HATA YAKALAMAK ve EXCEPTION SINIFLARI

Memik Yanık

Malumunuz olduğu üzere .NET Framework ve C# programlama dili Microsoft firması tarafından hazırlanıp kullanıma sunulalı az olmadı. Bu süreçte yani .NET Framework 1.0 dağıtıldıktan bu yana geçen sürede önce 1.1 sonra 2.0 ve en son olarak 3.0/3.5 sürümleri programcıların istifadesine sunuldu. Benzer gelişme C# programlama dili için de geçerlidir. İçinde bulunduğumuz günlerde .NET Framework 3.5 ile birlikte C# derleyicisinin 3.0 sürümü gelmektedir.

C# kodu içinde çalışma anında meydana gelmesi muhtemel hataları yakalamayla ilgili sınıflarda .NET Framework'ün ilk sürümden bugüne kayda değer bir gelişme olmadığı için C# konulu her kaynakta Exception sınıfları ile ilgili bilgiyi bulmak mümkündür. Ötesi 2002 yılından bu güne kadar hata yakalama sınıflarını anlatan çok sayıda makale programcılık konulu Web sitelerinde yayınlandı. Örneğin www.csharpnedir.com/makalegoster.asp?MId=295 adresindeki yazıda Resul Çavuşoğlu ayrıntılara girmese bile C# kodu dahilinde Exception sınıflarından nasıl yararlanıldığını gayet güzel anlatmaktadır. İkinci bir kaynak vermek gerekirse; ilk Türkçe C# kitabını yazma ünvanına sahip Ahmet Dizioğlu, Elif İçağasıoğlu ve Levent Çamlıbel'in ortak yazdıkları kitaplarında **Exception** sınıfları hakkında yeterli bilgiyi bulmak mümkündür. Kısaca özetlemek gerekirse; her kim Exception sınıfları ve hata yakalamak hakkında Türkçe kaynak ararsa ister kitap edinsin ister Web sitelerine baksın yeterli kaynağı zorlanmadan bulabilir. Exception sınıfları ile ilgili kaynakların Türkçe olması şartından vazgeçilirse MSDN'i saymasak bile binlerce makale, yüzlerce kitap bulmak mümkündür.

O zaman birileriniz kalkıp bana sormayacak mısınız: madem bu konuda kaynak bolluğu var o zaman neden bu konuda 30 sayfa uzunlukta bir makale yazıp yayınladınız? Tabii bu soru tek cümle ile cevaplanacak bir soru değildir. 2004 yılında yayınlanan ilk C# kitabımda hata yakalama ve Exception sınıflarının anlatıldığı sayfaların sayısı topu topu 7 idi. Yani yukarıda sözünü ettiğim ve Resul Çavuşoğlu tarafından kaleme alınan makalede, benim kitaptan Exception sınıfları hakkında daha fazla bilgiyi bulmak mümkündü. Çünkü sözünü ettiğim kitabı kaleme alırken kitabın sayfa sayısı fazla artmasın diye bazı konuları ayrıntılı anlatmak yerine kısaca değinmekle yetinmiştim. Sonra 2006 yılının sonunda yayınlanan ve C# 2.0'ı anlattığım kitabımda hata yakalama ve Exception sınıflarına ayırdığım sayfaların sayısını arttırıp 16'ya çıkarmıştım. Çünkü bu kitabımı 2 cilt olarak yayınlamaya karar vermiştim. Her ne kadar Exception sınıflarına C# 2.0 kitabımda 16 sayfa yer ayırmış olsam bile bu konuyu bütün yönleri ile anlattığımı söyleyemem. Derken C# 3.0 kitabımın üzerinde çalışırken Exception sınıflarına ve hata yakalama konusuna torpil geçip normalden daha fazla sayfa ayırdım. Çünkü istedim ki, C# üzerine kitap yazma cüretini gösteren Memik YANIK'ın **Exception** sınıfları hakkında az çok bilgi sahibi olduğumu insanlar bilsin. Daha doğrusu Exception sınıfları hakkında bu kadar ayrıntılı metni kaleme alan birisinin sıfıra bölme hatasını anlatırken değişken çalmak üzere başka kaynaklara başvurup başvurmayacağı konusunda işin meraklıları fikir sahibi olsunlar istedim. Bu metin yakında yayınlanacak C# 3.0 kitabımdaki ilgili bölümün küçük değişiklikler, eklemeler ve çıkarmalar yapılmış halidir.

Yazarlar Neden Yazarlar?

Bence bu sorunun cevabı basittir ve kısadır: Yazarlar bildiklerinin, düşündüklerinin, hissettiklerinin bilinmesini isterler. Yani bir arkadaş(Örneğin Selim Burak Şenyurt) bilgisayarın başına oturup LINQ konulu bir makale yazıyorsa, asıl demek istediği bence şudur: "Sayın okurlar bakın ben bu konuda günlerce araştırma yaptım, araştırmalarım sırasında öğrendiklerimi test ettim, sindirdim. Sizi bu konuda haberdar etmek istiyorum". Zaten bu kaygı ile kaleme alınan metin okunmaya değer olur. Tabii ki şartların zorlaması sonucu kaleme alınan tez, rapor vs. gibi metinler genellikle yukarıda işaret ettiğim güdü ile yazılmaz.

Tabii ki bu uzun makalede bozuk cümleler, yazım hataları, hatta teknik hatalar bile olabilir. Bu durumda okurun birisi beni hatadan haberdar ettiği zaman kendisine teşekkür edip hatamı düzeltme yoluna giderim. Eklemek istediğim 2. bir konu var: Microsoft ve diğer programlama dili üreten firmalar geliştirdikleri programlama dili ile ilgili bütün ayrıntıları açıklarlar. Yoksa insanların söz konusu programlama dilini öğrenip kullanmaları mümkün olmazdı. Bundan çıkardığım sonuç şudur: Öğretim üyeleri ve yazarlar olsa olsa programlama dilini kendi cümleleri ile anlatırlar; yeni bir şeyler söyleme ihtimali yoktur. Yazarların ve hocaların olsa olsa şöyle bir iddiası olabilir: Bu konuda bilgi sahibi olmayan birisine ben Exception sınıfları ve hata yakalamayı daha iyi anlatırım, Exception sınıflarını anlatırken vereceğim örnek kodlar daha anlaşılırdır vs. Yani elin gavuru bir programlama dili hazırlamış. Ötesi programlama dilinin bütün özelliklerini, metotlarını, sınıflarını vs. örnekleri ile açıklayıp anlatmış. Biz yazarlar ve hocalar ne yapıyoruz? Kendimizce bazı konuların önemli olduğuna kanaat getirip o konular hakkında fazladan birkaç cümle yazıyoruz. Bunu yaparken bilim yapmıyoruz, Amerika'yı yineden keşfetmiyoruz. Nasıl ki Türk Milli Takımının maçı hakkında ertesi gün bütün gazetelerin verdikleri kadrolar aynı ise, golleri atan futbolcular aynı ise bütün hocalar, bütün yazarlar C# programlama dilinden söz ederken int tipindeki değişkenlerin nasıl tanımlandığını benzer cümleler ile anlatırlar. Memik YANIK bu makalede konunun uzmanlarını geçelim, C# ile 3-5 satır kod yazan herkesin bilgi sahibi olduğu Exception sınıflarını kendince anlattı. Memik YANIK'ın iddiası olsa olsa şu olabilir: Exception sınıfları bu şekilde anlatılırsa daha kolay anlaşılır. Başka bir yazar veya hocanın yoğurt yiyişi farklı olacağı için bu sınıfları farklı cümlelerle anlatır. Madem anlatılan konu aynıdır o halde o cümleyi kim kaleme alırsa alsın bütün okurların aynı şeyi anlaması gerekir. Eğer bazı okurlar anlatılanlardan farklı sonuçlar çıkarıyorlarsa kabahat yazarındır ve yazar cümlelerinin çerçevesini iyi ayarlayamamıştır.

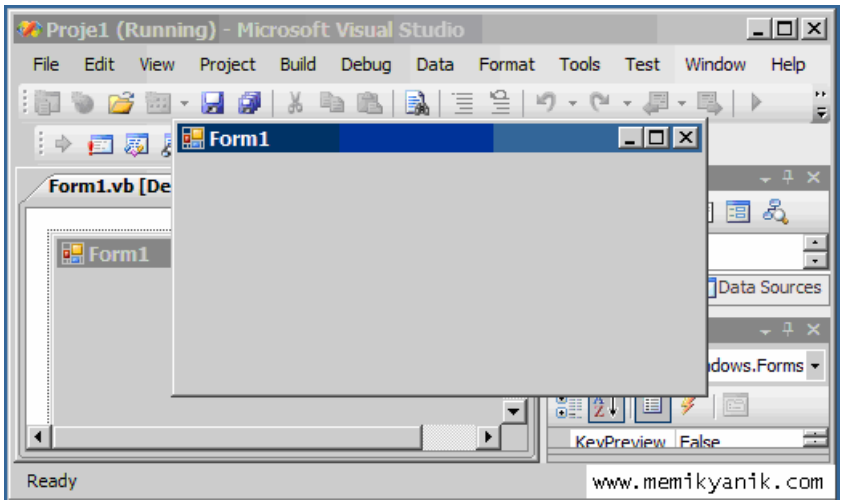
Projeleri Çalıştırmak

Bu metni hazırlarken Visual Studio'dan yararlandım. Sizler ücretsiz ve C# için hazırlanmış Express sürümden yararlanabilirsiniz. Tabii bilgisayarınıza .NET Framework'ü kurup(ki .NET Framework Windows Vista ile birlikte geliyor) Not Defteri ile kodunuzu yazıp konsolda yani DOS penceresinde derleyebilirsiniz. Bu şartlarda çalışma anında meydana gelecek hataları yakalamada herhangi bir sorun yaşamazsınız ama kodunuzdaki hataları ayıklamanız zahmetli olur. Çünkü Visual Studio ve Express Editon'la birlikte entegre çalışan bir hata ayıklayıcı(Debugger) verilmektedir. İşin özüne inebilmek için işe Visual Studio ile hazırlanan projelerin çalıştırılması ile başlayacağım.

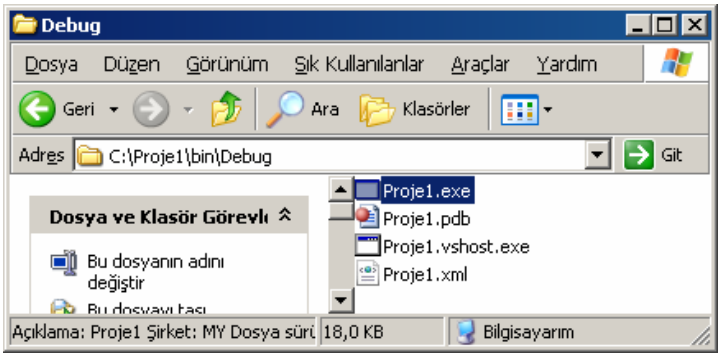
Bu amaçla Visual Studio ile gelen **Windows Forms Application** şablonundan yararlanıp "Proje1" adında C# projesi hazırladım. Bu projeyi "C:" sürücüsünde "\Proje1" klasörüne kaydettim. Projeyi kaydettiğim klasörü özellikle işaret ettim. Çünkü proje çalıştırıldığında geri planda Visual Studio tarafından yapılan işlemlerin üzerinde biraz durmak gerekecek.

Şimdi hazırladığım projede herhangi bir değişiklik yapmadan, başlangıç formuna kontrol yerleştirmeden ve proje için başka bir Class hazırlamadan çalıştıracağım. Projeleri çalıştırmak için genellikle **Debug** menüsündeki **Start Debugging** komutu kullanılmaktadır. Genellikle dedim; çünkü projeleri çalıştırmanın başka yöntemleri de var.

Her ne kadar konunun başlığı **Projeleri Çalıştırmak** olsa bile bu amaçla kullanacağım komutun adı Start Debugging. Yani yazdığım projeyi Debug etmek, test etmek ve mevcut hatalardan ayıklamak istiyorum. Başlangıçta Visual Studio penceresinin başlığı **Proje1-Microsoft Visual Studio** iken proje çalıştırıldığında Visual Studio penceresinin başlığı **Proje1[Running]-Microsoft Visual Studio** olmaktadır. Start menüsünden bu komutu vermek demek, projeyi entegre Debugger'ın kontrolünde test etmek demektir.



