Inhoud

Inleiding

De evolutie en verandering van C# blijft doorgaan. En daarmee verandert ook de C#-community. Voor meer ontwikkelaars is C# de eerste professionele programmeertaal. De leden van onze community worden niet gehinderd door de vooropgezette ideeën die gebruikelijk zijn onder degenen die C# zijn gaan gebruiken na jaren ervaring met andere op C gebaseerde talen. Zelfs de ontwikkelaars die C# al jaren gebruiken voelen de noodzaak om zich nieuwe werkwijzen eigen te maken als gevolg van de snelle opeenvolging van veranderingen. Sinds de compiler open source is, volgen de innovaties in de C#-taal elkaar snel op. De review van voorgestelde functionaliteit voor de C#-taal gebeurt nu door de hele community in plaats van een kleine groep taalexperts. De community kan nu ook deelnemen aan het ontwerpen van de nieuwe functionaliteit.

Wijzigingen in voorgestelde architecturen en toepassingen veranderen ook de taalidiomen die wij als C#-ontwikkelaars gebruiken. Het bouwen van toepassingen door het schrijven van microservices, distributed programma's en de scheiding van data en algoritmes maken allemaal deel uit van moderne softwareontwikkeling. De C#-taal heeft de eerste stappen gezet voor de opname van deze verschillende idiomen.

De organisatie van het *Handboek effectiever programmeren in C#* weerspiegelt zowel de wijzigingen in de taal als de wijzigingen in de C#-community. Dit boek neemt u niet mee op een historische reis door de veranderingen in de taal, maar biedt u advies over het gebruik van de huidige C#-taal. De verschillende adviezen behandelen de nieuwe taal- en structuureigenschappen en de praktijken die de community zich heeft eigengemaakt door het bouwen van verschillende versies van softwareproducten met C#. Deze vijftig adviezen vormen een verzameling aanbevelingen die u als professioneel ontwikkelaar kunt gebruiken om C# meer effectief in te zetten.

Dit boek gaat ervan uit dat u C# 7 gebruikt, maar het is geen uitputtende behandeling van de nieuwe taalfuncties. Dit boek biedt praktisch advies hoe u deze functies kunt gebruiken voor het oplossen van problemen waar u in de dagelijkse praktijk tegenaan kunt lopen. Het behandelt in het bijzonder die C# 7-functionaliteit waarmee u op een nieuwe en betere manier veelgebruikt idioom kunt schrijven. Een Internetzoekactie toont nog steeds de oplossingen die al jaren meegaan. Dit boek belicht deze oudere oplossingen en legt uit waarom de nieuwe functionaliteit in C# betere oplossingen mogelijk maakt.

Veel van de aanbevelingen in dit boek kunt u controleren met op Roslyn-gebaseerde analysers en codefixes. Ik onderhoud een repository van deze bronnen

op dit adres: **https://github.com/BillWagner/MoreEffectiveCSharpAnalyzers**. Wanneer u ideeën hebt of wilt bijdragen aan deze repository, schrijf dan een bijdrage of stuur me een verzoek.

Voor wie is dit boek bedoeld?

Het *Handboek effectiever programmeren in C#* is geschreven voor professionele ontwikkelaars voor wie C# de primaire programmeertaal is. Ik ga ervan uit dat u bekend bent met de syntaxis en eigenschappen van C# en dat u vaardigheid hebt in het schrijven van code in C#. Dit boek bevat geen uitleg over standaardinstructies en taaleigenschappen. In plaats daarvan wordt besproken hoe u de functionaliteit van de huidige versie van de C#-taal kunt integreren in uw dagelijkse ontwikkelwerk.

Daarnaast gaat dit boek ervan uit dat u enige kennis hebt van de *Common Language Runtime* (CLR) en de *just-in-time* (JIT) compiler.

