneğimizi inceleyelim: Ornek2
- Örneğimizde length ve width üye özellikleri özel olarak bildirilmiştir, bu nedenle Main () fonksiyonundan erişilemezler.
- Üye fonksiyonları AcceptDetails () ve Display () bu değişkenlere erişebilir.
- Üye işlevleri AcceptDetails () ve Display () public olarak bildirildiğinden, bunlara Main () 'den r adındaki Rectangle sınıfının bir örneği kullanılarak erişilebilir.

Kapsülleme

- Protected Erişim Belirteci
 - Protected erişim belirticisi, bir alt sınıfın, temel sınıfının üye değişkenlerine ve üye işlevlerine erişmesine izin verir.
 - Bu şekilde kalıtımın uygulanmasına yardımcı olur.
 - Protected erişim belirtecini kalıtım konusunda görmüştük.

Kapsülleme

- Internal Erişim Belirteci
 - Internal erişim belirticisi, bir sınıfın üye özelliklerini ve üye fonksiyonlarını geçerli derlemedeki diğer fonksiyonlara ve nesnelere göstermesine izin verir.
 - Başka bir deyişle, internal erişim tanımlayıcısına sahip herhangi bir üyeye, üyenin tanımlandığı uygulama içinde tanımlanan herhangi bir sınıf veya fonksiyondan erişilebilir.
 - Örneğimizi inceleyelim. Örnek3
 - Örneğimizde, üye fonksiyon GetArea () herhangi bir erişim belirteci ile bildirilmemiştir.
 - Bu durumda bu fonksiyon hangi erişim belirtecine sahip olur?
 - Private

Arabirimler ve soyut sınıflar

- Arabirim, Arabirimden türemiş tüm sınıfların izlemesi gereken sözdizimsel bir sözleşme olarak tanımlanır.
 - Arabirim sözdizimsel sözleşmenin 'ne' bölümünü tanımlar ve türetilen sınıflar sözdizimsel sözleşmenin 'nasıl' bölümünü tanımlar.
- Arabirimler, Arabirimin üyeleri olan özellikleri, yöntemleri ve olayları tanımlar.
 - Arabirimler sadece üyelerin beyanını içerir.
 - Üyeleri tanımlamak türeten sınıfın sorumluluğundadır.
 - Genellikle türetilen sınıfların izleyeceği standart bir yapı sağlamaya yardımcı olur.
- Örneğimizi inceleyelim. Örnek4

Arabirimler ve soyut sınıflar

- Arabirimler konusunda dikkat edilmesi gerekenler:
 - 📖 Arabirimde tanımlanan yöntemlerin adları ve dönüş türleri asıl tanımlarıyla kesin olarak eşleşmelidir.
 - 📖 Yöntemlerin aldığı parametreler kesin olarak eşleşmelidir.
 - 📖 Bir sınıf hem bir arabirimden hem de başka bir temel sınıftan türeyebilir.
 - 📖 Ayrıca sınıf birden fazla arabirimden türeyebilir.

Arabirimler ve soyut sınıflar

```
interface ILandBound
{
    ...
}
```

```
class Mammal
{
    ...
}
```

```
class Horse : Mammal , ILandBound
{
    ...
}
```

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
}
```

Arabirimler ve soyut sınıflar


- Soyut sınıflar arabirimlere benzerler.
 - 📖 Ancak fonksiyonların içeriklerini soyut sınıfta belirtilebilir.
 - 📖 Soyut sınıfın normal sınıflardan farkı soyut sınıfın nesnesinin oluşturulamamasıdır.
 - Soyut sınıflar **abstract** kelimesi ile oluşturulur.
 - Örneğimizi inceleyelim. Örnek5

Hata yönetimi

- Herhangi bir dilde oluşturulan bir programın çalışma aşamasında hatalar meydana gelebilir.
- C# dilinde herhangi bir hata durumunda özel durum (exception) oluşur.
 - 📖 Özel durumlar oluştuğunda farklı kod çalıştırabilmeniz mümkündür.
 - 📖 Böylece hata durumunda alınması gereken önlemler belirtilir.
- Bu amaçla iki işlem yapılması gerekmektedir. İşlemlerden ilki:
 - 📖 Programınızın komutları gerçekleştiren kod kısmı bir try bloğunun içine yazılmalıdır.
 - try bloğunun içindeki komutlar bir özel durum oluşmaz ise sıradan gerçekleşir.
 - Hata oluşursa, try bloğu sonlanır ve özel durumu yakalayan ve gerekli işlemleri yapan başka bir kod parçasına atlanır.

Hata yönetimi

- İkinci işlem:

 Herhangi bir hata durumunda yapılması gerekenler, try bloğunun takip eden bir veya birkaç catch bloğunda belirtilir.

- catch blokları özel durumları yakalamak için oluşturulmuştur.
- Farklı hatalar için farklı catch blokları yazabilir ve böylece farklı hatalar için farklı önlemler alabilirsiniz.

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (FormatException fEx)
{
    // Özel durumu işle
    ...
}
```

