Kapitel 9

Undantagshantering

9.1 Automatiskt genererade undantag

Undantag (eng. *exception*) betyder i programmering *fel*, närmare bestämt exekveringsfel som uppstår när datorns processor inte kan utföra programmets instruktioner (sid 14). Dessa fel syns inte vid kompilering. De leder ”bara” till att i bästa fall programmet och i sämsta fall datorn kraschar, när ett program som inte har några syntaxfel, exekveras. I regel är orsaken okänd och inte lätt att spåra just vid exekveringstillfället. Däremot borde man kunna förutse dem när man skriver kod. Till god programmeringsstil hör att man tar hand om ”farlig kod” redan när man programmerar. Det gäller förstås att kunna förutse i vilka situationer fel kan inträffa vid exekveringen. I så fall borde man bygga in en felhantering i koden. Alla moderna programmeringsspråk ställer verktyg till förfogande för felhantering som kallas *undantagshantering* (eng. *exception handling*). Detta kapitel introducerar bara de mest elementära begreppen och metoderna för undantagshantering i C#.

I programmet **`SimpleIf`** (sid 89) hade vi redan skrivit en egen felhantering. Programmet läste in två tal och dividerade dem med varandra. Men koden tillät division endast om det andra talet inte var `0`. Detta för att förhindra den matematiskt odefinierade divisionen med `0`. Inmatning av `0` till det andra talet genererade ”felmeddelandet”:

```
OBS!
Du har matat in 0 för det andra talet.
Det går inte att dividera med 0.
```

Programmeringstekniskt löste vi problemet då med två enkla **`if`**-satser. Nu ska vi försöka att göra det med de verktyg för undantagshantering som är inbyggda i C#.

```
// TryCatchTest.cs
// Förhindrar programavbrott med ett try catch-block
using System;

class TryCatchTest
{
   static void Main()
   {
      int no1 = 8, no2 = 0, div;

      try                                        // try catch-blocket
      {
         div = no1 / no2;
      }
      catch
      {
         Console.WriteLine("\n\tOBS! " +
                           "Du försökte dividera med 0.\n\t" +
                           "Det går inte att dividera med 0.");
      }
      Console.WriteLine("\n\tHär fortsätter programmet! \n");
   }
}
```

Det reserverade ordet **try** säger: ”Försök att …” dvs försök att utföra det block av satser som följer, i det här fallet försök att utföra satsen **div = no1 / no2;** som innebär att dela **8** med **0**. Om det uppstår något fel (undantag) ”fånga upp” – i koden **catch** – felet genom att utföra det block av satser som följer efter **catch**. Man har friheten att skriva i **catch**-blocket allt man önskar att det ska ske om **try**-blocket ”kastar ett undantag” dvs ger upphov till ett fel. Därför kallas hela konstruktionen **try catch**-blocket och är undantagshanteringens grundkoncept.

För att förenkla testet har vi i programmet ovan tilldelat **0** direkt till **no2** för att provocera fram undantaget **DivideByZeroException** som är ett fördefinierat undantag och samtidigt en subklass med samma namn i klassen **Exception**. Detta undantag genereras automatiskt av koden **no1/no2** när **no2** har värdet **0**. Hade vi inte hanterat detta undantag genom att placera koden i **try**-blocket och fånga upp det i **catch**-blocket, hade vi fått följande felmeddelande vid exekvering av programmet **TryCatchTest**:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at TryCatchTest.Main() in c:\C#\MyProject\TryCatchTest.cs:line 14
```

Testa gärna detta genom att kommentera bort hela **try catch**-blocket men behålla satserna **div = no1/no2;** och **Console.WriteLine("\n\tHär fortsätter programmet! \n");**. Samtidigt med felmeddelandet ovan avbryts programkörningen abrupt. Resten av programmet exekveras inte. Hade den kod som kastar undantaget stått i början av ett längre program hade stora mängder kod inte exekverats.

Om vi däremot hanterar undantaget som i **TryCatchTest** sker inget oväntat programavbrott. Istället exekveras koden i **catch**-blocket. Sedan fortsätter programflödet efter **catch**-blocket, resten av koden exekveras och programmet slutförs på ett regulärt sätt. Körresultatet av programmet **TryCatchTest** visar detta:

```
OBS! Du försökte dividera med 0.
Det går inte att dividera med 0.