Visual Studio ile hazırlanan C# projeleri **Debug** menüsünden komut verilerek çalıştırıldıkları zaman Visual Studio tarafından otomatik olarak EXE dosya hazırlanmaktadır. Proje çalıştırıldığında hazırlanan EXE dosyanın yerini aşağıda görebilirsiniz. Bu EXE dosya ancak proje hazırlanırken tercih edilen .NET Framework sürümünün(2.0, 3.0 veya 3.5) kurulu olduğu bilgisayarda çalışabilir. Aşağıda ekran görüntüsü verilen dosya listesindeki "pbd" uzantılı dosyaya dikkatinizi çekmek istiyorum. Kodun debug edilmesi yani hatalardan ayıklanması ile ilgili bilgiler bu dosyaya yazılmaktadır. Tabii bu makalede hata ayıklama işlemleri üzerinde durmayacağımız için pbd uzantılı dosyanın işlevinden ve nasıl kullanıldığında söz edilmeyecektir.

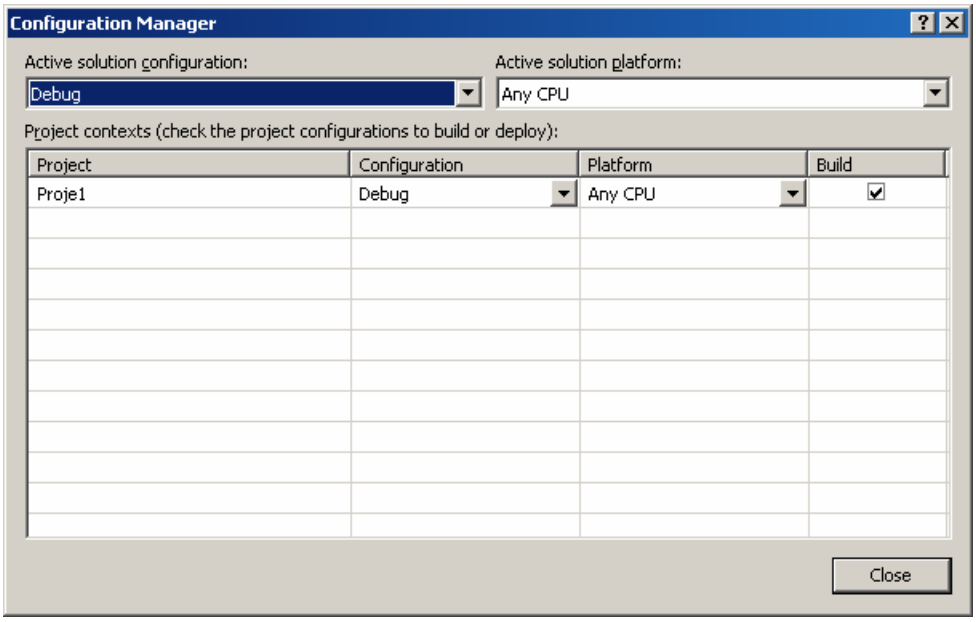


Yukarıda söylendiği gibi Visual Studio ile hazırladığınız C# projelerini çalıştırmak için **Debug** menüsünden **Start Debugging** komutunu verebilir veya direk F5 tuşuna basabilirsiniz. Aslında Debug menüsünden bu komutu vermekle Visual Studio'ya bir bakıma "hazırladığım projeyi entegre hata ayıklayıcısının nezaretinde test etmek istiyorum ve varsa hataları ayıklamak istiyorum" demiş oluyorsunuz. En başında belirtmek gerekir ki **Start Debugging** komutu verildiği zaman proje için otomatik olarak hazırlanan EXE dosya başkalarının kullanıp yararlanacağı kopya değildir. Elbette **Start Debugging** komutu sayesinde hazırlanıp projeye ait klasörün içinde yer alan "\Bin\Debug" klasörüne yerleştirilen EXE dosyayı alıp başka bilgisayarda çalıştırmak mümkündür. Ancak debug modunda iken hazırlanan EXE dosyanın dağıtılması önerilmiyor.

Şimdi gelelim şu **Debug** moduna. Programcılar genelde uygulamalarını geliştirmeyi ve test etmeyi Debug modunda yaparlar. Ne zaman ki uygulama tamamlanıp testlerden geçer o zaman **Release** sürümü hazırlayıp kullanıcılara öyle verirler. Yukarıdaki sayfalarda işaret edildiği gibi Visual Studio ile yeni C# projesi hazırlanıp kaydedildiği zaman projeye ait klasörün içinde "Bin" ve "Obj" adında 2 klasör hazırlanmaktadır. Derleme sırasında hazırlanan geçici dosyalar "Obj" klasörüne konulmaktadır. Obj klasörünün altında "Debug" ve "Release" adında 2 klasör hazırlanmaktadır. Aynı şekilde "Bin" klasöründe "Debug" ve Release" adında 2 klasör hazırlanmaktadır. Uygulamanın Debug sürümü Bin klasörünün içinde bulunan \Bin\Debug klasörüne konulurken Release sürümü **\Bin\Release** klasörüne yerleştirilmektedir.

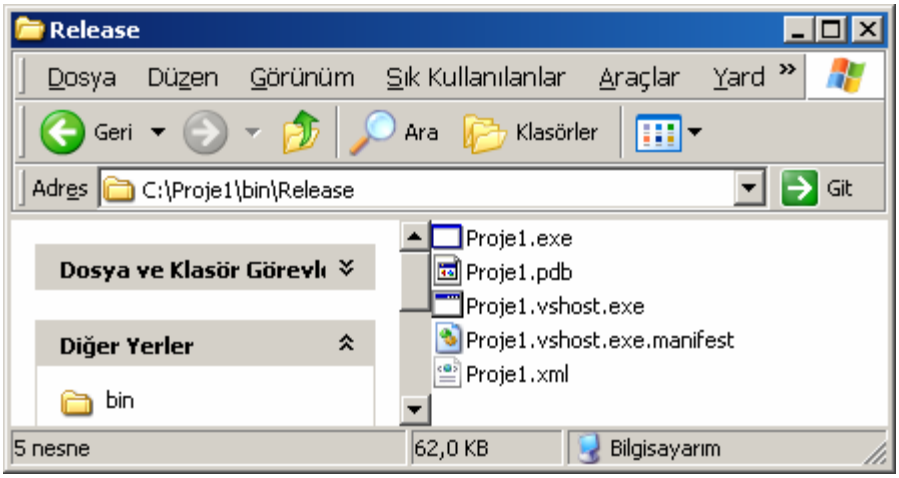
Debug modunda iken proje Debug menüsündeki **Start Debugging** komutu ile çalıştırıldığında hem "Obj" klasörünün altındaki Debug klasörüne hem de "Bin" klasörünün altındaki "Debug" klasöründe bazı dosyalar oluşmaktadır. Üzerinde çalıştığınız projeyi Ctrl+F5 tuşları ile çalıştırırsanız projeyi Debugger'dan bağımsız çalıştırmış olursunuz.

Debug modundan Release moduna geçmek istiyorsanız Visual Studio'nun Debug menüsünden komut verip **Cunfiguration Manager** diyalog kutusunu ekrana getirmelisiniz. Bu komut Debug menüsünde yer almıyorsa Tools menüsündeki **Options** komutu ile ekrana getirilen diyalog kutusunda ayarlama yapmalısınız.



Bu ekran görüntüsünü aldığım sırada üzerinde çalıştığım projenin dahil olduğu Solution bir tek projeye sahipti. Bu sırada Solution'da birden fazla proje olsaydı o projeler de listelenirdi. Bu diyalog kutusundaki **Active Solution configuration** ve **Active solution platform** liste kutularında yapılan seçimlerden Solution'daki bütün projeler etkilenmektedir.

"Active Solution configuration" liste kutusunu açıp **Release**'i seçerseniz Debug modundan Release moduna geçmiş olursunuz. Bu andan itibaren Debug menüsünden **Start Debugging** komutunu verip projeyi çalıştırırsanız EXE kodun Release sürümü hazırlanır ve "\Bin\Release" klasörüne yerleştirilir.



Hataları Yakalamak

Visual Studio ile C# uygulaması geliştirilirken konu hatalar olduğunda akla hemen iki kavram gelmektedir: **Hata Ayıklamak** ve **Hata Yakalamak**. Visual Studio ile birlikte gelen entegre hata ayıklayıcı(Debugger) sayesinde ilk bakışta görülmeyen hataları ayıklayabilmektesiniz. Bu bölümün konusu çalışma anında gelişen şartlara bağlı olarak meydana gelen hataları yakalamaktır.

Bir önceki bölümde bütün ayrıntıları ile olmasa bile hata ayıklama ve entegre debugger hakkında bilgi verildi. Hata ayıklama ve Debug işlemi hakkında anlatılanlar daha çok programcının işini kolaylaştırmaya yönelik işlemlerdi. Programcılıkta hata denildiği zaman asıl çalışma zamanında kullanıcının başrolde olduğu hatalar akla gelmektedir. Örneğin kullanıcıdan doğum tarihinin günü istenir ve gün bilgisi olarak kullanıcı 31'den büyük bir değeri girerse programcı herhangi bir tedbir almadıysa hata meydana gelir ve projenin çalışması kırılır. İşte bundan sonraki sayfalarda çalışma anında karşılaşılması muhtemel olan hataları nasıl yakalayabileceğinizden adım adım söz edilecektir.

C# projelerinde hata yakalama **try-catch-finally** blokları ile yapılmaktadır. Hata yakalama işlemlerinin nasıl yapıldığını örnek bir olay üzerinde anlatmak istiyorum. Kullanıcıdan bir işlem için tarih istediğinizi varsayalım. TextBox'a tarih olarak değerlendirilemeyecek bilgi girilirse programın çalışması kırılabilir. Aşağıda verilen metot işletildiğinde TextBox'ın içeriği dönüştürülüp **DateTime** tipindeki değişkene aktarılır.

```
private void textBox1_Leave(object sender, EventArgs e)
 {
   DateTime Tarih;
   Tarih = Convert.ToDateTime(textBox1.Text);
 }
```

Yerli yabancı yazarlar hata yakalamanın nasıl yapıldığını anlatmak istediklerinde genellikle kullanıcıdan DateTime tipinde bilgi isterler. Benim için tehlike asıl burada başlıyor: Şimdi birisi kalkıp "hata yakalama işlemini anlatırken kullanıcıdan tarih istemeyi senden önce akıl ettim, bu fikrin patenti bende derse" ne yaparım, kendimi nasıl savunurum? Tabii buna bir de değişken adı benzerliği eklenirse yandım demektir. Şimdi izninizle yukarıda verdiğim kodu biraz değiştireceğim ve "Tarih" yerine farklı ada sahip değişken tanımlayabildiğimi kanıtlamaya çalışacağım. Tabii burada benim için ikinci bir tehlike daha var: Ya benden önce yazarın birisi kitap veya makale yazıp DateTime tipindeki değişkene aktaracak bilgiyi kullanıcıdan TextBox aracılığı ile istemişse? Yani anlayacağınız programcılık üzerine yazmak zor iş.

```
private void textBox1_Leave(object sender, EventArgs e)
 {
   DateTime History;
   History = Convert.ToDateTime(textBox1.Text);
 }
```

Kullanıcı TextBox'a tarih veya zaman olarak değerlendirilebilinecek bilgi girdiği sürece bu kod hatasızca çalışır. Ne zamanki kullanıcı DateTime tipine dönüştürülemeyecek bilgiyi TextBox'a girip başka bir nesnenin üzerine giderse bu kod hata verir ve programın çalışması kırılır.