Over de inhoud

Data is alomtegenwoordig in de huidige wereld. In een objectgeoriënteerde aanpak zijn data en code onderdeel van een type met de daarbij behorende bevoegdheden. Een functionele aanpak behandelt methodes als data. Een servicegerichte aanpak scheidt de data van de code die de data bewerkt. De evolutie van C# heeft ervoor gezorgd dat de taal functionaliteit bevat die gemeenschappelijk is voor al deze paradigma's, hetgeen complicaties kan opleveren bij uw ontwerpkeuzes. Hoofdstuk 1 bespreekt deze keuzes en biedt een leidraad bij het kiezen van een geschikt taalidioom voor verschillende toepassingen.

Programmeren is in wezen het ontwerp van de API. Het brengt uw verwachtingen over het gebruik van uw code over aan de gebruikers. Het spreekt ook boekdelen over uw inzicht in de behoeften en verwachtingen van andere ontwikkelaars. In hoofdstuk 2 leert u hoe u hiervoor het rijke palet van C#-taalfuncties het beste inzet. U ziet hoe u luie evaluatie gebruikt, configureerbare interfaces maakt en onduidelijkheid voorkomt tussen de verschillende taalelementen in uw algemene interfaces.

Taak-gebaseerd asynchroon programmeren levert nieuwe idiomen op voor het samenstellen van toepassingen met asynchrone bouwstenen. Het beheersen van deze functionaliteit betekent dat u API's kunt maken voor asynchrone taken die duidelijk weergeven wat de code uitvoert, en die makkelijk zijn te gebruiken. In hoofdstuk 3 leert u hoe u de taak-gebaseerde asynchrone taalfunctionaliteit gebruikt voor code die wordt uitgevoerd door meerdere services en die daarvoor diverse bronnen gebruikt.

Hoofdstuk 4 neemt een specifieke subset van asynchroon programmeren onder de loep: *multithreaded parallel execution*. U ziet hoe u met PLINQ de verwerking van complexe algoritmes eenvoudiger verdeelt over meerdere cores en meerdere CPU's.

Hoofdstuk 5 bespreekt het gebruik van C# als een dynamische taal. In C# speelt het type een grote rol, met name statische typen. Tegenwoordig bevat een toenemend aantal programma's zowel dynamische als statische typen. Met C# kunt u gebruikmaken van dynamische programmeertechnieken zonder de voordelen van statische typen binnen het hele programma te verliezen. In hoofdstuk 5 leert u hoe u dynamische functies gebruikt en hoe u voorkomt dat dynamische typen het hele programma binnensluipen.

Hoofdstuk 6 sluit dit boek af met suggesties over hoe u kunt deelnemen aan de wereldwijde C#-community. Er zijn veel manieren waarop u kunt participeren en waarmee u kunt helpen bij de aanpassing van de programmeertaal die u dagelijks gebruikt.

Codeconventies

Het weergeven van code in een boek betekent het sluiten van compromissen wat betreft ruimte en duidelijkheid. Ik heb geprobeerd om de voorbeelden te focussen op het specifieke punt dat de code illustreert. Vaak betekent dat het weglaten van andere gedeelten van een klasse of een methode. Soms betekent dit ook het achterwege laten van error recovery code om ruimte te besparen. Een public methode dient de parameters en andere invoer te valideren, maar ook die code wordt hier meestal weggelaten. Uit ruimteoverwegingen is ook de validatie voor het aanroepen van methodes en `try`/`finally`-constructies weggelaten, deze controles zijn vaak opgenomen in ingewikkelde algoritmen.

