Inhoudsopgave

Zakhandboek C# 8.0

C# is een universele, typeveilige, objectgeoriënteerde programmeertaal met als doel de productiviteit van de programmeur te verhogen. Daartoe biedt de taal een evenwichtige mix van eenvoud, expressiviteit en performance. C# 8 is ontworpen voor Microsoft .NET Core 3 runtime en .NET Standard 2.1 (C# 7 was bedoeld voor Microsoft .NET Framework 4.6/4.7/4.8, .NET Core 2.x en .NET Standaard 2.0).

De programma's en codefragmenten in dit boek zijn hetzelfde als die in de hoofdstukken 2 tot en met 4 van *C# 8.0 in a Nutshell*; ze zijn allemaal te vinden in de interactieve voorbeelden in LINQPad. Aan de hand van deze voorbeelden en het boek leert u sneller omdat u de voorbeelden nu kunt bewerken en de resultaten direct kunt zien zonder de projecten en code in Visual Studio op te hoeven zetten.

Klik om de voorbeelden te downloaden op het tabblad **Samples** in LINQPad en vervolgens op **Download more samples**. LINQPad is gratis; ga naar **www.linqpad.net**.

Een eerste C#-programma

Hier is een programma dat 12 met 30 vermenigvuldigt en het resultaat, 360, op het scherm weergeeft. De dubbele schuine streep geeft aan dat de rest van de regel *commentaar* is:

```
using System;                 // Importeer de namespace

class Test                    // Classdeclaratie
{
  static void Main()          // Methodedeclaratie
  {
    int x = 12 * 30;          // Statement 1
    Console.WriteLine (x);    // Statement 2
  }                           // Einde van de methode
}                             // Einde van de class
```

De kern van dit programma bestaat uit twee statements. Statements worden in C# opeenvolgend uitgevoerd en afgesloten met een puntkomma. In de eerste statement wordt de *expressie* `12 * 30` berekend en het resultaat opgeslagen in een lokale *variabele* `x` van het type integer. In de tweede statement wordt de *methode* `WriteLine` van de class `Console` aangeroepen om de variabele `x` weer te geven in een tekstvenster op het scherm.

Met een *methode* wordt een handeling uitgevoerd door middel van een reeks statements, een *statementblok* genoemd. Zo'n blok staat tussen één paar accolades en bevat nul of meer statements. We hebben een enkele methode met de naam `Main` gedefinieerd.

Door higher-level functies te schrijven die lower-level functies aanroepen, kunnen we een programma vereenvoudigen. We kunnen ons programma als volgt herstructureren (*refactor*) met een herbruikbare methode waarin een geheel getal met 12 wordt vermenigvuldigd:

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));    // 360
    Console.WriteLine (FeetToInches (100));   // 1200
  }
```

```
  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Een methode met een retourtype ontvangt *invoergegevens* van de aanroepende functie door middel van *parameters*, en voert de *uitvoergegevens* terug naar de aanroepende functie door een *retourtype* op te geven. In het vorige voorbeeld hebben we een methode gedefinieerd met de naam `FeetToInches` die een parameter heeft voor de input van feet en een retourtype voor de output van inches, beide van het type `int` (integer).

De vaste waarden `30` en `100` zijn de argumenten die worden doorgegeven aan de methode `FeetToInches`. De methode `Main` in ons voorbeeld heeft lege haakjes omdat deze geen parameters heeft; hij is `void` omdat hij geen waarde naar de caller (aanroepende methode of functie) retourneert. C# herkent een methode met de naam `Main` als standaard startpunt van uitvoering. De methode `Main` kan optioneel een integer retourneren (in plaats van `void`) om een waarde naar de uitvoeringsomgeving te retourneren. De methode `Main` accepteert eventueel ook een array strings als parameter (die dan wordt gevuld met argumenten die aan de executable worden doorgegeven). Bijvoorbeeld:

```
static int Main (string[] args) {...}
```

Een array (zoals `string[]`) vertegenwoordigt een vast aantal elementen van een bepaald type (zie *Arrays* op pagina 39).

Een methode is een van de verschillende soorten functies in C#. Een ander soort functie die we gebruikten, is de *operator* `*`, die vermenigvuldiging uitvoert. Er zijn ook *constructors*, *properties*, *events*, *indexers* en *finalizers*.