Burada yapılması gereken şudur: Hataya neden olma ihtimali yüksek olan bu satırı **try** bloğuna almak ve meydana gelebilecek hataları **catch** bloğunda yakalamaktır. Bu amaçla yukarıda verdiğim kodu aşağıdaki gibi düzenledim.

```
private void textBox1_Leave(object sender, EventArgs e)
 {
   DateTime History;
   try
    {
      History = Convert.ToDateTime(textBox1.Text);
    }
   catch
```

```
    {
      MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
      textBox1.Focus();
    }
  }
```

Bu kodda hataya neden olabilme ihtimali olan satırı **try** bloğuna aldım. Ardından **catch** bloğunda tahmin ettiğim hata meydana geldiğinde yapılacak işlemler için hazırlık yaptım. **catch** bloğunda kullandığım **Focus**() metodu sayesinde kontrol tekrar formdaki ilk TextBox'a geçer.

2. veya 3. denemede kullanıcı TextBox'a uygun bilgi girerse hata meydana gelmez ve dolayısıyla **catch** bloğuna yazılan satırlar işletilmez. catch-try bloklarının çalışma şeklindeki ayrıntıları ortaya çıkarmak için Convert sınıfının **ToDateTime**() metodu ile string bilginin DateTime tipine çevrildiği satırı ayrı bir metot olarak düzenledim("Tarih" adına sahip olmayan değişken tanımlayabilme becerisine sahip olduğumu bu şekilde kanıtladıktan sonra yine Türkçe değişken adına dönelim).

```
private void textBox1_Leave(object sender, EventArgs e)
 {
   DateTime Tarih;
   try
    {
     Tarih = tarih_yap(textBox1.Text);
    }
   catch
    {
      MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
      textBox1.Focus();
    }
 }
 public DateTime tarih_yap(string str)
 {
    return Convert.ToDateTime(str);
 }
```

Her ne kadar **Convert** sınıfının **ToDateTime**() metodu ile string bilgiyi DateTime tipinde bilgiye dönüştüren satır ayrı bir metodun içinde yer alsa bile "tarih_yap()" adını verdiğim bu metot **try** bloğunun içinden çağrılıp işletildiği için bu metotta meydana gelecek hatalar yine yakalanır. Başka bir deyişle try bloğunun içinden çağrılan **tarih_yap()** metodunda herhangi bir hata meydana gelmesi halinde programın işletimi yine **catch** bloğuna geçer.

Bundan şu sonuç çıkarılabilir: C# uygulamalarının başlangıç noktası **Main**() metodu içinde try-catch bloğu hazırlanırsa, başka bir deyişle uygulamayı başlatan **Run**() metodu Try bloğuna yazılırsa uygulamanın neresinde hata meydana gelirse gelsin bu hata yakalanır. Eklemek gerekirse, C# uygulamaları Main() metodundan itibaren çalışmaya başlar ve Main() metodunun sonuna gelindiğinde ise uygulamanın çalışması sona ermiş olur.

```
static void Main()
 {
   try
    {
      Application.Run(new Form1());
    }
   catch
    {
     MessageBox.Show("Herhangi bir yerde hata meydana geldi");
    }
 }
```

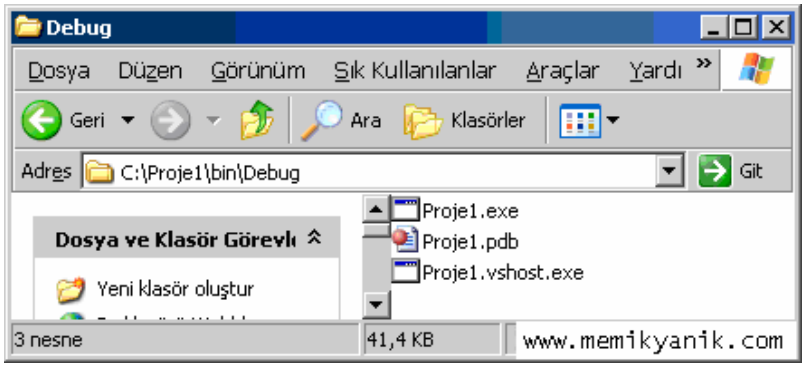
try-catch-finally blokları ile ilgili olarak hemen söylenmesi gereken birkaç kural daha var. Örneğin try bloğundan hemen sonra ya catch ya da finally bloğu gelmelidir. Bu nedenle aşağıdaki gibi düzenlenen try-catch bloğu derleme sırasında hataya neden olur. Çünkü bu kodda try ile catch blokları arasında bir satır bulunmaktadır.

```
private void textBox1_Leave(object sender, EventArgs e)
 {
   DateTime Tarih;
   try
    {
     Tarih = Convert.ToDateTime(textBox1.Text);
    }
   this.Text = "Hata yakalama blokları";
   catch
    {
     MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
     textBox1.Focus();
    }
 }
```

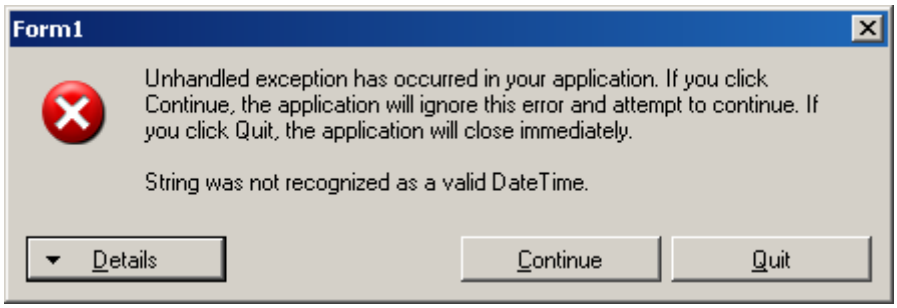
try-catch-finally blokları ancak bir metodun içinde yer alabilirler. Çünkü C# derleyicisi ancak metotlar şeklinde düzenlenmiş kod parçalarını işletebilmektedir. Bu nedenle try bloğu içinde metot kodlanmak istenirse derleme sırasında hata meydana gelir.

Hata yakalama konusuna bu şekilde giriş yaptıktan sonra şimdi biraz işin geri planı üzerinde duralım. Bildiğiniz gibi programcılar genellikle uygulamalarını Debug modunda hazırlayıp entegre hata ayıklayıcısının desteği ile hatalardan ayıklarlar. Ayrıca çalışma anında karşılaşılan kullanıcı kaynaklı hatalarından korunmak veya içinde bulunulan şartlardan dolayı meydana gelmesi muhtemel hataları yakalayabilmek için **try-catch** blokları hazırlarlar. Daha sonra uygulamalarının Release sürümlerini hazırlayıp kullanıcıların istifadesine sunarlar.

Bu döngüyü gözetip yukarıda verdiğim koddaki try-catch deyimlerini sildim. Başka bir deyişle olası hataları yakalamak üzere yaptığım hazırlıkları iptal ettim. Devamında **Build** menüsünden **Build** komutunu verip üzerinde çalıştığım projeyi derledim. Daha önceki konulardan bildiğiniz gibi **Build** menüsünden komut verilip üzerinde çalışılan proje derlendiğinde Debug modunda iken hazırlanan EXE dosya "\bin\debug", **Release** modunda iken derleme yapıldığında ise EXE dosya "\bin\release" klasörüne konulmaktadır. Derleme işlemini yaparken Debug modunda olduğum için derleme sonucu hazırlanan EXE dosya aşağıda işaret edildiği üzere "\bin\debug" klasörüne yerleştirildi.



Bu klasördeki EXE dosya çift tıklanıp proje çalıştırılıp TextBox'a girilen bilginin tarihsel bilgiye dönüştürülmesi sırasında hata meydana gelmesi sağlanırsa ekrana aşağıda verdiğim diyalog kutusu gelir. Bu diyalog kutusundaki **Continue** düğmesini tıklayıp hatanın atlanılıp göz ardı edilmesini sağlayabilir veya **Quit** düğmesini tıklayıp uygulamanın çalışmasını sona erdirebilirsiniz. Meydana gelen hata hakkında ayrıntılı bilgi edinilmek istendiğinde ise **Details** düğmesi tıklanır.

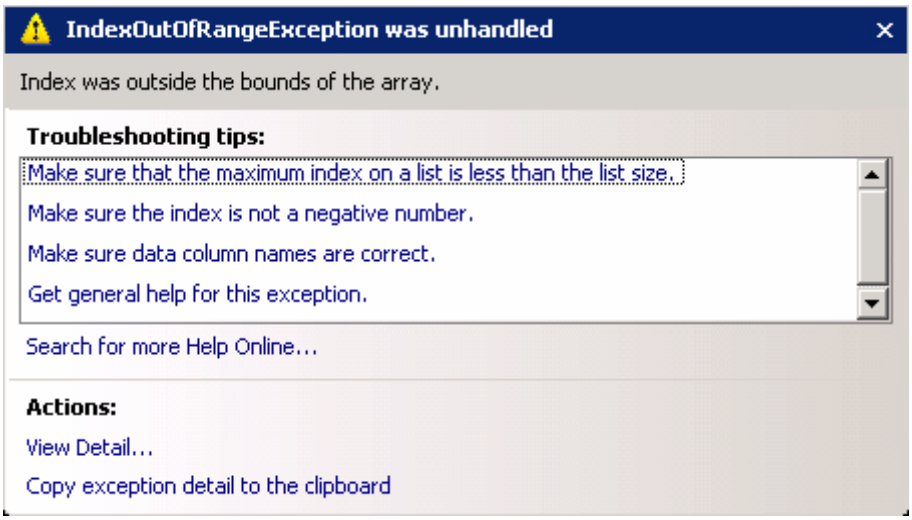


Tahmin edeceğiniz gibi bu proje Debug menüsündeki **Start Debugging** komutu ile çalıştırılsaydı bu hata ile karşılaşıldığında projenin çalışması kırılırdı ve hataya neden olan satır işaret edilirdi. Çünkü Debugger'ın başlangıç ayarlarında böyle bir hata ile karşılaşıldığı zaman uygulamanın çalışması sona erdirilmektedir. Yukarıda ise uygulamayı entegre debugger'dan bağımsız çalıştırdığım için uygulamanın çalışması kırılmadı. Eğer yukarıda yaptığım gibi çalışma anında meydana gelmesi muhtemel bu hatayı yakalamak için try-catch bloğu hazırlasaydım projeyi ister debug modunda, ister Visual Studio kurulu olmadığı başka bir bilgisayarda çalıştırmış olsaydım değişen bir şey olmazdı.

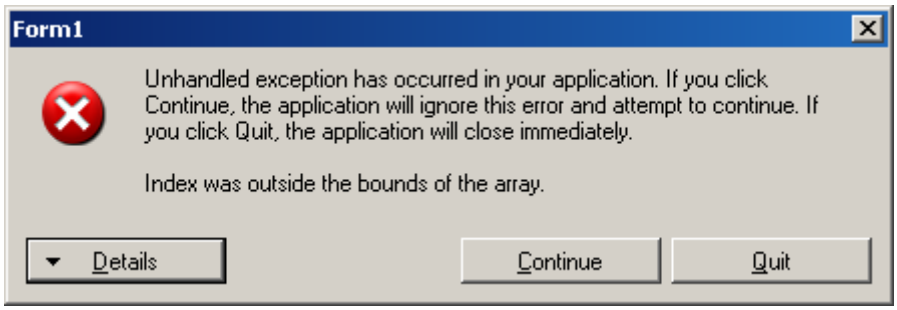
Şimdi hata yakalama konusunda 2. bir örnek vereceğim. Hazırlayacağım örnekte 5 elemanlı bir dizi değişken tanımlayıp **for** döngüsü ile dizi değişkenin elemanlarına bilgi aktaracağım. Dizi değişkene aktaracağım bilgileri forma yerleştirmiş olduğum ListBox'tan alacağım. Aşağıda verilen 4 satırlık kodda tanımladığım 3 değişken("Sayi", "Takimlar" ve "i") C# programcıları ve yazarlar tarafından çok sık kullanılan değişken adlarıdır. "i", "sayi" ve "takimlar" adlı değişkenleri başka kaynaklardan yararlanmadan tanımlayabildiğime bu konuda araştırma yapanları belki inandırabilirim. "Sayi" yerine "number", "i" yerine "I" ve "takimlar" yerine "ekipler" adında değişkenler tanımlayabilirdim.

```
string[] takimlar = new string[5];
int adet = listBox1.Items.Count;
for (int i=0; i<adet; i++)
   takimlar[i]=listBox1.Items[i].ToString();
```

ListBox, 5 veya daha az elemana sahip iken bu kod sorunsuz olarak çalışır. Ancak ListBox'ın eleman sayısı 5'ten fazla iken bu kod işletildiğinde programın çalışması kırılır ve aşağıdaki gibi bir hata mesajı alınır.