Ik veronderstel ook dat de meeste ontwikkelaars de toepasselijke namespace (*naamruimte*) kunnen vinden wanneer een voorbeeld een van de veelgebruikte namespaces gebruikt. U kunt ervan uitgaan dat elk voorbeeld impliciet de volgende `using`-statements bevat:

```
using System;
using static System.Console;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

Feedback geven

Ondanks mijn beste inspanningen – en van de mensen die de tekst hebben nagekeken – is het mogelijk dat er fouten zijn geslopen in de tekst of voorbeelden. Als u denkt dat u een fout hebt gevonden, neem dan contact met mij op via e-mail **bill@thebillwagner.com** of via Twitter **@billwagner**. Errata vindt u op **http://thebillwagner.com/Resources/MoreEffectiveCS**. Veel van de adviezen in dit boek zijn geïnspireerd door e-mail- en Twitter-gesprekken met andere C#-ontwikkelaars. Hebt u vragen of opmerkingen over de aanbevelingen in dit boek, neem dan contact op met mij. Discussies van algemeen belang vindt u terug op mijn blog op **http://thebillwagner.com/blog**.

Dankwoord

Er zijn veel mensen aan wie ik dank verschuldigd ben voor hun bijdrage aan dit boek. Ik heb het voorrecht om al de nodige jaren deel uit te maken van een verbazingwekkende C#-community. Iedereen op de *C# Insidersmailinglijst* (binnen of buiten Microsoft) heeft bijgedragen met ideeën en conversaties, waardoor dit een beter boek is geworden.

Ik moet een paar leden van de C#-community persoonlijk bedanken voor de hulp met ideeën, en met het omzetten van ideeën in concrete aanbevelingen. De gesprekken met Jon Skeet, Dustin Campbell, Kevin Pilch, Jared Parsons, Scott Allen en vooral Mads Torgersen zijn de basis voor veel nieuwe ideeën in deze uitgave.

Voor deze uitgave had ik de beschikking over een geweldig team van technische recensenten. Jason Bock, Mark Michaelis en Eric Lippert bestudeerden de tekst en voorbeelden en verbeterden zeer de kwaliteit van het boek dat u nu in handen hebt. Hun recensies waren grondig en compleet, en meer dan dat kan niemand vragen. Daarnaast kwamen ze met aanbevelingen die me hielpen om veel van de onderwerpen beter uit te leggen.

Het team van Addison-Wesley is een droom om mee te werken. Trina MacDonald is een fantastische redacteur, leermeester en de drijvende kracht die het werk gedaan krijgt. Ze steunt op Mark Renfrow en Olivia Basegio, net als ik. Hun bijdragen garandeerden dat het voltooide manuscript een kwalitatief hoogstandje was, van cover tot cover en alles daartussenin. Curt Johnson blijft op marketingtechnisch gebied fantastisch werk leveren. Het maakt niet uit welk medium u hebt gekozen, Curt heeft de hand gehad in het bestaan van dit boek.

Ik vind het een eer om deel uit te maken van Scott Meyers's serie boeken. Hij bekijkt elk manuscript en biedt suggesties en opmerkingen voor verbetering. Scott is ongelooflijk grondig en zijn ervaring in software, hoewel niet in C#,

betekent dat hij alle gebieden vindt waar ik iets niet duidelijk genoeg heb uitgelegd of waar een advies goed op zijn plaats zou zijn. Zijn feedback, zoals altijd, is van onschatbare waarde geweest in de voorbereiding van deze uitgave.

Zoals altijd heeft mijn familie mij de tijd gegeven zodat ik dit manuscript kon voltooien. Mijn vrouw Marlene wachtte geduldig talloze uren, terwijl ik wegging om te schrijven of voorbeelden te maken. Zonder haar steun zou ik dit boek nooit hebben afgekregen, en het zou ook niet zo bevredigend zijn geweest om deze projecten te voltooien.

Over de auteur

Bill Wagner is een van 's werelds belangrijkste C#-ontwikkelaars en hij is lid van de ECMA C# Standards Committee. Hij is voorzitter van de Humanitarian Toolbox, is benoemd tot Microsoft Regional Director en is elf jaar .NET MVP, en hij trad onlangs toe tot de .NET Foundation Advisory Council. Bill heeft samengewerkt met bedrijven, variërend van starters tot grote ondernemingen, aan het verbeteren van hun softwareontwikkelingsproces en het uitbreiden van hun softwareontwikkelingsteams. Hij werkt momenteel met Microsoft aan het .NET Core content team. Hij creëert leermiddelen voor ontwikkelaars die geïnteresseerd zijn in de C#-taal en .NET Core. Bill behaalde een BS in Computer Science aan de Universiteit van Illinois in Champaign-Urbana.

Hoofdstuk 1

Werken met gegevenstypen

C# werd oorspronkelijk ontworpen voor softwareontwikkeling met objectgeoriënteerde technieken, waarin de gegevens en de functionaliteit worden samengebracht. Na verloop van tijd zijn er nieuwe idiomen toegevoegd ter ondersteuning van steeds vaker voorkomende programmeertechnieken. Een van die trends is het scheiden van de gegevensopslag van de methoden die deze gegevens bewerken. Deze tendens wordt gedreven door de overgang naar gedistribueerde systemen, waarin een applicatie wordt verdeeld in kleinere processen, die elk één functie of enkele verwante functies uitvoeren. De introductie van een nieuwe strategie voor het scheiden van gegevens en methode resulteert natuurlijk ook in de ontwikkeling van nieuwe programmeertechnieken. En zo leiden nieuwe programmeertechnieken weer tot nieuwe taalfuncties.

In dit hoofdstuk leert u technieken om data te scheiden van de methoden die deze gegevens manipuleren of verwerken. Deze gegevens zijn niet altijd objecten; soms zijn het functies en passieve datacontainers.

Advies 1: Gebruik eigenschappen in plaats van toegankelijke data members

Eigenschappen (*properties*) zijn altijd een onderdeel geweest van de C#-taal, maar sinds de eerste release van C# zijn verschillende uitbreidingen geïntroduceerd waardoor de eigenschappen nog krachtiger zijn. U kunt bijvoorbeeld verschillende toegangsbeperkingen opgeven voor de getter en setter. Gebruikt u Auto-Property, dan hoeft u niet handmatig typen te declareren voor de eigenschappen (in plaats van data members) en kunt u automatisch access modifiers toekennen, zoals Read-Only. Expression-bodied data members bieden een nog beknoptere syntaxis. Als u nog steeds velden als public declareert bij uw datatypen, stop daar dan mee. Schrijft u nog steeds Get- en Set-methoden met de hand, houd er dan mee op. Met eigenschappen maakt u data members eenvoudig beschikbaar als onderdeel van de public interface, terwijl u toch de verwachte inkapseling behoudt van een objectgeoriënteerde omgeving. Eigenschappen zijn taalelementen die toegankelijk zijn alsof ze data members zijn, maar ze worden geïmplementeerd als methoden.

Sommige members van een type worden het best weergegeven als data: de naam van een klant, de (*x*, *y*)-coördinaten van een punt of de omzet van vorig jaar. Met eigenschappen kunt u een interface maken die werkt alsof deze rechtstreeks toegang heeft tot de datavelden, maar die nog steeds alle voordelen biedt van een methode. Clientcode benadert eigenschappen alsof deze public velden benadert. De feitelijke implementatie gebruikt echter methoden, waarin u het gedrag van de eigenschappen-accessors definieert.

Het .NET Framework gaat ervan uit dat u eigenschappen gebruikt voor uw public data members. Sterker nog, de klassen voor gegevensbinding in het .NET Framework ondersteunen eigenschappen, geen public data fields. Dit geldt voor alle bibliotheken voor gegevensbinding: WPF, Windows Forms en Web Forms. Gegevensbinding koppelt een eigenschap van een object aan een besturingselement van de gebruikersinterface. Hierbij wordt *reflection* gebruikt om de genoemde eigenschap te vinden in een type:

