# C# - Felhantering

## Felhantering

För att undvika att vårt program kraschar när exempelvis en användare matar in otillåtet input bör vi implementera felhantering i programmet. Detta gör vi enklast med hjälp av så kallade *try/catch*-block. Dessa block tillåter oss att testa koden om den fungerar och kör den om så är fallet, och om inte så fångar den vårt error och låter oss bestämma vad som ska hända. Till sist kommer vi gå igenom *Parse* och *tryParse* som vi använder för att översätta strängar till numeriska datatyper.

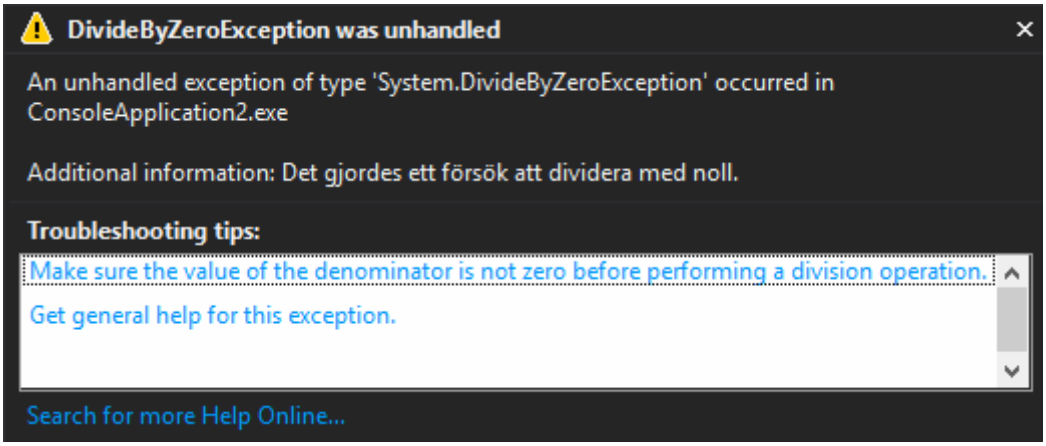
## Try/Catch/Finally

I det här exemplet skapar vi en applikation vars enda uppgift är att dividera två tal och skriva ut summan. Koden ser ut så här:

```
int tal1 = 10;
int tal2 = 0;
int summa = tal1 / tal2;

Console.WriteLine(summa);
Console.ReadLine();
```

Som vi vet kan vi inte dividera med noll. Om vi kör programmet nu får vi error-meddelande:




Här får vi meddelandet ”DivideByZeroException was unhandled” som berättar att vi försökt dividera med noll. I vårat fall kan vi ju bara ändra tal2 att vara något annat än noll, men i många fall handlar det om inmatning från användare och detta måste då förebyggas så programmet inte kraschar vid användning och det gör vi med hjälp av ett *try/catch*-block.

Ett *try/catch*-block består av nyckelorden *try*, *catch* och *finally*.
*Try* – Inuti *try* placeras koden vi ska försöka köra.
*Catch* – *Catch* kräver ett argument med ett Exception. Här inne bestämmer vi vad vi ska göra vid ett exception.
*Finally* – Koden inuti här körs efter någon av try eller catch oavsett vad som hänt. *Finally* är valfritt att inkludera.

Så här ser uppbyggnaden ut för en *try/catch/finally*:

```
try
{
    // Kod som ska testas
}
catch (Exception e)
{
    // Här bestämmer vi vad vi vill göra
    // om try-koden inte fungerar.
    // Dvs vi får ett exception
}
finally
{
    // Här skriver vi kod vi vill ska köras
    // oavsett vad som händer.
    // Finally-blocket är valfritt att ha med.
}
```

```
catch (DivideByZeroException e)
{
    // Körs om vi delar med noll
}
catch (IndexOutOfRangeException e)
{
    // Körs om exempelvis index > array längd
}
catch (NullReferenceException e)
{
    // Körs om vi refererar till ett tomt objekt
}
catch (Exception e)
{
    // Körs om inget av de ovan stämmer
}
```

På bilden åt höger ser ni att vi använt oss av flera *catch*-argument, detta är för att demonstrera att vi kan ha flera *catch*-block i en *try/catch*. Som ni ser finns det flera olika typer av exceptions(ni kan googla och se hur många som finns) och dessa specificerar vi när vi vet vilka exceptions vi kan få så att vi i *catch*-blocket kan skriva specifik kod just mot det felet. I variabelnamnet ”e” finns beskrivningen på felet. Denna kan vi skriva ut direkt till vår konsol eller i en logg-fil om vi har en sådan.