Tahmin edeceğiniz gibi uygulamanın kırılmasını sağlayan entegre hata ayıklayıcıdır. Eğer bu projeyi derleyip EXE dosyayı Visual Studio'dan yani Entegre hata ayıklayıcıdan bağımsız çalıştırsaydım uygulamanın çalışması kırılmayıp aşağıdaki gibi bir mesaj alınırdı.

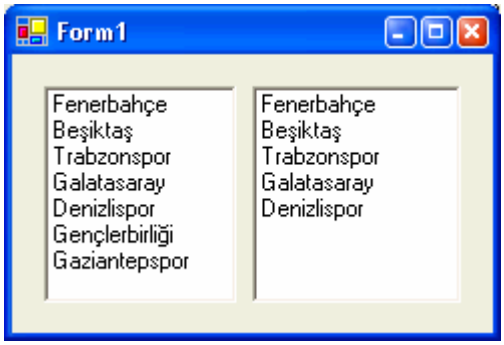


Bu hatanın meydana gelme nedeni dizi değişkenin olmayan 6. elemanına bilgi aktarılmak istenmesidir. Yapılan tanımlamadan dolayı dizi değişken 5 elemana sahip olduğu için hata meydana geldi. Bu gibi hataların önüne geçip programın çalışmasının kırılmasını önlemek için aşağıdaki gibi **try-catch** blokları hazırlanabilir.

```
private void Form1_Click(object sender, EventArgs e)
 {
   string[] takimlar = new string[5];
   int adet = listBox1.Items.Count;
   try
    {
      for (int i = 0; i < adet; i++)
      takimlar[i] = listBox1.Items[i].ToString();
    }
   catch
    {
      MessageBox.Show("Dizi Değişkenin sınırı aşıldı\n"+
           "Yalnızca ilk 5 eleman diziye aktarıldı");
    }
  listBox2.Items.AddRange(takimlar);
 }
```

Bu kodda hata meydana geleceğini öngördüğüm satırları **try** bloğuna aldım. Bu satırlarda hata meydana gelmezse kod sorunsuz çalışır ve programın işletimi **catch** bloğundan sonraki ilk satıra geçer. Hata meydana gelirse hatanın meydana geldiği noktada işleme ara verilir ve **catch** bloğundaki satırlara geçilir.

Bu örnekte **catch** bloğunda kullanıcıya mesaj vermekle yetindim. Bu şekilde düzenlenen kodun programın çalışmasını kırmadığını göstermek için forma 2. bir ListBox yerleştirdim ve **AddRange**() metodu ile dizi değişkenin elemanlarını 2. ListBox'a aktardım.



Şimdi yukarıda hazırladığım catch bloğunda değişiklik yapacağım. Hazırlamış olduğum örnekte **try** bloğundaki satırlar nasıl bir hataya neden olurlarsa olsunlar, başka bir anlatımla nasıl bir hata meydana gelirse gelsin catch bloğu işletilir. Bu catch bloğundaki satırların yalnızca dizi değişkenlerle ilgili olarak indis sorunu yaşandığında işletilmesini isteseydim aşağıdaki gibi düzenleme yapardım.

try bloğundaki satırlarda çalışma anında dizi değişkenlerle ilgili bir hata meydana geldiğinde **IndexOfRangeException** sınıfının örneği hazırlanıp catch bloğuna gönderilir(aşağıda verdiğim

kodda “Hata” yerine “Hataspor” adında değişken tanımlayıp kullandım. Birkaç satırdan meydana gelen bu kodda bilseydim ki Orhan babaya ayıp olmaz “hata” yerine “hatalarimla_sev_beni” adında bir değişken tanımlayıp kullanırdım.

```
private void Form1_Click(object sender, EventArgs e)
 {
  string[] takimlar = new string[5];
  int adet = listBox1.Items.Count;
  try
   {
    for (int i = 0; i < adet; i++)
      takimlar[i] = listBox1.Items[i].ToString();
   }
  catch(IndexOutOfRangeException Hata)
   {
    MessageBox.Show("Meydana gelen hatanın mesajı:\n"+ Hata.Message);
   }
  listBox2.Items.AddRange(takimlar);
 }
```

Bu örnekte sistem tarafından fırlatılan istisnanın veya hazırlanıp catch bloğuna gönderilen **IndexOutOfRangeException** tipindeki nesnenin **Message** özelliğinin içeriğini ekrana yazdım. Exception sınıfından türetilen IndexOutOfRangeException sınıfının Message özelliğinden başka **Source**, **HelpLink**, **StackTrace**, **TargetSize** ve **InnerException** gibi özellikleri de bulunmaktadır.

Bu kodu tekrar yorumlamak gerekirse; try ve catch blokları birlikte çalışmakta ve try bloğundaki herhangi bir satırda hata meydana geldiğinde meydana gelen hataya göre hemen bir Exception nesnesi hazırlanıp programın işletimi catch bloğuna geçmektedir.

Yukarıda verdiğim kodda catch bloğuna gönderilen exception nesnesinin **IndexOutOfRangeException** tipinde olup olmadığını araştırdım. Bu sırada catch bloğuna gönderilen Exception nesnesi IndexOutOfRangeException sınıfının örneği ise catch bloğuna yazılan satırlar işletilir.

try bloğunda meydana gelen hatadan dolayı catch bloğuna gönderilen bütün nesneler **System**'de bulunan **Exception** sınıfından türetilen sınıfların örnekleridir. Bu kodda dizi değişkenin olmayan bir elemanı üzerinde işlem yapılmak istendiği için catch bloğuna **IndexOutOfRangeException** tipinde bir nesne gönderilir. Konu üzerinde düşünmenizi sağlamak için bu makalenin ilk sayfalarında verdiğim örneği aşağıdaki gibi düzenledim.

```
private void textBox1_Leave(object sender, EventArgs e)
 {
   DateTime History;
   try
    {
     History = Convert.ToDateTime(textBox1.Text);
    }
   catch(IndexOutOfRangeException Hata)
    {
      MessageBox.Show("Girdiğiniz tarih veya saat yanlış");
      textBox1.Focus();
    }
 }
```

Bu şartlarda TextBox'a DateTime tipine dönüştürülemeyecek bilgi girilip TextBox'tan ayrılıp **Leave** olayı meydana getirildiğinde try bloğundaki satırlar hataya neden olur ve programın işletimi catch bloğunda geçer. Ne ki catch bloğunda dönüştürme işlemi sonucu meydana gelen hata yakalanamaz. Çünkü catch deyimine ait parantezin içinde **IndexOutOfRangeException** tipinde değişken tanımlandığı için catch bloğundaki satırlar işletilmeyip meydana gelen hatadan dolayı programın çalışması kırılır.

Şimdi ise hata yakalamanın nasıl yapıldığı anlatılırken ilk akla gelen sıfıra bölme hatasını yakalayan bir örnek vereceğim. Bu amaçla forma 3 TextBox ve 1 düğme yerleştirdim. İlk 2 TextBox'a bilgi girilip "Hesapla" adını verdiğim düğme tıklandığında ilk 2 TextBox"ın içeriklerini Byte tipindeki değişkene aktarıp 1. sayıyı ikinciye bölüp çıkan sonucu 3. TextBox'a yazacağım. Bu işlemi yapacak kodu aşağıda verdim.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   byte Sayi1 = Convert.ToByte(textBox1.Text);
   byte Sayi2 = Convert.ToByte(textBox2.Text);
   float Sayi3 = Sayi1 / Sayi2;
   textBox3.Text = Sayi3.ToString();
 }
```

İzninizle tam burada biraz durup bir şeyler eklemek istiyorum. Bu 4 satırlık kod ile ilgili olarak önce amacımın ne olduğunu ortaya koyacağım. Amacım net: **Sıfıra bölme hatasını anlatacağım.** Eğer ilkokul öğrencilerine sıfıra bölme hatasını anlatıyor olsaydım elime bir hesap makinesi alır ve herhangi bir sayıyı sıfıra bölerdim. Hesap makinesi sıfıra bölme işleminin sonucu olarak LCD ekranına yazdığı Error'u öğrencilere gösterip onları kolayca ikna ederdim. Madem konumuz programcılık ve kullanıcıların yapması muhtemel sıfıra bölme hatasını yakalayacağız; birbirine böleceğimiz 2 sayıyı kullanıcıdan isteriz. Ayrıca kendisine mümkünse ikinci sayının yani bölen'in sıfır olmasını rica ederiz. Amacımız sıfıra bölme hatasını yakalamak olduğuna göre hata yapılmadan hatayı yakalama imkanımız olamayacağına göre kullanıcıdan 2 sayı isteyeceğiz. Asıl sorun ise kullanıcıdan sayıları isterken yaratıcı olabilmek veya kimsenin aklına gelmeyen bir tekniği kullanabilmektir. Kullanıcıdan sayı isterken "bu sayıları şu TextBox'lara gir" dersek başkalarının fikrini çalmış olabiliriz.

Ne yazık ki yıllardır bilgisayar ve programcılık üzerine yazdığı binlerce sayfada milyonlarca cümleyi kuran Memik YANIK bundan 4 yıl kadar C# üzerine yazarken boş bulunup birbirine böleceği 2 sayıyı kullanıcıdan isterken TextBox'lara girmesini istemişti. Tabii bununla kalmayıp kullanıcının TextBox'lara girdiği bilgileri dönüştürüp Sayi1, Sayi2 adında 2 değişkene aktarmıştı. Memik Yanık bu 2 değişkeni yıllar önce yazdığı kitaplarında kullanmış olsa bile bu değişkenleri namı hesabına kaydetmeyi, register etmeyi akıl etmemiş.

Başka makale ve kitap yazarlarının telif haklarına tecavüz etmemek için bu 4 satırlık kodu aşağıdaki gibi düzenledim. Bu kodda TextBox'lara girilecek sayılar **Byte** tipindeki değişkenlere aktarılmaktadır. Formdaki ilk 2 TextBox'a 255'ten küçük bir sayı girilirse bu kod hatasızca çalışır. Ancak kullanıcı 2. TextBox'a 0 yazarsa sıfıra bölme hatası meydana gelir ve programın çalışması kırılır.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   byte Number1 = Convert.ToByte(textBox1.Text);
   byte Number2 = Convert.ToByte(textBox2.Text);
   float Number3 = Number1 / Number2;
   textBox3.Text = Number3.ToString();
 }
```