```
textBoxCity.DataBindings.Add("Text",
    address, nameof(City));
```

Deze code koppelt de eigenschap Text van het besturingselement textBoxCity aan de eigenschap City van het object address. Dit werkt niet met een public data field met de naam City; de ontwerpers van de Framework Class Library hebben hiervoor geen ondersteuning ingebouwd. Het gebruik van public data members wordt in object oriented programming (*OOP*) gezien als een slechte

werkwijze, zodat ondersteuning hiervoor niet is opgenomen in de Framework Class Library. Deze omissie is een reden te meer om de juiste OOP-technieken te volgen.

Gegevensbinding is alleen van toepassing op klassen met elementen die worden weergegeven door de code van uw gebruikersinterface (UI). Maar dat betekent niet dat u eigenschappen uitsluitend moet gebruiken in de UI-code: gebruik ook eigenschappen voor andere klassen en structuren. Eigenschappen zijn veel eenvoudiger te wijzigen wanneer u later de eisen of het gedrag wilt aanpassen. Stel dat u besluit dat het type Klant nooit een lege naam mag hebben. Als u de eigenschap voor het veld Naam als public hebt ingesteld, dan is deze nieuwe eis eenvoudig toe te voegen, omdat dit slechts op één plaats hoeft te gebeuren:

```
public class Customer
{
    private string name;
    public string Name
    {
        get => name;
        set
        {
            if (string.IsNullOrWhitespace(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    nameof(Name));
            name = value;
        }
        // More elided
    }
}
```

Had u hiervoor public data members gebruikt, dan had u de gehele code moeten doorlopen en aanpassen op iedere locatie waar de naam van de klant wordt toegewezen. Dat kost meer tijd, veel meer tijd.

Omdat eigenschappen worden geïmplementeerd met methoden, is het toevoegen van ondersteuning voor multithreading eenvoudiger. U breidt de implementatie van de get- en set-accessors uit met gesynchroniseerde toegang tot de data (zie advies 39 voor meer informatie):

```
public class Customer
{
    private object syncHandle = new object();
```

```
    private string name;
    public string Name
    {
        get
        {
            lock (syncHandle)
                return name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    nameof(Name));
            lock (syncHandle)
                name = value;
        }
    }
    // More elided
}
```

Eigenschappen hebben dezelfde taalfuncties als methoden. En eigenschappen kunnen ook virtueel zijn:

```
public class Customer
{
    public virtual string Name
    {
        get;
        set;
    }
}
```

Merk op dat de laatste paar voorbeelden de impliciete eigenschapsyntaxis gebruiken. Het is gebruikelijk om met een eigenschap een backing store op te zetten. Meestal hoeft u geen code voor validering toe te voegen aan de accessors van de eigenschap. De C#-taal ondersteunt de vereenvoudigde impliciete syntaxis van eigenschappen zodat veel van de ceremoniële code achterwege kan blijven waarmee een eenvoudig veld zichtbaar is als eigenschap. De compiler creëert een private veld (meestal een *backing store* genoemd) en implementeert de benodigde logica voor zowel de get- als de set-accessor.

U kunt eigenschappen ook als abstract definiëren en eigenschappen definiëren als onderdeel van een interfacedefinitie. Hiervoor gebruikt u vergelijkbare syn-

taxis als bij impliciete eigenschappen. In het volgende voorbeeld ziet u een eigenschapsdefinitie in een generieke interface. Hoewel de syntaxis consistent is met impliciete eigenschappen, bevat deze interfacedefinitie geen enkele implementatie. Het definieert een contract waaraan elk type moet voldoen dat deze interface implementeert.

```
public interface INameValuePair<T>
{
    string Name { get; }

    T Value { get; set; }
}
```

Eigenschappen zijn volwaardige, eersteklas taalelementen die een uitbreiding zijn van methoden die interne gegevens benaderen of wijzigen. Alles wat u kunt doen met member-functies, kunt u doen met eigenschappen. Met het gebruik van eigenschappen vermijdt u ook een andere belangrijke bron van fouten die met velden mogelijk is: u kunt geen eigenschap aan een methode doorgeven met het keyword `ref` of `out`.

De accessors voor een eigenschap zijn twee verschillende methoden die binnen het type zijn gecompileerd. U kunt in C# bij een eigenschap verschillende access modifiers opgeven voor de get- en set-accessors. Deze flexibiliteit geeft u nog meer controle over de zichtbaarheid van de gegevenselementen die u met eigenschappen instelt:

```
public class Customer
{
    public virtual string Name
    {
        get;
        protected set;
    }
    // Remaining implementation omitted
}
```

De syntaxis van de eigenschap reikt verder dan eenvoudige gegevensvelden. Als uw type geïndexeerde items bevat als onderdeel van de interface, dan kunt u indexeerders gebruiken. Indexeerders zijn hier geparametriseerde eigenschappen. Deze werkwijze is een handige manier om een eigenschap te maken die de items in een reeks retourneert:

```
public int this[int index]
{
```

```
    get => theValues[index];
    set => theValues[index] = value;
}
     private int[] theValues = new int[100];

// Accessing an indexer:
int val = someObject[i];
```

Indexeerders hebben dezelfde taalondersteuning als de eigenschappen van een enkel item: ze worden geïmplementeerd als methoden die u schrijft, zodat u elke verificatie of berekening binnen de indexeerfunctie kunt toepassen. Een indexeerder kan virtueel of abstract zijn, u kunt ze declareren in interfaces, en ze kunnen read-only of read-write zijn. Eendimensionale indexeerders met numerieke parameters zijn toegankelijk voor gegevensbinding. Andere indexeerders met niet-integere parameters kunt u gebruiken om kaarten te definiëren:

```
public Address this[string name]
{
    get => addressValues[name];
    set => addressValues[name] = value;
}
private Dictionary<string, Address> addressValues;
```

In overeenstemming met de multidimensionale arrays in C#, kunt u multidimensionale indexeerders maken met overeenkomstige of verschillende typen op elke as:

```
public int this[int x, int y]
    => ComputeValue(x, y);

public int this[int x, string name]
    => ComputeValue(x, name);
```

Merk op dat alle indexeerders zijn gedeclareerd met het keyword `this`. U kunt een indexeerder in C# geen naam geven, dus heeft elke aparte indexeerder in een type een duidelijke parameterlijst nodig om onduidelijkheid te voorkomen. Bijna alle mogelijkheden van eigenschappen zijn ook beschikbaar voor indexeerders: indexeerders kunnen virtueel of abstract zijn; zij kunnen afzonderlijke access modifiers hebben voor setters en getters. Er is echter één verschil: u kunt geen impliciete indexeerders maken, iets wat wel kan met eigenschappen.

Deze functionaliteit van eigenschappen is prima en het is een fijne verbetering ten opzichte van eerdere versies van C#. Maar wellicht bent u nog steeds geneigd om een eerste implementatie met data members te maken en – zodra u de voordelen daarvan nodig hebt – de data members te vervangen door eigenschappen. Dat klinkt als een redelijke strategie, maar het is verkeerd. Bekijk het volgende gedeelte van een klassendefinitie:

```
// Using public data members, bad practice:
public class Customer
{
    public string Name;