In ons voorbeeld zijn de twee methoden opgenomen in een *class* (klasse). Met een class construeren we een objectgeoriënteerde bouwsteen met functie- en datamembers. De class `Console` doet dat met de members die de invoer-/uitvoerfunctionaliteit van de opdrachtregel afhandelen, zoals de methode `WriteLine`. Onze class `Test` neemt twee methoden: de methode `Main` en de methode `FeetToInches`. Een class is een soort *type*, dat we later in *Basiskennis van typen* (pagina 17) nader gaan bekijken.

Op het 'buitenste' niveau van een programma zijn typen geordend in *namespaces*. Met het directief `using` stellen we de namespace `System` ter beschikking van onze

applicatie zodat deze de class `Console` kan gebruiken. We kunnen al onze classes binnen de namespace `TestPrograms` als volgt definiëren:

```
using System;

namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

De .NET Core-bibliotheken zijn geordend in geneste namespaces. Dit is bijvoorbeeld de namespace die typen bevat voor het verwerken van tekst:

```
using System.Text;
```

Het directief `using` is er voor het gemak; u kunt ook naar een type verwijzen met de volledig gekwalificeerde naam, namelijk de typeaanduiding voorafgegaan door de namespace, zoals `System.Text.StringBuilder`.

Compilatie

De C#-compiler compileert broncode, gespecificeerd als een verzameling bestanden met de extensie .cs, tot een *assembly*, de packaging en deployment unit in .NET. Een assembly kan een applicatie of een bibliotheek zijn, met dit verschil dat een applicatie een startpunt heeft (de methode `Main`), terwijl een bibliotheek dat niet heeft. Het doel van een bibliotheek is dat deze aangeroepen kan worden (via referentie) door een toepassing of andere bibliotheken. De .NET Core-runtime (en het .NET Framework) bestaan uit een verzameling bibliotheken.

Voor het werken met de compiler kunt u een geïntegreerde ontwikkelomgeving (IDE) gebruiken, zoals Visual Studio of Visual Studio Code, of deze handmatig aanroepen vanaf de opdrachtregel (CLI). Om handmatig een consoletoepassing met .NET Core te compileren, downloadt u eerst de .NET Core SDK en maakt u als volgt een nieuw project aan:

```
dotnet new console -o MyFirstProgram
cd MyFirstProgram
```

Hiermee maakt u een map aan met de naam MyFirstProgram, die een C#-bestand met de naam Program.cs bevat dat u vervolgens kunt bewerken. Om de compiler te starten, roept u `dotnet build` aan (of `dotnet run`, waarmee het programma wordt gecompileerd en uitgevoerd). De uitvoer wordt naar een submap onder *bin\debug* weggeschreven, die *MyFirstProgram.dll* (de uitvoer-assembly) en *MyFirstProgram.exe* (die het gecompileerde programma rechtstreeks uitvoert) bevat.

Syntaxis

De C#-syntaxis leunt sterk op die van C en C++. In dit hoofdstuk behandelen we de syntaxiselementen van C# aan de hand van het volgende programma:

```
using System;

class Test
{
  static void Main()
  {
    int x = 12 * 30;
    Console.WriteLine (x);
  }
}
```

Identifiers en keywords

Identifiers zijn namen die programmeurs kiezen voor hun classes, methoden, variabelen enzovoort. Dit zijn de identifiers in ons voorbeeldprogramma, in volgorde van waarin ze optreden:

```
System   Test   Main   x   Console   WriteLine
```

Een identifier moet een heel woord zijn, hoofdzakelijk bestaande uit Unicode-tekens en beginnend met een letter of underscore. C#-identifiers zijn hoofdlettergevoelig. Volgens afspraak moeten parameters, lokale variabelen en private velden in lower camelCase gezet worden (bijvoorbeeld `myVariable`) en alle andere identifiers in PascalCase (bijvoorbeeld `MyMethod`).

Keywords zijn namen die iets speciaals betekenen voor de compiler. Dit zijn de keywords van ons voorbeeldprogramma:

```
using   class   static   void   int
```

De meeste keywords zijn *gereserveerde woorden*, wat betekent dat u ze niet als identifier kunt gebruiken. Hier is de volledige lijst met C#-keywords:

```
abstract        as              base            bool
break           byte            case            catch
char            checked         class           const
continue        decimal         default         delegate
do              double          else            enum
event           explicit        extern          false
```

```
finally         fixed           float           for
foreach         goto            if              implicit
in              int             interface       internal
is              lock            long            namespace
new             null            object          operator
out             override        params          private
protected       public          readonly        ref
return          sbyte           sealed          short
sizeof          stackalloc      static          string
struct          switch          this            throw
true            try             typeof          uint
ulong           unchecked       unsafe          ushort
using           virtual         void            while
```

Conflicten vermijden

Als u echt een identifier wilt gebruiken die botst met een gereserveerd keyword, kunt u dit doen door het vooraf te laten gaan door het voorvoegsel `@`. Bijvoorbeeld:

```
class class  {...}      // Verboden!
class @class {...}      // Dit mag wel
```

Het `@`-teken maakt geen deel uit van de identifier zelf. Dus `@myVariable` is hetzelfde als `myVariable`.

Contextuele keywords

Sommige keywords zijn *contextueel*, wat betekent dat ze ook kunnen worden gebruikt als identifier, zonder een @-teken. De contextuele keywords zijn:

```
add             alias           ascending       async
await           by              descending      dynamic
equals          from            get             global
group           into            join            let
nameof          on              orderby         partial
remove          select          set             value
var             when            where           yield
```

Bij contextuele keywords is er geen sprake van meerduidigheid binnen de context waarin ze worden gebruikt.

Literals, punctuators en operatoren

Literals zijn stukjes primitieve data die letterlijk in het programma zijn ingebed. De literals in ons voorbeeldprogramma zijn `12` en `30`. Met *punctuators* bakenen we de structuur van het programma af. De punctuators (leestekens) in ons programma zijn `{`, `}` en `;`.

Met accolades vatten we meerdere statements samen in een statementblok. De puntkomma beëindigt een (niet-blok) statement. Statements kunnen meerdere regels bevatten:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Een *operator* transformeert en combineert expressies. De meeste operatoren in C# worden aangeduid met een symbool, zoals de operator voor vermenigvuldiging, `*`. Dit zijn de operatoren in ons programma:

```
. ()   *   =
```

Een punt geeft een member van iets aan (of een decimale punt bij numerieke literals). De haakjes verschijnen in ons voorbeeld waar we een methode declareren of aanroepen; lege haakjes betekenen dat de methode geen argumenten accepteert. Het isgelijkteken zorgt voor toewijzing (het dubbele isgelijkteken, `==`, voert een gelijkheidsvergelijking uit).

Commentaar

C# biedt twee verschillende stijlen van broncodedocumentatie: *commentaar binnen één regel* en *commentaar over meerdere regels*. Commentaar binnen één regel begint met een dubbele forward slash en gaat door tot het einde van de regel. Bijvoorbeeld:

```
int x = 3;   // Commentaar over de toewijzing van 3 aan x
```

Een commentaar over meerdere regels begint met `/*` en eindigt met `*/`. Bijvoorbeeld:

```
int x = 3;   /* Dit is commentaar dat
                twee regels beslaat */
```

Commentaar kan XML-documentatietags insluiten (zie *XML-documentatie* op pagina 220).

Basiskennis van typen

Een type definieert de blauwdruk voor een waarde. In ons voorbeeld gebruiken we twee literals van het type `int` met de waarden 12 en 30. We hebben ook een variabele van het type `int` met de naam `x` gedeclareerd.

Een *variabele* verwijst naar een opslaglocatie die mettertijd verschillende waarden kan bevatten. Een *constante* vertegenwoordigt daarentegen altijd dezelfde waarde (hierover later meer).

Alle waarden in C# zijn een *instantie* van een specifiek type. De betekenis van een waarde, en van de reeks mogelijke waarden die een variabele kan hebben, wordt bepaald door zijn type.

Voorbeelden van voorgedefinieerde typen

Voorgedefinieerde typen (ook *ingebouwde typen* genoemd) zijn typen die speciaal door de compiler worden ondersteund. Het type `int` is een voorgedefinieerd type voor het weergeven van de verzameling gehele getallen die in 32-bitsgeheugen passen, van -2^{31} tot $2^{31}-1$. We kunnen functies, zoals rekenen met instanties van het type `int`, als volgt uitvoeren:

```
int x = 12 * 30;
```

Een ander voorgedefinieerd C#-type is `string`. Het type `string` staat voor een reeks tekens, zoals ".NET" of "http://oreilly.com". We kunnen als volgt door middel van functies met strings werken:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);      // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message);         // Hello world2015
```

Het voorgedefinieerde type `bool` heeft precies twee mogelijke waarden: `true` en `false`. Het type `bool` wordt gewoonlijk gebruikt om de uitvoeringsflow voorwaardelijk te vertakken met een `if`-statement. Bijvoorbeeld:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");
```

```
int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

De namespace `System` in .NET Core bevat tal van belangrijke typen die niet voorgedefinieerd zijn door C# (zoals `DateTime`).