Aynı şekilde TextBox'lara 255'ten büyük bir değer yazılırsa taşma(değişkenler byte tipinde olduğu için) meydana gelir ve programın çalışması kırılır. Hata meydana geldiği zaman programın çalışmasının kırılmasını engellemek için try-catch bloğu hazırladım. Değişkenlere Sayi1, Sayi2 ve Sayi3 gibi adlar vermenin mecburi olmadığını böylece kanıtladıktan sonra tekrar Türkçe değişken adlarına dönelim.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   byte Sayi1, Sayi2;
   float Sayi3;
```

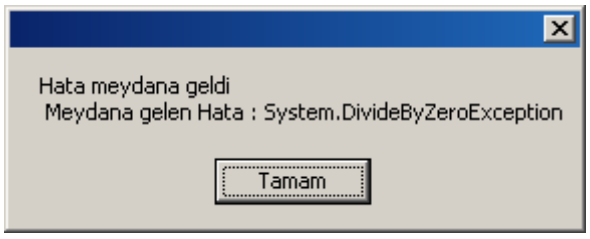
```
        try
         {
           Sayi1 = Convert.ToByte(textBox1.Text);
           Sayi2 = Convert.ToByte(textBox2.Text);
           Sayi3 = Sayi1 / Sayi2;
           textBox3.Text = Sayi3.ToString();
         }
        catch
         {
           MessageBox.Show("Hata meydana geldi");
           textBox1.Focus();
         }
    }
```

Ek açıklama yapmak gerekirse bu koddaki **try** bloğundaki satırlar işletilirken ister taşma hatası ister sıfıra bölme hatası meydana gelsin her şartta kullanıcıya catch bloğundaki mesaj verilip tekrar ilk TextBox'ın üzerine gidilir. Konu üzerinde düşünmenizi sağlamak için bu kodun catch bloğunu aşağıdaki gibi düzenledim. C# programlama dilinin ne "Error" ne de "error" adında bir anahtar kelimesi olmadığı için bu kodda yakalanacak Exception nesnesinin aktarılacağı değişkenin adını "Hata" olarak seçmek yerine "error" adını kullanmanız önerilir. Elin gavurları gelip bu değişken bana aittir demeyeceğine göre sorun yaşamazsınız.

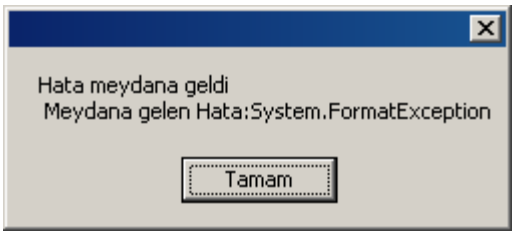
```
catch(Exception Hata)
 {
   object Tip = Hata.GetType();
   MessageBox.Show("Hata meydana geldi\n Meydana gelen Hata:"+
              Tip.ToString());
   textBox1.Focus();
 }
```

Burada yapılan şudur: catch anahtar kelimesine ait parantezlerin içinde **Exception** tipinde bir değişken tanımladım. catch deyimi, try bloğundan gönderilen nesnenin tipinin parantezlerin içinde tanımlanan değişkenle aynı tipte olup olmadığına bakar. Gönderilen Exception nesnesi **System.Exception** tipinde ise catch bloğundaki satırlar işletilir. Hata nedeniyle try bloğundan catch bloğunda gönderilen exception nesnesinin tipi ister **IndexOutOfRangeException** olsun ister **OverflowException** her şartta catch bloğundaki satırlar işletilir. Çünkü **IndexOutOfRangeException** ve **OverflowException** sınıfları **System.Exception** sınıfından türetilmiş sınıflardır.

Bu koda önce **object** tipinde bir değişken tanımlayıp catch bloğuna gönderilen nesnenin tipini **GetType**() metodu ile öğrendim. Catch bloğundaki satırların işletilmesinin nedeni sıfıra bölme iken aşağıdaki gibi bir mesaj alınır.



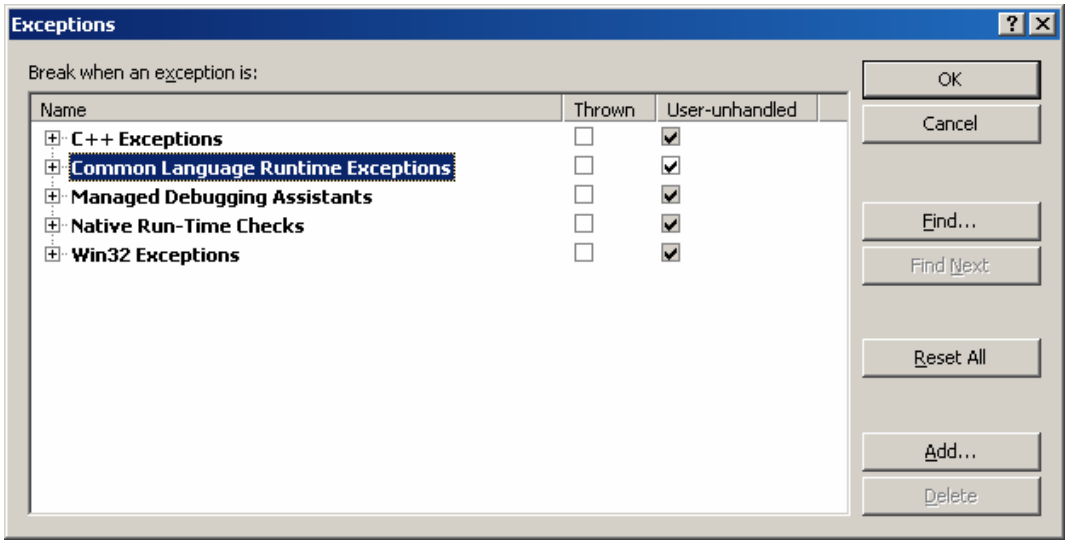
Gördüğünüz gibi bu sırada meydana gelen hata sıfıra bölme işlemi kaynaklı olduğu için catch bloğuna DivideByZeroException sınıfının örneği gönderilmiş. TextBox'lardan birisine Convert sınıfının **ToByte**() metodu ile byte tipine dönüştürülemeyecek bilgi girip ondan sonra yukarıda verdiğim kodu işletseydim aşağıdaki gibi hata mesajı alırdım.



Entegre Hata Ayıklayıcısı ve Exception Sınıfları

Yukarıda sözü edilen sınıflardan başka .NET Framework ile birlikte OverflowException, InvalidCastException, FormatException, ArgumentException, ArithmeticException, OutOfMemoryException, ArrayTypeMismatchException NullReferansException ve MemberAccessException gibi sınıflar da gelmektedir.

En çok kullanılan hata yakalama sınıflarından böylece biraz söz ettikten sonra mevcut Exception sınıflarıyla entegre hata ayıklayıcının ilişkisinden biraz söz edelim. Bu amaçla **Debug** menüsünden komut verip ekrana **Exceptions** diyalog kutusunu getirdim.

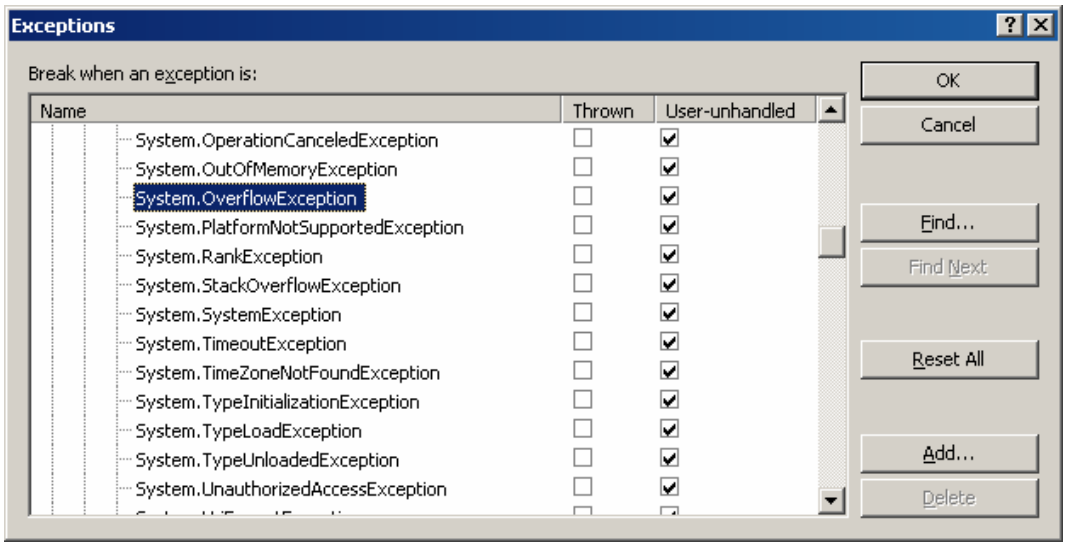


Bu diyalog kutusunda C# uygulamaları dahilinde yararlanabileceğiniz Exception sınıfları ile entegre hata ayıklayıcısı arasındaki ilişki ayarlanmaktadır. Bu konuda bilgi vermek için forma bir TextBox ve düğme yerleştirip "islem" adını verdiğim düğme için aşağıda verdiğim kodu hazırladım.

```
private void islem_Click(object sender, EventArgs e)
  {
    byte sayi;
    try
     {
       sayi = Convert.ToByte(textBox1.Text);
     }
   catch
    {
      MessageBox.Show("Hata meydana geldi");
    }
 }
```

Tahmin edeceğiniz gibi Debug modunda iken uygulama **Start Debugging** komutu ile çalıştırılıp formdaki ilk TextBox'a 255'ten büyük bir değer girilmesi halinde hata meydana gelir ve OverflowException sınıfının örneği hazırlanıp catch bloğuna gönderilir.

Şimdi öyle bir ayarlama yapacağım ki uygulama Debug modunda iken Start Debuging komutu ile çalıştırıldığında catch bloğundaki satırların işletilmesi yerine uygulamanın çalışmasının kırılmasını sağlayacağım. Bu amaçla **Exceptions** diyalog kutusunda önce **Common Language Runtime Exceptions** adlı exception sınıfı grubuna ait namespace'lerin listelenmesini sağladım. Ardından **System**'de yer alan **OverflowException** sınıfını buldum.



Verilen ekran görüntüsünde tespit edebileceğiniz gibi **Thrown** özelliği pasif iken **User-unchandled** özelliği aktif durumdadır. Başka bir deyişle kodun herhangi bir yerinde meydana gelebilecek ve OverflowException sınıfının fırlatılmasına neden olan hata yakalanmazsa programın çalışması kırılır. Yukarıda verdiğim kodda TextBox'a 255'ten büyük bir değer girilirse, dolayısıyla byte tipindeki değişkene 255'ten büyük bir sayı aktarılmak istenirse meydana gelebilecek hata yakalandığı için uygulamanın çalışması kırılmaz. Başka bir deyişle mevcut ayarlarda byte değişkene 255'ten büyük bir değer aktarılmak istenirse ve OverflowException hatasını yakalayan bir catch bloğu yoksa uygulamanın çalışması kırılır. Benzer şekilde .NET Framework ile gelen sınıfların hemen hepsi için **User-unhanded** sütunundaki onay kutusu seçili durumda olduğundan yakalanmayan bütün hatalar uygulama Debug modunda iken çalışmanın kırılmasına neden olurlar.

Bu diyalog kutusunda **System**'de yer alan OverflowException sınıfının User-unchandled özelliğini aktif durumda iken ayrıca **Thrown** özelliğini aktif duruma getirdikten sonra OK düğmesi ile bu diyalog kutusunu kapattım. Ardından yukarıda verdiğim kodu aşağıdaki gibi düzenledim. Değişken adı benzerliği olmasın diye "sayi" yerine "number" adını kullandım.