    // Remaining implementation omitted
}
```

Deze klassendefinitie beschrijft een klant met een naam. U kunt de naam opvragen of instellen met de gebruikelijke notatie:

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

Dat is eenvoudig en rechttoe rechtaan. U kunt zich voorstellen dat u later het data member Name kunt kunt vervangen door een eigenschap, en dat de code ongewijzigd zou blijven werken. Dat is min of meer waar. Eigenschappen zijn bedoeld om op data members te lijken als ze worden benaderd, dat is immers het doel van de syntaxis. Maar eigenschappen zijn *geen* gegevens; de toegang tot een eigenschap genereert andere MSIL-instructies (*Microsoft Intermediate Language*) dan de toegang tot gegevens.

Hoewel eigenschappen en data members compatibel zijn in de broncode, zijn ze niet binair compatibel. In het voor de hand liggende geval betekent deze beperking dat wanneer u van een public data member overschakelt naar de equivalente public eigenschap, u alle code die gebruikmaakt van het public data member, opnieuw moet compileren. C# behandelt binaire samenstellingen als eersteklasburgers. Eén doel van de taal is dat het mogelijk is om één enkele bijgewerkte assemblage vrij te geven zonder de volledige toepassing te updaten. Het veranderen van een data member in een eigenschap verbreekt de binaire compatibiliteit, en daarmee wordt het upgraden van een enkele assemblage veel moeilijker.

Als u de MSIL-instructies voor een eigenschap bekijkt, zou u zich kunnen afvragen wat de relatieve performance is van eigenschappen en data members. De prestaties met eigenschappen zullen niet sneller zijn dan met data members, maar waarschijnlijk ook niet langzamer. De just-in-time (JIT) compiler verwerkt

inline bepaalde methodeaanroepen, inclusief die van property accessors. Als de JIT-compiler inline de property accessors afhandelt, dan zijn de prestaties van data members en eigenschappen hetzelfde. Zelfs wanneer een property accessor niet inline wordt afgehandeld, dan is het verschil in prestaties de verwaarloosbare tijd van een functieaanroep. Dit verschil is alleen in een klein aantal situaties meetbaar.

Eigenschappen zijn methoden die zichtbaar zijn vanuit de aanroepende code zoals gegevens, iets wat waarschijnlijk bij uw gebruikers een aantal verwachtingen wekt. Zij zien de toegang tot een eigenschap alsof het de toegang tot data is. Immers, zo ziet het eruit. Uw property accessors moeten deze verwachtingen waar maken. Get-accessors zouden geen waarneembare bijwerkingen moeten hebben. Daarentegen wijzigen set-accessors de status, en gebruikers zouden deze veranderingen moeten kunnen zien.

Property accessors wekken dus ook prestatieverwachtingen bij uw gebruikers. Een property access ziet eruit als een data field access. Er zouden geen significant andere prestatiekarakteristieken mogen zijn ten opzichte van een eenvoudige data access. Property accessors dienen geen langdurige berekeningen uit te voeren of aanroepen tussen applicaties te maken (zoals het uitvoeren van databasequery's) of andere langdurige bewerkingen uitvoeren die niet consistent zijn met wat uw gebruikers verwachten van een property accessor.

Gebruik eigenschappen wanneer u gegevens beschikbaar maakt in de public of protected interfaces van een type. Gebruik een indexeerder voor reeksen of dictionary's. Declareer alle data members zonder uitzondering als private. Deze instellingen bieden onmiddellijk ondersteuning voor gegevensbinding, waardoor u in de toekomst veel eenvoudiger de implementatie van de methoden kunt aanpassen. Het extra werk dat nodig is om een variabele in een eigenschap in te kapselen, bedraagt één of twee minuten van uw dag. Ter vergelijking, wilt u achteraf eigenschappen gebruiken en ervoor zorgen dat uw programma's correct blijven werken, dan brengt dat vele uren werk met zich mee. Wanneer u nu een beetje tijd investeert, dan bespaart u zichzelf later veel tijd.

Advies 2: Kies impliciete eigenschappen boven variabele data

Uitbreidingen aan de syntaxis van eigenschappen in C# laten u uw ontwerpdoel met eigenschappen duidelijk maken. De moderne C#-syntaxis maakt het mogelijk dat u achteraf ook uw ontwerp kunt wijzigen. Begin met eigenschappen en maak veel toekomstige scenario's mogelijk.

Wanneer u benaderbare gegevens aan een klasse toevoegt, zijn de eigenschapsaccessors vaak simpele omslagen (*wrappers*) voor uw gegevensvelden. Als dat het geval is, dan verhoogt u de leesbaarheid van uw code door impliciete eigenschappen te gebruiken:

```
public string Name { get; set; }
```

De compiler maakt het backing field met een door de compiler gegenereerde naam. Met de setter van de eigenschap kunt u de waarde van het backing field wijzigen. Omdat de compiler de naam van het backing field genereert, moet u ook binnen de eigen klasse de eigenschapsaccessor aanroepen om het backing field te wijzigen. Dat is geen probleem: het aanroepen van de eigenschapsaccessor doet het werk, en omdat de gegenereerde eigenschapsaccessor een enkel toewijzingsstatement is, wordt het waarschijnlijk inline uitgevoerd. Het runtimegedrag van de impliciete eigenschap is hetzelfde als het runtimegedrag bij het benaderen van het backing field, ook wat betreft performance.

Impliciete eigenschappen ondersteunen dezelfde access modifiers als hun expliciete tegenhangers. U kunt zelf de gewenste restricties toevoegen aan de set accessor:

```
public string Name
{
    get;
    protected set;
}
// Or
public string Name
{
    get;
    internal set;
}
// Or
public string Name
{
    get;
    protected internal set;
}
// Or
public string Name
{
    get;
    private set;
}
```