Här fortsätter programmet!
```

Observera att programflödet inte återgår till den punkt tillbaka där undantaget kastades i **try**-blocket utan fortsätter linjärt, dvs efter **catch**-blocket. Så kod som står efter den ”farliga koden” i **try**-blocket exekveras endast om inget undantag inträffar. Testa gärna själv genom att i programmet **TryCatchTest** lägga in någon utskriftssats i slutet av **try**-blocket. Den sats kommer inte att utföras eftersom **no1/no2** genererar undantag.

Det finns en uppsjö av automatiskt genererade undantag i C# som är fördefinierade i subklasser till klassen **Exception** som finns i namnutrymmet **System**. Varje gång ett undantag inträffar skapas ett objekt av en sådan klass där all information om undantaget lagras. Andra exempel på automatiskt genererade undantag är **IndexOutOfRangeException** som inträffar när man överskrider en arrays gränser (sid 198) och **NullReferenceException** som uppstår när man använder en referens som har värdet **null** dvs inte pekar på något objekt (sid 146).

9.2 Egengenererade undantag

Undantaget **DivideByZeroException** var i programmet **TryCatchTest** (sid 232) automatiskt genererad, förorsakat av koden **no1/no2** och av att variabeln **no2** hade värdet **0**. Men det finns i C# också möjligheten att programmeraren själv genererar ett undantag vilket ger oss friheten att kontrollera våra program med avseende på tillförlitlighet och stabilitet av kod. Detta kan man göra bl.a. med det reserverade ordet **throw** (eng. att kasta). Att kasta ett undantag betyder att generera ett sådant, vilket man kan göra genom att sätta **throw** framför ett objekt av någon undantagsklass. Följande program demonstrerar detta:

```
// ThrowTest.cs
// Kastar ett undantag med throw och hanterar det med try catch
using System;

class ThrowTest
{
   static double SafeDiv(double no1, double no2)        // Metod
   {
      if (no2 == 0)
         throw new DivideByZeroException(); // Undantag kastas
      else                                  // Objekt skapas
         return no1 / no2;
   }

   static void Main()
   {
      try                                   // Undantag hanteras
      {
         Console.WriteLine(SafeDiv(8, 0));             // Anrop
      }
      catch(DivideByZeroException e)        // catch + parameter
      {
         Console.WriteLine(e.ToString());   // Undantag skrivs ut
      }
   }
}
```

throw-satsen

new DivideByZeroException() är ett objekt av typ **DivideByZeroException**. Genom att sätta **throw** framför det genereras (kastas) ett sådant undantag:

```
throw new DivideByZeroException();
```

Denna sats ersätter koden **no1/no2** som i förra avsnitt förorsakade det automatiskt genererade undantaget. Därför är denna kod flyttad efter **else** och utförs därmed endast om **no2** inte är lika med **0**. Satsen är inbyggd i metoden **SafeDiv()** som anropas i

try-blocket. Därmed genereras undantaget där, vilket länkar programflödet till **catch**-blocket. Huvudet till **catch**-blocket ser här annorlunda ut:

```
catch(DivideByZeroException e)
```

Det ser ut som en metod med en parameterlista i vilken en referens **e** definieras till det ovan skapade undantagsobjektet av typ **DivideByZeroException**. Vi har alltså att göra med en annan variant av **catch** jämfört med programmet **TryCatchTest** (sid 232) där **catch** saknade parameterlista. Med hjälp av referensen **e** som pekar på det kastade undantagsobjektet kan vi nu i **catch**-blocket anropa objektets **ToString()**-metod:

```
Console.WriteLine(e.ToString());
```

Detta anrop resulterar i följande utskrift av programmet **ThrowTest** :

```
System.DivideByZeroException: Attempted to divide by zero.
   at ThrowTest.SafeDiv(Double no1, Double no2) in
c:\C#\MyProject\ThrowTest.cs:line 10
   at ThrowTest.Main() in c:\C#\MyProject\ThrowTest.cs:line 19
```

Observera att detta inte är ett felmeddelande, därför att vi har ju hanterat undantaget **DivideByZeroException** i **try catch**-blocket och skrivit ut dess **ToString()**-metod. **ToString()** är en strängrepresentationsmetod definierad i en superklass som ärvs av alla fördefinierade klasser, så även av klassen **DivideByZeroException**. Därför kan vi använda den med referensen **e** som pekar på det kastade undantagsobjektet av denna klass. Metoden **ToString()** innehåller objektets fullständiga information i strängform. Genom att anropa den i utskriftssatsen ser vi denna information. Den anger först sin källa: **System.DivideByZeroException**. Sedan talar den om vilken typ av undantag det rör sig om: **Attempted to divide by zero**. Resten av informationen handlar om var exakt i programmet undantaget inträffade.

Samma information som vi får med **e.ToString()** ges vidare till det felmeddelande som automatiskt skrivs ut om vi inte hanterar undantaget. Den enda skillnaden är att det hela inleds då med att det är ett ohanterat undantag:

```
Unhandled Exception: ...
```

Då hade detta varit ett verkligt felmeddelande.

Man kan ju undra vilken praktisk relevans programmet **ThrowTest** har och varför och i vilka situationer man använder **throw**-satsen. När ska man låta C# upptäcka möjliga fel och generera undantag automatiskt och när ska vi skriva kod för att själva kasta och hantera undantag? Alla dessa frågor är inte bara berättigade utan också intressanta. Problemet är bara att deras svar spränger *Programmering 1*:s ramar. Programmet **ThrowTest** har endast pedagogisk relevans. Dess uppgift är att ge en första introduktion till de mest elementära grundbegreppen och metoderna inom undantagshantering.