Hoofdstuk 12

Klassen

12.1 Klassen

Een klasse is een groepje methoden. Dat hebben we in de programma's tot nu toe wel gezien: we definieerden steeds een of meerdere klassen (in ieder geval een subklasse van `Form`) met daarin methoden zoals constructormethoden, event-handlers, en wat we verder maar handig vonden.
Een klasse heeft ook nog een andere rol: het is het type van een object. Dat aspect is tussen alle library-feitenkennis een beetje onderbelicht gebleven. In deze sectie bekijken we daarom een hele eenvoudige klasse, waarin dit duidelijker wordt.
De voorbeeld-klasse heet `Kleur`. Deze klasse is dus het type van `Kleur`-objecten. Met zo'n object kun je een kleur beschrijven. We doen hier dus nog eens over wat in de library ook al bestaat: hiervoor is er immers al `Color`. Onze klasse `Kleur` is een mogelijk alternatief hiervoor, oftewel: het geeft een kijkje in de keuken van de library-schrijver hoe je een klasse zoals `Color` zelf had kunnen maken.
blz. 213 De programmatekst staat in listing 68. Een voorbeeld van hoe deze klasse in een programma blz. 214 gebruikt zou kunnen worden staat in listing 69.

Klasse: (ook) type van een object
Een klasse is dus, behalve een groepje methoden, ook het type van een object. Dus als er een klasse `Kleur` is, kunnen we variabelen declareren zoals:

```
Kleur oranje, paars;
```

De variabelen bevatten verwijzingen naar een object. Zolang we de variabelen nog geen waarde hebben gegeven, hebben ze nog de waarde `null`. Ze gaan daadwerkelijk naar een object wijzen na toekenningsopdrachten, waarin de constructormethode van `Kleur` wordt aangeroepen:

```
oranje = new Kleur();
paars = new Kleur();
```

Object: groepje variabelen
Een object is een groepje variabelen dat bij elkaar hoort. Maar welke variabelen zitten er nu precies in een `Kleur`-object? Dat wordt bepaald in de definitie van de klasse. De opbouw van een object wordt beschreven door de klasse die zijn type is.

Variabele-declaraties in een klasse
Behalve methodes kunnen er ook declaraties van variabelen in een klasse staan. Dat zijn de 'declaraties boven in de klasse' die we al zo vaak hebben gebruikt. In een eenvoudig geval zou een klasse *alleen maar* variabele-declaraties kunnen bevatten:

```
class Kleur
{
    public byte Rood;
    public byte Groen;
    public byte Blauw;
}
```

Met deze declaraties wordt de opbouw van objecten van het type `Kleur` beschreven: elk `Kleur`-object bestaat uit drie getallen, met de naam `Rood`, `Groen` en `Blauw`. De getallen hoeven niet zo groot te worden, dus daarom gebruiken we `byte` in plaats van `int`.

```
namespace KleurKlasse
{
    public class Kleur
    {
        public byte Rood, Groen, Blauw;
        public static byte Maximaal = 255;

        public Kleur()
        {
            Rood = Maximaal; Groen = Maximaal; Blauw = Maximaal;
        }
        public Kleur(byte x)
        {
            Rood = x; Groen = x; Blauw = x;
        }
        public Kleur(byte r, byte g, byte b)
        {
            Rood = r; Groen = g; Blauw = b;
        }
        public Kleur(Kleur orig)
        {
            Rood = orig.Rood; Groen = orig.Groen; Blauw = orig.Blauw;
        }
        public Kleur(string s)
        {
            string[] velden = s.Split(' ');
            Rood = byte.Parse(velden[0]);
            Groen = byte.Parse(velden[1]);
            Blauw = byte.Parse(velden[2]);
        }
        public override string ToString()
        {
            return $"{Rood} {Groen} {Blauw}";
        }
        public byte Grijswaarde()
        {
            return (byte)(0.3 * Rood + 0.6 * Groen + 0.1 * Blauw);
        }
        public void MaakDonkerder()
        {   Rood = (byte)(Rood * 0.9);
            Groen = (byte)(Groen * 0.9);
            Blauw = (byte)(Blauw * 0.9);
        }
        public Kleur DonkerdereVersie()
        {   Kleur res = new Kleur(this);
            res.MaakDonkerder();
            return res;
        }
        public static Kleur Zwart = new Kleur(0, 0, 0);
        public static Kleur Geel = new Kleur(Maximaal, Maximaal, 0);

        public static Kleur Parse(string s)
        {
            return new Kleur(s);
        }
    }
}
```