Voorbeelden van eigen typen

Net zoals we complexe functies kunnen bouwen op basis van eenvoudige functies, kunnen we complexe typen bouwen op basis van primitieve typen. In dit voorbeeld definiëren we een eigen type (*custom type*) met de naam `UnitConverter`, een class die dient als blauwdruk voor conversies van eenheden:

```
using System;

public class UnitConverter
{
  int ratio;                              // Veld

  public UnitConverter (int unitRatio)    // Constructor
  {
    ratio = unitRatio;
  }

  public int Convert (int unit)           // Methode
  {
    return unit * ratio;
  }
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInches = new UnitConverter(12);
    UnitConverter milesToFeet = new UnitConverter(5280);

    Console.Write (feetToInches.Convert(30));   // 360
```

```
    Console.Write (feetToInches.Convert(100));  // 1200
    Console.Write (feetToInches.Convert
                    (milesToFeet.Convert(1)));  // 63360
  }
}
```

Members van een type

Een type bevat *datamembers* en *functiemembers*. Het datamember van `UnitConverter` is het *veld* met de naam `ratio`. De functiemembers van `UnitConverter` zijn de methode `Convert` en de *constructor* van `UnitConverter`.

Symmetrie van voorgedefinieerde typen en custom typen

Een mooi aspect van C# is dat voorgedefinieerde typen en custom typen weinig van elkaar verschillen. Het voorgedefinieerde type `int` dient als blauwdruk voor integers. Het biedt plaats voor 32-bitsgegevens en zorgt voor functiemembers die die gegevens gebruiken, zoals `ToString`. Op dezelfde manier fungeert ons custom type `UnitConverter` als blauwdruk voor eenhedenconversies. Het biedt plaats voor gegevens – `ratio` – en zorgt voor functiemembers die die gegevens verwerken.

Constructors en instantiatie

Gegevens worden gecreëerd door een type te *instantiëren*. We kunnen voorgedefinieerde typen instantiëren met behulp van een literal zoals `12` of `"Hello world"`.

De operator `new` creëert instanties van een custom type. We zijn onze methode `Main` gestart door twee instanties van het type `UnitConverter` aan te maken. Onmiddellijk nadat de operator `new` een object heeft geïnstantieerd, wordt de *constructor* van het object aangeroepen om initialisatie uit te voeren. Een constructor wordt net zo gedefinieerd als een methode, behalve dat de methodenaam en het retourtype worden gereduceerd tot de naam van het omhullende type:

```
public UnitConverter (int unitRatio)   // Constructor
{
  ratio = unitRatio;
}
```

Instantie- versus static members

De data- en functiemembers die op de instantie van het type inwerken, worden *instantie*members genoemd. De `Convert`-methode van `UnitConverter` en de `ToString`-methode van `int` zijn voorbeelden van instantiemembers. Standaard zijn members altijd instantiemembers.

Data- en functiemembers die niet op de instantie van het type inwerken, maar op het type zelf, moeten als `static` worden aangeduid. De methoden `Test.Main` en `Console.WriteLine` zijn static methoden. De class `Console` is eigenlijk een *static class*, wat betekent dat *al* zijn members static zijn. U creëert eigenlijk nooit instanties van `Console`; één console wordt over de hele applicatie gedeeld.

Laten we de instantie eens tegen static members afzetten. In de volgende code heeft het instantieveld `Name` betrekking op een instantie van een bepaalde `Panda`, terwijl `Population` betrekking heeft op de verzameling van alle `Panda`-instanties:

```
public class Panda
{
  public string Name;           // Instantieveld
  public static int Population; // Static veld

  public Panda (string n)       // Constructor
  {
    Name = n;                   // Wijs n toe aan instantieveld
    Population = Population+1;  // Hoog het static veld op met 1
  }
}
```

De volgende code creëert twee instanties van `Panda`, schrijft hun namen naar het scherm en vervolgens de totale populatie:

```
Panda p1 = new Panda ("Pan Dee");
Panda p2 = new Panda ("Pan Dah");

Console.WriteLine (p1.Name);      // Pan Dee
Console.WriteLine (p2.Name);      // Pan Dah

Console.WriteLine (Panda.Population);   // 2
```

Het keyword public

Het keyword `public` biedt members aan aan andere classes. Als het veld `Name` in `Panda` niet als `public` was aangeduid, zou het `private` zijn en zou de class `Test` er geen toegang toe hebben. Door een member public te maken kan een type met de buitenwacht communiceren: “Dit is wat ik andere typen wil laten zien – al het andere zijn private implementatiedetails.” In objectgeoriënteerde termen zeggen we dat de public members de private members van de class inkapselen (*encapsulation*).