Nu ska vi applicera en *try/catch/finally* till vårat tidigare exempel och då ser det ut så här:

```
int tal1 = 10;
int tal2 = 0;
int summa = 0;

try
{
    // Försök att dividera våra tal
    summa = tal1 / tal2;
}
catch (DivideByZeroException e)
{
    // Här kan vi få ett DivideByZeroException och vi väljer
    // att skriva ut felmeddelandet till vår konsol
    Console.WriteLine(e);
}
finally
{
    // Till slut vill vi skriva ut summan oavsett vad som händer
    Console.WriteLine("Summan är: " + summa);
}

Console.ReadLine();
```

Vi delat fortfarande tal1 med noll, så vi vet att vi kommer få ett felmeddelande. Här väljer vi att skriva ut felmeddelandet till vår konsol för att visa felet för användaren.

Vår output till konsolen:

```
System.DivideByZeroException: Det gjordes ett försök att dividera med noll.
   vid ConsoleApplication2.Program.Main(String[] args) i C:\Users\     \Desktop\
School\C#\ConsoleApplication2\ConsoleApplication2\Program.cs:rad 20
Summan är: 0
```

Här skrivs felmeddelandet ut och berättar att vi försökte dividera med noll. Därefter på en ny rad skrivs ”Summan är: 0” ut och programmet är fortfarande igång. Vi har alltså hanterat alla eventuella fel som kan uppstå i vårt program.

Ett dåligt program kraschar vid fel, ett bra program hanterar dem.
Viktigt att ni lägger detta på minnet!

## Parse och TryParse

När vi hanterar inmatning från användare, exempelvis från konsolen eller en textbox, så kommer inmatningen som en sträng. I en applikation där vi frågat användaren om ett tal att multiplicera blir detta ett problem då vi inte kan utföra aritmetik på text(Hexadecimal krävs lite extra kod). Med hjälp av parsing kan vi översätta strängar till numeriska datatyper.

Den enklaste att utföra är ”*datatyp.Parse(strängtext);*” Den ser ut så här:

```
string userInput = "1234";

int parsedInput = int.Parse(userInput);

Console.WriteLine(parsedInput);
```

```
string userInput = "1234x";

int parsedInput = int.Parse(userInput);

Console.WriteLine(parsedInput);
```

I den vänstra bilden ser ni att strängen userInput bara innehåller siffror medans i den högra avslutas strängen med ett ”x”. Då den vänstra bara innehåller siffror lyckas int.Parse översätta strängen till en integer utan problem, däremot i den högra bilden där ett ”x” existerar i strängen får vi ett exception då strängen inte längre endast består av numeriska värden. Använd därför bara en Parse på ställen vi är säkra att vi bara kan få numeriska värden.

Måste vi ta input från en användare(där det kan bli fel) så kan vi använda oss av en *TryParse*.

```
string userInput = "1234x";
int parsedInput;
// In med userInput, om det lyckas, ut med värdet i parsedInput
bool resultat = int.TryParse(userInput, out parsedInput);

if(resultat)
    Console.WriteLine(parsedInput);
else
    Console.WriteLine("Gick ej att översätta från sträng till int");
```

En *TryParse* är en bool, som blir true eller false baserat på om parsningen lyckades eller inte. En *TryParse* fungerar genom att ta in ett värde, försöker översätta den till numeriskt värde och om den lyckas sparas den i parsedInput och bool result blir true. Annars blir result falskt och programmet fortsätter utan att ge oss ett exception.