Listing 68: KleurKlasse/Kleur.cs

```
using System;

namespace KleurKlasse
{
    class Voorbeeld
    {
        static void Main()
        {
            Kleur wit, paars, oranje, lichtgrijs, donkergrijs;

            wit = new Kleur();
            paars = new Kleur(255, 0, 255);
            oranje = new Kleur(255, 128, 0);
            lichtgrijs = new Kleur(180);
            donkergrijs = new Kleur(60);

            byte x = oranje.Grijswaarde();
            Kleur oranjeInZwartwit = new Kleur(x);

            oranje.MaakDonkerder();
            string s = oranje.ToString();

            Kleur donkerPaars = paars.DonkerdereVersie();
            Kleur donkerGeel = Kleur.Geel.DonkerdereVersie();

            Console.WriteLine($"DonkerOranje: {oranje}");
            Console.WriteLine($"DonkerPaars: {donkerPaars}");
            Console.WriteLine($"DonkerGeel: {donkerGeel}");
            Console.ReadLine();
        }
    }
}
```

Listing 69: KleurKlasse/Voorbeeld.cs

Omdat de variabelen `public` zijn, kunnen ze ook vanuit andere klassen gebruikt worden. In de klasse `Voorbeeld` kunnen we dus bijvoorbeeld schrijven:

```
Kleur oranje;
oranje = new Kleur();
oranje.Rood  = 255;
oranje.Groen = 128;
oranje.Blauw = 0;
```

De constructormethode
In de klasse kunnen we een constructormethode definiëren. Dat is een methode met dezelfde naam als de klasse. Het doel van de constructormethode is om de variabelen van het object een zinvolle beginwaarde te geven, bijvoorbeeld:

```
public Kleur()
{
    Rood = 255; Groen = 255; Blauw = 255;
}
```

De constructormethode wordt automatisch aangeroepen zodra we met `new Kleur()` een nieuw object aanmaken. Het nieuwe object wordt meteen onder handen genomen door de constructormethode. Met de constructormethode uit het voorbeeld is elk nieuw gemaakt object dus de kleur wit.
Als er geen constructormethode is gedefinieerd, worden alle variabelen met de waarde 0 (voor getallen) of `null` (voor objectverwijzingen) gevuld. In dat geval zou elk nieuw kleur-object dus juist de kleur zwart beschrijven.

Constructormethoden met parameters
Er mogen meerdere constructormethoden zijn, die zich onderscheiden door het aantal en het type van de parameters. Vaak maken programmeurs een constructormethode met precies zveel parameters als er variabelen in de klasse zijn, zodat we die elk afzonderlijk een waarde kunnen geven. In onze `Kleur`-klasse zou dat zijn:

```
public Kleur(byte r, byte g, byte b)
{
    Rood = r; Groen = g; Blauw = b;
}
```

Maar er zijn ook tussenvormen mogelijk, bijvoorbeeld met één parameter, die dan als waarde voor alledrie de variabelen wordt gebruikt:

```
public Kleur(byte x)
{
    Rood = x; Groen = x; Blauw = x;
}
```

Een voorbeeld van gebruik van deze constructoren is:

```
Kleur wit, paars, lichtgrijs, donkergrijs;
wit = new Kleur();
paars = new Kleur(255, 0, 255);
lichtgrijs = new Kleur(180);
donkergrijs = new Kleur(60);
```

Andere methoden in de klasse
Er kunnen natuurlijk ook nog 'gewone' methoden in de klasse staan. Methoden in de klasse `Kleur` nemen een `Kleur`-object onder handen. Dat wil zeggen: ze mogen de waarden van `Rood`, `Groen` en `Blauw` gebruiken.
Een methode zou aan de hand daarvan een resultaatwaarde kunnen opleveren:

```
public byte Grijswaarde()
{
    return (byte)(0.3 * Rood + 0.6 * Groen + 0.1 * Blauw);
}
```

Deze methode kunnen we aanroepen met een van onze kleuren onder handen. Er wordt een

resultaatwaarde teruggegeven, dus de aanroep heeft de status van een expressie, die we hier aan de rechterkant van een toekenningsopdracht gebruiken:

```
byte x = oranje.Grijswaarde();
Kleur oranjeInZwartwit = new Kleur(x);
```

Sommige methoden hebben geen resultaatwaarde. Er staat dan `void` in de header. Over het algemeen zullen dit soort methoden het object veranderen. Dit is een voorbeeld:

```
public void MaakDonkerder()
{
    Rood = (byte)(Rood * 0.9);
    Groen = (byte)(Groen * 0.9);
    Blauw = (byte)(Blauw * 0.9);
}
```

De aanroep van een void-methode heeft de status van een opdracht. Een voorbeeld van zo'n aanroep is:

```
oranje.MaakDonkerder();
```

Deze neemt het object `oranje` onder handen, en laat het gewijzigd achter.