```
private void islem_Click(object sender, EventArgs e)
  {
    byte number;
    try
     {
       number = Convert.ToByte(textBox1.Text);
       if (number == 100)
          throw new OverflowException();
     }
   catch
    {
      MessageBox.Show("Hata meydana geldi veya 100 sayısını girdiniz");
    }
  }
```

Önce bu kodu biraz yorumlayalım: TextBox'a 255'ten büyük bir değer girilip bu kod işletilirse **ToByte**() metodunun kullanıldığı satır hataya neden olur ve catch bloğuna OverflowException sınıfının örneği gönderilir. TextBox'a 100 sayısı girilip bu kod işletildiğinde ise **try** bloğundaki karşılaştırma doğru değerini vereceği için catch bloğuna yine OverflowException hata sınıfının

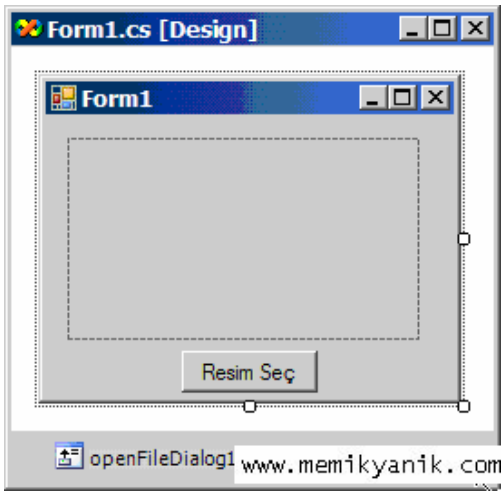
örneği gönderilir. Tekrar etmek gerekirse ister TextBox'a 100 ister 255'ten büyük bir sayı yazılsın OverflowException sınıfının örneği catch bloğuna fırlatılır.

Bu şartlarda yani **Exceptions** diyalog kutusunda OverflowException sınıfının **Thrown** özelliği seçili durumda iken Debug modundaki uygulama **Start Debugging** komutu ile çalıştırıldığında TextBox'a ister 100 girilsin ister 255'ten büyük sayı girilsin uygulamanın çalışması kırılır. Çünkü Exception diyalog kutusunda yaptığım ayarlamanın anlamı şudur: Her ne nedenden olursa olsun ister direk Throw deyimi ile, ister yapılmak istenen işlem OverflowException hatasını üretirse uygulamanın çalışması kırılır.

Tabi uygulamayı **Build** menüsündeki komutlarla derleyip sonra klasik yöntemlerle(yani EXE dosyanın olduğu klasöre gidip EXE dosyayı çift tıklayarak) Visual Studio'dan bağımsız olarak çalıştırırsanız TextBox'a ister 255'ten büyük sayı girilsin ister 100 girilsin catch bloğundaki satırlar işletilir ve uygulamanın çalışması kırılmaz ve catch bloğu işletilerek hata yakalanır.

Finally Bloğu

Bazen herhangi bir nedenden dolayı yarı yolda vazgeçilen bir işlemden dolayı geride bazı kalıntılar kalır. Bu gibi durumlar kaynak israfına neden olur. Bu konuda bilgi vermek için aşağıda verdiğim ancak pratik değeri olmayan uygulamayı hazırladım.



Kullanıcı bir resmin dosyasının içeriğini görmek istediğinde "Resim Seç" düğmesini tıklayacak. Bu düğme tıklandığında Aç diyalog kutusu ekrana getirilecek ve istediği resim dosyasını seçecek. Bu işlemler için aşağıda verdiğim kodu hazırladım. Kullanıcı Aç diyalog kutusunda PictureBox'ta görüntülenebilecek uygun bir resim dosyasını seçtiği sürece bu kod hatasızca çalışır.

1994 yılında yayınlanan Clipper kitabımın birçok sayfasında "DosyaAdi" adında değişken tanımlamışım. 2004 yılında yayınlanan C# kitabımın bir çok sayfasın "dosya" adında değişkenler tanımlamışım. Doğrusunu söylemek gerekirse kısacak kodları hazırlarken değişkenin adı şu olsun bu olmasın diye kaygım yıllardır olmadı. Değişkene ad seçerken genellikle değişkene aktaracağım bilgiyi göre seçim yaparım. Örneğin tanımladığım değişkene Fenerbahçe bilgisini aktarmayı düşünüyorsam değişkene genellikle "takim" adını veririm. Yıllar önce yani 1990'lı yılların başında programcılık kitabı yazarlarının ve hocaların arasında "dosya" yerine "kutuk" adında değişken tanımlama alışkanlığı yaygındı. Sonra "kutuk" unutuldu ve "dosya" diyenler çoğaldı. Gerçekte programcılık sitelerindeki makalelerde ve kitaplarda "File" karşılığı "dosya" adında değişken tanımlama alışkanlığı çok yaygındır.

```
private void Resim_sec_Click(object sender, EventArgs e)
 {
   openFileDialog1.ShowDialog();
   string kutuk = openFileDialog1.FileName;
```

```
    Bitmap Resim = new System.Drawing.Bitmap(kutuk);
    pictureBox1.Image = (Bitmap)Resim;
    Resim = null;
  }
```

Ancak kullanıcı **OpenFileDialog** nesnesi sayesinde ekrana getirilen diyalog kutusunu dosya seçmeden kapatır veya uygun olmayan bir dosyayı seçerse hata meydana gelir. Kullanıcının yanlış yaptığı veya yapmadığı dosya seçiminden dolayı meydana gelecek hata konusunda kendisine bilgi vermek için kodu aşağıdaki gibi düzenledim.

```
private void Resim_sec_Click(object sender, EventArgs e)
 {
   System.Drawing.Bitmap Resim;
   openFileDialog1.ShowDialog();
   string kutuk = openFileDialog1.FileName;
   try
    {
      Resim = new System.Drawing.Bitmap(kutuk);
      pictureBox1.Image = (Bitmap)Resim;
      Resim = null;
    }
  catch
   {
     MessageBox.Show("Uygun dosya seçmediniz");
   }
 }
```

Kullanıcı "Aç" diyalog kutusunda resim dosyası seçmezse veya uygun olmayan bir dosyayı seçerse **catch** bloğundaki satır işletilir ve kullanıcıya bu konuda bilgi verilir. Ancak bu durumda **Bitmap** tipindeki nesne bellekte yaşamaya devam eder. Çünkü "Resim" adını vermiş olduğum Bitmap nesnesini **null** yapan satırı **try** bloğuna yazmıştım.

Hata meydana gelsin veya gelmesin mutlaka işletilmesini istediğiniz satırlar varsa bu satırları **finally** bloğuna yazmalısınız. finally bloğunun nasıl kullanıldığını anlatmak için yukarıda verdiğim kodu aşağıdaki gibi değiştirdim. Bitmap tipindeki değişkeni blok içinde tanımladığım için Garbage Collector zaten bir süre sonra devreye girip bu nesneyi bellekten temizler ama buradaki amacımız deneysel.

```
private void Resim_sec_Click(object sender, EventArgs e)
 {
   System.Drawing.Bitmap Resim;
   openFileDialog1.ShowDialog();
   string Dosya = openFileDialog1.FileName;
   try
    {
      Resim = new System.Drawing.Bitmap(Dosya);
      pictureBox1.Image = (Bitmap)Resim;
    }
    catch
    {
      MessageBox.Show("Uygun dosya seçmediniz");
    }
   finally
    {
       Resim = null;
    }
   }
```

İstenirse birden fazla **catch** bloğu ve/veya iç-içe **try-catch** bloğu hazırlayabilirsiniz. Söz konusu hatanın birden fazla nedenden kaynaklanması ihtimali varsa try bloğu için birden fazla catch bloğu anabilir. Aşağıda iç-içe try-catch bloğunun kalıbı bulunmaktadır.

```
try
{
  try
   {
   }
  catch
    {
    }
  finally
   {
   }
 }
catch
 {
 }
finally
 {
 }
```

Hata Yakalama Sınıfları

C# projelerinde hataları yakalarken kullanabileceğiniz çok sayıda sınıf bulunmaktadır. Yukarıdaki sayfalarda System.Exception, OverflowException, DivideByZeroException ve IndexOutOfRangeException sınıflarından kısaca söz edildi. Hata yakalama sınıflarının işlevleri biraz zor anlaşıldığı için yukarıda kısaca anlattığım konulardan tekrar söz edeceğim. Öncelikle aşağıda verdiğim kodu incelemenizi istiyorum.

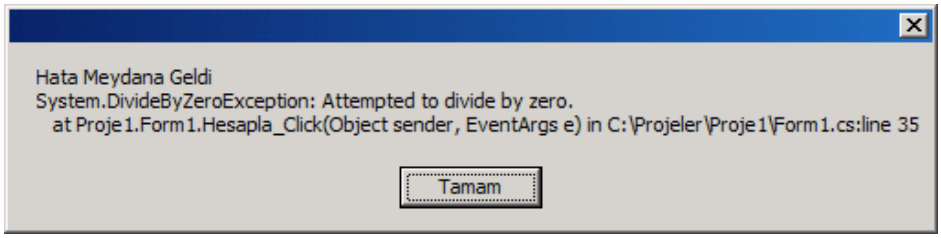
```
private void Hesapla_Click(object sender, EventArgs e)
 {
   try
    {
      byte Sayi1 = Convert.ToByte(textBox1.Text);
      byte Sayi2 = Convert.ToByte(textBox2.Text);
      float Sonuc = Sayi1 / Sayi2;
      textBox3.Text = Sonuc.ToString();
    }
   catch(System.Exception Hata)
    {
      MessageBox.Show("Hata Meydana Geldi" + "\n\r" + Hata);
      textBox1.Focus();
    }
 }
```

Bu metni ilk kaleme aldığımda alışkanlıktan Sayi1, Sayi2 adında değişkenler tanımladım. Başka kaynaklardan yararlanmadan değişkenleri tanımlayabildiğimi kanıtlamak için yukarıda verdiğim kodu aşağıdaki gibi bir değişiklik yaptım.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   try
    {
      byte Number1 = Convert.ToByte(textBox1.Text);
      byte Number2 = Convert.ToByte(textBox2.Text);
      float Netice = Number1 / Number2;
      textBox3.Text = Netice.ToString();
    }
   catch(System.Exception Hataspor)
    {
      MessageBox.Show("Hata Meydana Geldi" + "\n\r" + Hataspor);
      textBox1.Focus();
    }
 }
```

2. TextBox'a sıfır(0) girilip bu kod işletilirse sıfıra bölme hatası meydana gelir(yani catch bloğuna **DivideByZeroException** nesnesi fırlatılır) ve meydana gelen hata ile ilgili olarak ekrana hata mesajı

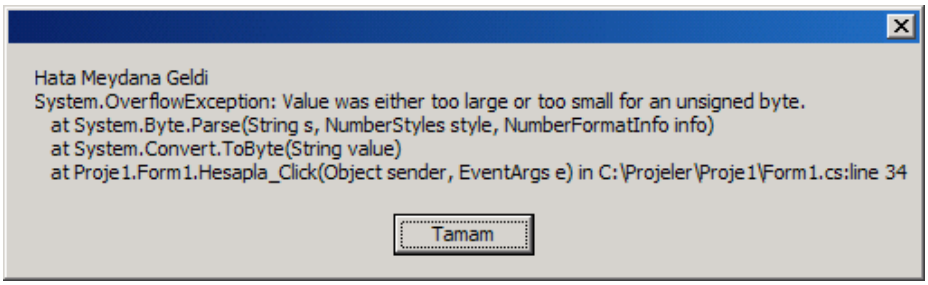
getirilir. Hata nedeniyle ekrana getirilen diyalog kutusundaki mesaja bakarsanız hatanın sıfıra bölmeden kaynaklandığını fark edebilirsiniz.



Bu kodda yukarıdaki sayfalarda verilen örneklerden farklı olarak "Hataspor" adını verdiğim bir nesne var. Bildiğiniz gibi nesneleri **new** anahtar kelimesini kullanıp ilgili sınıfın yapıcı metodundan yararlanarak hazırlıyorduk. Ancak burada "Hataspor" adını verdiğim **System.Exception** tipindeki nesne otomatik olarak hazırlandı. Bu kodda catch bloğuna DivideByZeroException nesnesi gönderilmesine ve catch deyimine ait parantezlerin içinde **System.Exception** tipinde değişken tanımlanmasına rağmen hata yakalandı.

Nasıl ki Object tipindeki değişkenlere istenen tipteki bilgiler aktarılabiliniyorsa bütün hata sınıflarının türetildiği **Exception** sınıfı tipindeki değişkene istenen hata sınıfının örneği aktarılabilinmektedir. try bloğundan catch bloğuna gönderilen DivideByZeroException nesnesi hatasızca dönüştürülüp System.Exception tipindeki değişkene aktarılabildiği için hata yakalanıp catch bloğundaki satırlar işletilebilmektedir.

Sıfıra bölme dışında başka bir hata meydana gelirse ekrana getirilen hata mesajı farklı olur. Çalışma anında TextBox'lardan birisine 255'ten büyük bir değer yazılıp bu kod işletilmiş olunsaydı aşağıdaki gibi hata mesajı ekrana getirilirdi.



Şimdi yukarıda verdiğim kodun **try** bloğunda değişiklik yapıp sıfıra bölme hatası meydana geldiğinde başka mesaj, TextBox'a byte değişkene sığmayacak büyüklükte bir değer girildiğinde başka bir mesajın ekrana getirilmesini sağlayacağım. Aşağıda verilen kodu incelerseniz bu 2 catch bloğunda catch deyimlerine ait parantezlerin arasına sınıfların adını yazmakla yetinip değişken tanımlamadım. try bloğundan gönderilen nesneyi catch bloğu içinde kullanmayacaksanız parantezlerin arasına ilgili sınıfın adını yazmanız yeterlidir.

```
catch(DivideByZeroException)
 {
   MessageBox.Show("Sıfıra bölme hatası yaptınız");
   textBox1.Focus();
 }
catch(OverflowException)
 {
    MessageBox.Show("Büyük sayı girdiğiniz");
    textBox1.Focus();
 }
```

Bu kodu dikkatle incelerseniz **try** bloğundan sonra birden fazla **catch** bloğu olduğunu görürsünüz. Hazırladığım örneğin catch bloklarını bu şekilde değiştirip TextBox'ların birisine karaktersel bilgi

girip bu kodu işletirsem meydana gelen hata yakalanamaz ve programın çalışması kırılır. Çünkü TextBox'a karaktersel bilgi girilip bu kod işletilirse **Convert** sınıfının **ToByte()** metodunun kullanıldığı satırlar hataya neden olur.

```
Number1 = Convert.ToByte(textBox1.Text);
Number2 = Convert.ToByte(textBox2.Text);
```

Convert sınıfı ile string bilgiler byte tipine dönüştürmeye çalışılırken meydana gelebilecek hataları yakalamak istiyorsanız 3. bir catch bloğu hazırlayabilirsiniz. **Convert** sınıfının herhangi bir metodu ile dönüştürme yapılırken hata meydana geldiğinde catch bloğuna **FormatException** sınıfının örneği gönderilmektedir.