Hata yönetimi

- Hata oluşan kod bloğunda bir catch bloğu yok ise
 - 📖 Kod bloğunu çağıran yönteme dönüş yapılır. Bu yöntemde bir catch bloğu var ise hata yakalanır. Aksi takdirde bu yöntemi çağıran yönteme dönüş yapılır.
 - 📖 Bu şekilde çağırı geçmiş izlenerek bütün kod parçalarında catch bloğu aranır.
 - 📖 Böyle bir blok bulunamazsa program işlenmemiş bir özel durum ile sonlanır.
 - Bu en istemediğimiz durumdur.
 - 📖 İşlenmemiş özel durumları önlemek için oluşabilecek hatalara uygun catch blokları yazılmalıdır.

Hata yönetimi

- Olası hata türleri arasında aşağıdakiler sıralanabilir.
 - `System.IO.IOException`
 - `System.IndexOutOfRangeException`
 - `System.ArrayTypeMismatchException`
 - `System.NullReferenceException`
 - `System.DivideByZeroException`
 - `System.InvalidCastException`
 - `System.OutOfMemoryException`
 - `System.StackOverflowException`

Hata yönetimi

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (FormatException fEx)
{
    //...
}
catch (OverflowException oEx)
{
    //...
}
```


Hata yönetimi

- Herhangi bir hatayı yakalayıp kullanıcıya en azından hatanın sebebini açıklayan bir metin gösterilmek istenirse genel bir catch bloğu yazılabilir.

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (Exception ex) // bu genel bir catch işleyicidir
{
    //...
}
```

Hata yönetimi

- İstenirse program içerisinde belli olaylar için anlamlı özel durumlar yaratılabilir.

```
public static string monthName(int month)
{
    switch (month)
    {
        case 1 :
            return "January";
        case 2 :
            return "February";
        ...
        case 12 :
            return "December";
        default :
            throw new ArgumentOutOfRangeException("Bad month");
    }
}
```


- Oluşacak özel durumun “Message” özelliğine belirtilen metin yazılacaktır.

Hata yönetimi

- Bir özel durum oluştuğunda programın akışının değişeceği unutulmamalıdır.
- Örneğin aşağıdaki kod parçasını inceleyelim.

```
TextReader reader = src.OpenText();
string line;
while ((line = reader.ReadLine()) != null)
{
    source.Text += line + "\n";
}
reader.Close();
```

- **reader.Close()** komutu çalışmadıkça ilgili kaynak tutulu kalır.

 Bu sebeple bu komutun hata durumunda dahi çalıştığından emin olmalısınız.

Hata yönetimi

- Bir özel durum oluşsa bile çalışması istenen komutlar **finally** bloğunun içine konmalıdır.
 - 📁 **finally** bloğu **try** bloğundan hemen sonra veya **try** bloğundan sonraki en son **catch** bloğundan sonra yazılır.
- Program **finally** bloğu ile alakalı **try** bloğuna girdi ise, **finally** bloğu kesinlikle çalışacaktır.
 - 📁 Özel durum oluşursa ve **finally** bloğundan önce **catch** bloğu var ise önce **catch** bloğu çalışır sonra ise **finally** bloğu çalışır.
 - 📁 Özel durum oluşursa ve **finally** bloğundan önce **catch** bloğu yoksa önce **finally** bloğu çalışır.
- Önceki kod parçamız **finally** bloğu ile şu şekilde değiştirilmelidir.

Hata yönetimi

```
TextReader reader = null;
try
{
    reader = src.OpenText();
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        source.Text += line + "\n";
    }
}
finally
{
    if (reader != null)
    {
        reader.Close();
    }
}
```



Hata yönetimi

- Hata yönetimi
 - Örneğimiz inceleyelim: Örnek6.
 - Örneğimizi inceleyelim: Örnek7.