De methode ToString

Het is gebruikelijk om in een klasse ook een methode te schrijven die een `string` maakt, waarmee het object tekstueel zichtbaar gemaakt kan worden. Dit is vooral ook handig bij het debuggen van programma's. We doen dat in onze klasse dus ook:

```
public override string ToString()
{   return $"{Rood} {Groen} {Blauw}";
}
```

Maar waarom staat er `override` in de header van deze methode? De klasse `Kleur` is toch geen subklasse van een andere klasse, waarvan de oorspronkelijke methode `ToString` een nieuwe invulling kan krijgen?
Toch wel, want klassen die in hun header niet tot subklasse van een andere klasse worden gemaakt, zijn automatisch een subklasse van de klasse `object`. Dat is de oer-superklasse van alle klassen. Daarom heet hij ook `object`, want het enige wat alle klassen gemeenschappelijk hebben, is dat ze het type zijn van een object.
In de klasse `object` zit een `virtual` methode `ToString`, die dus bedoeld is om te overriden in een subklasse. En dat is wat we hier doen.
Het voordeel hiervan is, dat deze methode automatisch wordt aangeroepen als een object in een *interpolated string* (met zo'n dollar-teken) wordt gebruikt, bijvoorbeeld:

```
string s = $"de donkere versie van oranje is: {oranje}";
```

Een string terug-converteren naar een object

Soms is het handig om zo'n string (die misschien door een gebruiker nog is aangepast) weer terug te converteren naar een object. Dat is wat lastiger, want dan moet je zo'n string weer uit elkaar peuteren. Met behulp van de methode `Split` is dat ook weer niet erg lastig. We doen dit in nog een extra constructormethode:

```
public Kleur(string s)
{   string[] velden = s.Split(' ');
    Rood = byte.Parse(velden[0]);
    Groen = byte.Parse(velden[1]);
    Blauw = byte.Parse(velden[2]);
}
```

Static declaraties

De variabele-declaraties in de klasse bepalen hoe elk object van de klasse is opgebouwd. Hierop is één uitzondering: variabelen die `static` zijn gedeclareerd, zitten *niet* in elk object. Ze zitten alleen maar in de klasse omdat ze er zijdelings iets mee te maken hebben. In onze klasse hebben we zo'n variabele:

```
public static byte Maximaal = 255;
```

Static methoden

Ook methoden kunnen `static` zijn. Deze methoden hebben *geen* object onder handen. Ze zitten alleen maar in de klasse omdat ze er zijdelings iets mee te maken hebben.
Een klassieker in dit genre is een methode `Parse`, die in veel klassen aanwezig is. Deze methode heeft een string als parameter, en levert een nieuw object op van deze klasse. We kunnen hem gemakkelijk schrijven met behulp van de constructormethode-met-string-parameter die we al maakten:

```
public static Kleur Parse(string s)
{
    return new Kleur(s);
}
```

Static methoden hebben geen object onder handen. Bij de aanroep schrijf je dus ook geen object voor de punt. In plaats daarvan staat er de naam van de klasse. Dus een aanroep zou er zo uit kunnen zien:

```
Kleur test;
test = Kleur.Parse( "123 45 6" );
```

Dit geldt ook voor alle static methoden uit de library. Bijvoorbeeld alle varianten van `Parse` (zoals `byte.Parse` die we zonet nog gebruikt hebben). Maar ook alle methoden uit de klasse `Math`, zoals `Math.Sqrt`.

Static variabelen met de eigen klasse als type

Static variabelen mogen van elk willekeurig type zijn: int, string, enzovoorts. Maar dus ook van het type van de klasse zelf. We kunnen dus in de klasse `Kleur` static variabelen neerzetten die zelf ook van het type kleur zijn. Dit is handig om alvast een paar standaardkleuren beschikbaar te maken.
We schrijven dus in de klasse `Kleur` onder andere:

```
public static Kleur Zwart = new Kleur(0, 0, 0);
public static Kleur Geel = new Kleur(Maximaal, Maximaal, 0);
```

Deze variabelen zijn, omdat ze static zijn, daarna beschikbaar als `Kleur.Zwart` en `Kleur.Geel`.
Precies deze truc wordt ook gebruikt in de echte klasse `Color`. Daarom mogen we dingen schrijven als `Color.LightGreen` en `Color.AntiqueWhite`.