Aşağıdaki gibi 3 catch bloğu hazırlayıp hem ilk TextBox'a byte tipindeki değişkene aktarılamayacak büyüklükte bir sayı girip hem de 2. TextBox'a 0 yazıp sıfıra bölme hatasının meydana gelmesini sağlarsanız ilk TextBox'ın içeriği byte değişkene aktarılırken hata meydana geleceği için catch bloğuna **OverflowException** nesnesi gönderilir ve 2. catch bloğu işletilir.

```
catch(DivideByZeroException)
 {
   MessageBox.Show("Sıfıra bölme hatası yaptınız");
   textBox1.Focus();
 }
catch(OverflowException)
 {
   MessageBox.Show("Büyük sayı girdiğiniz");
   textBox1.Focus();
 }
catch (FormatException)
 {
   MessageBox.Show("Yanlış türde bilgi girdiniz");
   textBox1.Focus();
 }
```

Hata yakalama konusuna devam etmek üzere forma bir **PictureBox** yerleştirdim ve aşağıda verdiğim kodu hazırladım. Bu kodda önce "Resim" adını verdiğim **Image** nesnesi için değişken tanımlanmaktadır. Ardından **FromFile**() metodu ile ilgili BMP dosyası okunup **Image** nesnesi hazırlanıp bu nesnenin referansı PictureBox'a aktarılmaktadır.

```
System.Drawing.Image Resim;
Resim = System.Drawing.Image.FromFile("C:\\athena.bmp");
pictureBox1.Image = Resim;
```

FromFile() metoduna parametre olarak verilen dosya bulunmazsa veya içeriği Image nesnesine aktarılmaya uygun olmayan bir dosyanın adı **FromFile**() metoduna parametre olarak verilirse hata meydana gelir. Kodu aşağıdaki gibi düzenlemiş olsaydım hata yakalanırdı.

```
System.Drawing.Image Resim;
try
 {
   Resim = System.Drawing.Image.FromFile("C:\\athena.bmp");
   pictureBox1.Image = Resim;
 }
catch(System.IO.FileNotFoundException)
 {
   MessageBox.Show("Dosya bulunamadı");
 }
```

Bu kodda kullandığım **FileNotFoundException** sınıfı şimdiye kadar hakkında bilgi verilen hata sınıflarından farklı olarak **System.IO**'da tanımlıdır. Bu nedenle ya bu namespace kullanılan sınıfların arandığı namespace listesine **using** deyimi dahil edilmeli veya namespace adı sınıf adından

önce yazılmalıdır. Meydana gelmesi muhtemel olan hata ile ilgili hangi tip Exception nesnesinin oluşturulacağını tahmin edemiyorsanız **Exception** adlı genel sınıfı kullanabilirsiniz.

```
System.Drawing.Image Resim;
try
 {
   Resim = System.Drawing.Image.FromFile("C:\\athena.bmp");
   pictureBox1.Image = Resim;
 }
catch(System.Exception)
 {
   MessageBox.Show("Hata meydana geldi");
 }
```

Bu şartlarda hangi nedenden kaynaklanırsa kaynaklansın herhangi bir hata meydana geldiğinde try-catch bloğundaki **catch(System.Exception)** karşılaştırması doğru değeri vereceğinden bloklanan satırlar işletilir.

Bu durumda Exception nesnesini inceleyip meydana gelen hatanın izini sürebilirsiniz. Bu konuda bilgi vermek için yukarıda verdiğim kodu aşağıdaki gibi düzenledim. Hata meydana geldiğinde bu catch bloğu sayesinde hazırlanan **Exception** nesnesinin hangi sınıftan türetildiği öğrenilip ona göre işlem yapılabilir.

```
catch(System.Exception Hata)
 {
   MessageBox.Show("Hata meydana geldi" + "\n\r" +
     "Orijinal Hata mesajı:"+ Convert.ToString(Hata.GetType()));
 }
```

Yukarıda sözü edilen sınıflardan başka OverflowException, InvalidCastException, DevideByZeroException, FormatException, ArgumentException, ArithmeticException, OutOfMemoryException, ArrayTypeMismatchException, NullReferansException ve MemberAccessException gibi sınıflar da .NET Framework ile birlikte gelmektedir.

InvalidCastException Sınıfı

Daha önceki konulardan bildiğiniz gibi değişik tipteki bilgileri başka bir tipe dönüştürmek mümkündür. Yine operatörler adlı bölümde söz edildiği gibi **as** ve **is** operatörlerinden yararlanıp tip dönüşümleri yapılabilmektedir. Tip dönüştürme işlemleri sırasında hata meydana geldiğinde **InvalidCastException** hatası meydana gelmektedir. InvalidCastException sınıfını anlatmak için aşağıda verdiğim kodu hazırladım.

```
object Nesne = 2008;
try
 {
   int Yil = (int)Nesne; // Tehlikeli bir değişken…
   this.Text = Yil.ToString();
 }
catch (System.InvalidCastException Hataspor)
 {
   MessageBox.Show("Dönüştürme hatası meydana geldi" +
               n\r" + Hataspor.Message);
 }
```

Bu kodda "Nesne" adını verdiğim(yoksa felsefe ve yayıncılık kavramı olan nesnel'i mi kullansaydım!) **object** tipindeki değişkenin içeriği rakamlardan meydana geldiği için **Unboxing** işlemi yapılıp sorunsuz bir şekilde **int** tipindeki değişkene aktarılır. object tipindeki değişkenin içeriği int tipindeki değişkene aktarılırken Unboxing işlemi başarısız olsaydı **InvalidCastException** hatası ile karşılaşılırdı.

Diğer yandan null yapılmış bir nesnenin özelliklerini kullanmak isterseniz **NullReferenceException** hatası ile karşılaşılır. Bu hata ile nasıl karşılaşıldığını göstermek için bir Bitmap nesnesi hazırladım. Ardından bu nesneyi null yapıp **Width** ve **Height** özelliklerini kullanmayı denedim. Aşağıda verdiğim kodda "i" ve "j" tehlikeli değişkenler oldukları için alışkanlıklarımı bir tarafa bırakıp çift "ii" ve çift "jj" tercih ettim.

```
System.Drawing.Bitmap Resim;
openFileDialog1.ShowDialog();
string Dosya = openFileDialog1.FileName;
try
 {
   Resim = new System.Drawing.Bitmap(Dosya);
   Resim = null;
   int ii = Resim.Width;
   int jj = Resim.Height;
 }
catch(NullReferenceException)
 {
   MessageBox.Show("Nesne null yapılmış");
 }
```

Throw Deyimi İle Exception Nesnesi Hazırlamak

Şimdiye kadar anlatılanlardan çıkarmış olabileceğiniz gibi herhangi bir nedenden dolayı çalışma anında bir hata meydana geldiğinde uygulama tarafından karşılaşılan hataya uygun olarak ilgili hata sınıfının örneği hazırlanıp catch bloğuna gönderilmektedir. Bizim yaptığımız, fırlatılan nesnenin hangi sınıfın örneği olduğunu öğrenip ona göre kullanıcıya mesaj verip tedbir almaktı. Burada asıl vurgulamak istediğim konu, sistem bir exception fırlattığında bu Exception nesnesine kaynaklık edecek sınıfı kendisinin seçtiğidir.

Şimdi kendim **throw** deyiminden(anahtar kelime) yararlanıp bir **Exception** nesnesi hazırlayacağım. Bu amaçla kullanıcıdan gün, ay ve yıl bilgilerini isteyip bunları birleştirip tarih bilgisi elde edeceğim. Bu amaçla forma 4 TextBox yerleştirip aşağıda verdiğim kodu hazırladım.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   DateTime Tarih; // Kendinizi garantiye almak için siz "T" deyin
   int gun = Convert.ToInt16(textBox1.Text);
   int ay = Convert.ToInt16(textBox2.Text);
   int yil = Convert.ToInt16(textBox3.Text);
   Tarih=Microsoft.VisualBasic.DateAndTime.DateSerial(yil,ay, gun);
   textBox4.Text = Tarih.ToShortDateString();
 }
```

Kullanıcı TextBox'lara uygun değerleri girdiğinde bu kod sorunsuz çalışır. Ancak gün bilgisi olarak 31'den, ay bilgisi olarak 12'den büyük bir değer girdiğinde gün, ay ve yıl bilgileri birleştirilip tarih elde edilmek istendiğinde hata meydana gelir. Yanlış bilgi girişinden dolayı programın kırılmasını önlemek için yukarıda yapıldığı gibi try-catch bloğu hazırlayabilir veya kullanıcının girdiği bilgileri **DateSerial**() metoduna parametre olarak vermeden önce kendiniz kontrol edebilirsiniz. Pratik değeri olmasa bile aşağıda verdiğim kod hatasız çalışır ve DateSerial() metoduna parametre olarak verilecek ay ve gün bilgilerinin yanlış olması engellenir.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   DateTime History;
   int gun = Convert.ToInt16(textBox1.Text);
   int ay = Convert.ToInt16(textBox2.Text);
   int yil = Convert.ToInt16(textBox3.Text);
   if (gun > 31 || ay > 12)
     try
```

```
      {
        throw new OverflowException("Yanlış bilgi girdiniz");
      }
     catch (Exception Hataspor)
     {
       MessageBox.Show(Hataspor.Message);
       textBox1.Focus();
     }
   History= Microsoft.VisualBasic.DateAndTime.DateSerial(yil, ay, gun);
   textBox4.Text = History.ToShortDateString();
 }
```

Kullanıcı ay veya gün bilgilerinde sınırı aştığında catch bloğu işletilir. try bloğunda şimdiye kadar yapılanlardan farklı olarak **throw** deyimi ile **OverflowException** nesnesi hazırlanmaktadır. Burada throw deyimi ile **Exception** nesnesi hazırlandığı için sıra catch bloğunu işletmeye gelir.

catch bloğunda ise kullanıcıya mesaj verildikten sonra programın işletimi başa alınıp kullanıcıya girdiği bilgileri düzeltme şansı verilir. **throw** deyimi ile **OverflowException** nesnesi yerine başka bir Exception nesnesi hazırlayabilirdim. Hemen hatırlatmak gerekirse, throw anahtar kelimesiyle ilgili exception sınıfının örneğini alma işlemine daha çok kendi özel hata sınıflarını hazırlayan programcılar gerek duymaktadır.

Şimdi throw deyimi ile ilgili 2. bir örnek vereceğim. Bu amaçla **Bol**() adında ve float tipinde 2 parametreye sahip bir metot hazırladım. Bu metot ile birbirine bölünecek sayılar TextBox'lardan alınacaktır. Sıfıra bölme hatası ile ilgili metottaki "sayi1" ve "sayi2" adındaki parametreleri yani değişkenleri sonradan "s1" ve "s2" olarak değiştirdim.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   float bolunen, bolen, sonuc;
   bolunen = Convert.ToSingle(textBox1.Text);
   bolen = Convert.ToSingle(textBox2.Text);
   if (bolen == 0)
    try
    {
      throw new DivideByZeroException("Sıfıra bölme hatası");
    }
    catch (DivideByZeroException Hata)
    {
      MessageBox.Show(Hata.Message);
      textBox2.Focus();
    }
    else
    {
     sonuc = bol(bolunen, bolen);
     textBox3.Text = sonuc.ToString();
    }
  }
 float bol(float s1, float s2)
 {
   return s1 / s2;
 }
```

İlk 2 TextBox'a yazılan bilgileri float tipindeki değişkenlere aktardıktan sonra bölme işlemini yapmadan 2. sayının yani bölenin 0 olup olmadığını araştırdım. İkinci sayı 0 ise programın işletiminin try-catch bloğuna geçmesini sağlayıp **throw** anahtar kelimesi ile DivideByZeroException sınıfının örneğini hazırladım.

Tabii bu kodda TextBox'lara girilen bilgiler **Convert** sınıfının **ToSingle**() metodu ile float tipe dönüştürülmeye uygun değilse başka hatalar meydana gelir. Hazırladığım bu kodda catch bloğunda farklı hata bir sınıfının adını yazsaydım veya **DivideByZeroException** dışında başka bir sınıfın

örneğini hazırlasaydım sıfıra bölme hatası yakalanamazdı. Şimdi farklılık olsun diye try-catch bloğu ile **throw** deyiminin farklı yerlerde olmasını sağlayacağım.

```
private void Hesapla_Click(object sender, EventArgs e)
 {
   float bolunen, bolen, sonuc;
   bolunen = Convert.ToSingle(textBox1.Text);
   bolen = Convert.ToSingle(textBox2.Text);
   try
    {
      sonuc = bol(bolunen, bolen);
      textBox3.Text = sonuc.ToString();
    }
   catch (DivideByZeroException Hata)
    {
      MessageBox.Show(Hata.Message);
      textBox2.Focus();
    }
 }
float bol(float s1, float s2)
 {
   if (s2 == 0)
      throw new DivideByZeroException("Sıfıra bölme hatası");
   else
     return s1 / s2;
 }
```

Bu kodda **throw** anahtar kelimesi ile oluşturduğum DivideByZeroException nesnesinin yalnızca **Message** özelliğiyle ilgilendim. DivideByZeroException ve diğer Exception sınıflarının ayrıca **Source**, **StackTrace**, **TargetSite**, **InnerException** gibi özellikleri bulunmaktadır. Örneğin Source özelliği hatanın meydana geldiği Assembly'ın adını tutmaktadır. Hataya neden olan metotlarla ilgileniyorsanız **StackTrace** özelliğine bakabilirsiniz. Hataya neden olan metotları araştırırken System.Diagnostics'deki **StackTrace** sınıfından yararlanabilirsiniz. Bu sınıfın nasıl kullanıldığını aşağıda görebilirsiniz.

```
StackTrace izle = new StackTrace(Hata);
foreach (StackFrame fr in izle.GetFrames())
 {
   listBox1.Items.Add(fr.GetMethod());
 }
```

Exception Sınıfı Hazırlamak

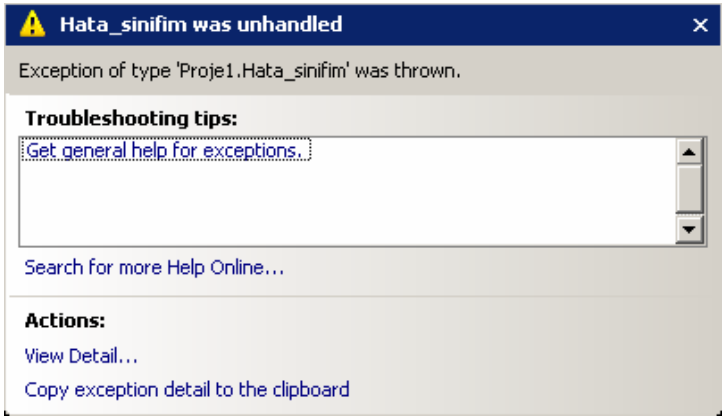
Yukarıdaki sayfalarda sözü edilen ve hata yakalamaya yönelik bütün sınıflar **Exception** sınıfından türetilmiş sınıflardır. Hata sınıfınızı kendiniz hazırlamak istiyorsanız hazırlayacağınız sınıf **Exception** veya **ApplicationException** sınıfının mirasçısı olmalıdır. Programcılar kendi exception sınıflarını özellikle veritabanlarından kaynaklanan hatalar söz konusu olduğunda hazırlama gereğini duymaktadırlar. Bu konuda adım adım bilgi vermek aşağıdaki gibi sınırlı özelliklere sahip bir class hazırladım.

```
public partial class Hata_sinifim : Exception
 {
   public Hata_sinifim()
    {
     }
 }
```

Bu sınıfın yapıcı metodu herhangi bir parametre ve satıra sahip değildir. Bu şekilde kendi Exception sınıfınızı hazırladıktan sonra herhangi bir uygulama dahilinde kullanabilirsiniz. Bu konuda bilgi vermek için forma bir TextBox ve "Tamam" adını verdiğim bir düğme yerleştirip aşağıda verdiğim kodu hazırladım.

```
private void Tamam_Click(object sender, EventArgs e)
 {
   if (textBox1.Text == "12345")
     MessageBox.Show("Sisteme giriş yaptınız");
   else
    throw new Hata_sinifim();
 }
```

Kullanıcı çalışma anında formdaki TextBox'a şifresini girip Tamam adını verdiğim düğmeyi tıklayıp bu kodu işletince TextBox'a yazılan bilginin "12345" olup olmadığı araştırılır. Girilen bilgi 12345 değilse **throw** anahtar kelimesi ile kendi hazırladığım exception sınıfının örneği alınıp fırlatılmaktadır. Kendi hazırladığım exception sınıfı, mirasçısı olduğu sınıftan farklı olmadığı için TextBox'a 12345'ten farklı bir bilgi girilirse programın çalışması kırılır ve sistem aşağıda verilen diyalog kutusunu ekrana getirir.

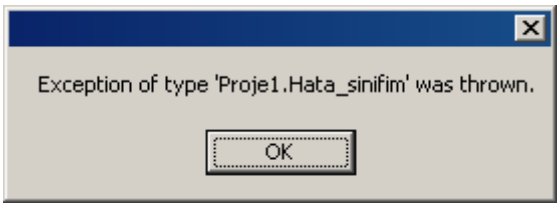


"Tamam" düğmesi için yazılan kodda try-catch bloğuna yer vermiş olsaydım programın çalışması kırılmazdı. Bu amaçla yukarıda verdiğim kodu aşağıdaki gibi düzenledim.

```
private void Tamam_Click(object sender, EventArgs e)
 {
   try
    {
      if (textBox1.Text == "12345")
         MessageBox.Show("Sisteme giriş yaptınız");
      else
         throw new Hata_sinifim();
    }
  catch (Hata_sinifim Hata)
    {
      MessageBox.Show(Hata.Message);
    }
 }
```

TextBox'a 12345'ten farklı bir değer girilip bu kod işletilirse catch bloğundaki satır sayesinde ekrana aşağıdaki gibi hata mesajı getirilirdi. Çünkü try bloğunda kullanıcının TextBox'a şifre niyetine girmiş olduğu bilginin "12345" olup olmadığını araştırdım.

Kullanıcı TextBox'a 12345 dışında başka bilgi girdiyse throw anahtar kelimesi ile kendi hazırladığım sınıfın örneğini aldım. Başka bir deyişle hatalı bir durumu ihbar edip programın akışının catch bloğuna geçmesini sağladım. catch bloğunda ise throw anahtar kelimesi ile hazırlanan exception nesnenin orijinal mesajını ekrana getirdim.



Kendi hazırladığınız hata sınıfı ile ilgili olarak ekrana kendi hata mesajınızı getirmek istiyorsanız sınıfının yapıcı metodunu aşağıdaki gibi değiştirebilirsiniz. Yaptığım değişiklik sayesinde kendi hazırladığım hata sınıfının yapıcı metodunun string tipte bir parametreye sahip olmasını sağladım.

```
public Hata_sinifim(string msg) : base(msg)
 {
 }
```

Kendi hazırladığınız exception sınıfının yapıcı metodunu bu şekilde değiştirdikten sonra **throw** anahtar kelimesi ile Exception nesnesini fırlatırken sınıfın yapıcı metoduna aşağıdaki gibi mesaj olarak kullanmak istediğiniz metni parametre olarak vermelisiniz.

```
throw new Hata_sinifim("Yanlış Şifre Girdiniz");
```

Yukarıda kendi hazırladığım hata yakalama sınıfına taban sınıf olarak **Exception** sınıfını seçtim. Ne ki kendi hazırladığınız istisnai durum sınıfları için Exception yerine **ApplicationException** sınıfını başlangıç noktası olarak seçerniz işiniz kolaylaşcaktır.

Bu konunun üzerinde biraz durabilmenizi sağlamak için şimdi ikinci bir Exception sınıfı hazırlayacağım. Bu amaçla hazırladığım sınıf 4 yapıcı metoda sahip olacak ve **ApplicationException** sınıfını mirasçısı olacaktır. Kendi hazırladığım Exception sınıfta 4 yapıcı metoda yer vermemin nedeni taban sınıf olarak seçtiğim ApplicationException sınıfın 4 yapıcı metoda sahip olmasıdır.

```
public partial class Hata_sinifim : ApplicationException
 {
  private string mesajim;
  public string ozellik_mesajim
   {
      get { return mesajim; }
      set { mesajim = value; }
   }
  public Hata_sinifim() : base()
   {
   }
  public Hata_sinifim(string mesaj) : base(mesaj)
   {
   }
  public Hata_sinifim(string mesaj, Exception inner) :base(mesaj, inner)
   {
   }
  public Hata_sinifim(string mesaj, string mesajim) : base(mesaj)
   {
     ozellik_mesajim = mesajim;
   }
 }
```

İlk bu şekilde düzenlediğim sınıfın 2. parametresi Exception tipinde olan yapıcı metottan söz edeceğim. Yukarıdaki sayfalarda Exception nesnesinin InnerException özelliğinin adı sayılmıştı. Bu özellik, catch bloğuna yazılan kodlar hataya neden olduğunda işlevsel olmaktadır. Bu konuda bilgi vermek için üzerinde çalıştığım projede bu sınıftan aşağıdaki gibi yararlandım. Tabii bu örneğin pratik bir değerinin olmadığını hemen söylemek gerek. **InnerException** özelliğinden söz etmek için aşağıdaki gibi 2. sınıf hazırladım.

```
public class Test_sinifi
 {
  public void Hatayi_uret()
    {
      try
       {
        this.inner_uret();
       }
      catch (Exception H)
       {
        throw new Hata_sinifim("Normal hatamız", H);
       }
    }
   public void inner_uret()
    {
      throw new Hata_sinifim("inner(önceki) hatamız");
    }
 }
```

Oldukça basit olan bu sınıfı incelerseniz yapıcı metoda sahip olmayıp Hatayi_uret() adını verdiğim **void** metotta try-catch bloğu bulunmaktadır. Eğer bu örnekte işlevsellik kaygım olsaydı try-catch bloğuna hataya neden olma ihtimali yüksek satırlar yazardım. Bunu yapmak yerine catch bloğunda hata üretilen veya kendi hazırladığım exception sınıfının örneğinin alındığı inner_uret() metodunu çağırdım. Bu metotta zaten hata üretilmesine rağmen catch bloğunda kendi exception sınıfımın ikinci bir örneğini aldım. Başka bir deyişle bu sınıf sayesinde 2 hatanın arka arkaya meydana gelmesi sağlanmaktadır. Şimdi ise "Test_sinifi" adını verdiğim bu sınıfın örneğini alacağım. Bu amaçla üzerinde çalıştığım Windows Forms uygulamasında aşağıdaki gibi bir hazırlık yaptım.

```
private void button1_Click(object sender, EventArgs e)
 {
  Test_sinifi test_nesne = new Test_sinifi();
  try
   {
     test_nesne.Hatayi_uret();
   }
  catch (Exception hata)
   {
     MessageBox.Show(hata.InnerException.Message);
     MessageBox.Show(hata.Message);
   }
}
```

Hata yakalama ve Exception sınıfları hakkında yazacaklarım bu kadarla sınırlı olmamasına rağmen makale daha da uzamasın diye burada kestim.

Memik YANIK
memiky@superonline.com
memikyanik@hotmail.com
www.memikyanik.com