```
// Or
// Can be set only in the constructor:
public string Name { get; }
```

Impliciete eigenschappen genereren dezelfde code voor een eigenschap met backing field als die u zelf in een eerdere versie van C# handmatig zou hebben geschreven. Gebruikt u impliciete eigenschappen, dan wordt u productiever en uw klassen zijn beter leesbaar. De declaratie van een impliciete eigenschap toont aan iedereen die uw code leest precies wat u wilt maken. De extra informatie kan achterwege blijven, zodat het bestand overzichtelijk blijft.

Aangezien impliciete eigenschappen dezelfde code genereren als expliciete eigenschappen, kunt u impliciete eigenschappen gebruiken om virtuele eigenschappen te definiëren, virtuele eigenschappen te overschrijven (*override*) of een eigenschap te implementeren die is gedefinieerd in een interface.

Wanneer u impliciet een virtuele eigenschap maakt, hebben afgeleide klassen geen toegang tot de door de compiler gegenereerde backing store. Echter, overschrijvingen (*overrides*) hebben wel toegang tot de accessmethodes `get` en `set` van de basiseigenschappen, net als met elke andere virtuele methode:

```
public class BaseType
{
    public virtual string Name
    {
        get;
        protected set;
    }
}

public class DerivedType : BaseType
{
    public override string Name
    {
        get => base.Name;
        protected set
        {
            if (!string.IsNullOrEmpty(value))
                base.Name = value;
        }
    }
}
```

U hebt twee extra voordelen wanneer u impliciete eigenschappen gebruikt. Ten eerste: wanneer u later de impliciete eigenschap wilt vervangen door een concrete implementatie voor gegevensvalidatie of andere acties, dan maakt u binair-compatibele wijzigingen binnen de klasse. Ten tweede: de validatie komt slechts op één locatie terecht.

In eerdere versies van C# benaderden de meeste ontwikkelaars het backing field rechtstreeks om het binnen de eigen klasse te wijzigen. Die werkwijze produceert code die de validatie en foutcontrole verspreidt door het gehele bestand. Elke wijziging in het backing field van een impliciete eigenschap roept de (eventueel private) eigenschapsaccessor aan. Verander daarom de impliciete eigenschapsaccessor in een expliciete eigenschapsaccessor en schrijf dan alle code voor de validatie in de nieuwe accessor:

```
// Original version
public class Person
{
    public string FirstName { get; set;}
    public string LastName { get; set; }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}

// Later updated for validation
public class Person
{
    public Person(string firstName, string lastName)
    {
        // Leverage validation in property setters:
        this.FirstName = firstName;
        this.LastName = lastName;
    }
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "First name cannot be null or empty");
            firstName = value;
        }
    }
```

```
    private string lastName;
    public string LastName
    {
        get => lastName;
        private set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Last name cannot be null or empty");
            lastName = value;
        }
    }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}
```

Wanneer u impliciete eigenschappen gebruikt, dan komt alle validatie op één plaats terecht. Als u de accessor kunt blijven gebruiken in plaats van het backing field rechtstreeks te benaderen, dan blijft alle veldvalidatie op één locatie.

Impliciete eigenschappen hebben één belangrijke beperking: u ze niet gebruiken op typen met het attribuut `Serializable`. De bestandsindeling voor persistente opslag is afhankelijk van de veldnaam van de door de compiler gegenereerde backing store. Er is geen garantie dat die veldnaam hetzelfde blijft, dat wil zeggen, die naam kan veranderen telkens wanneer u de klasse aanpast.

Ondanks hun beperkingen besparen impliciete eigenschappen ontwikkelingstijd, ze produceren leesbare code, en ze bevorderen een stijl van ontwikkeling waarin alle validatie voor veldwijzigingen op één locatie gebeurt. Als uw code overzichtelijker is, dan is het onderhoud van die code ook eenvoudiger.

Advies 3: Kies voor constante value-typen

Constanten (*immutable types*) zijn eenvoudig te begrijpen: nadat ze zijn gemaakt, houden ze een constante waarde. Als u de parameters valideert waarmee u het object construeert, dan kunt u er zeker van zijn dat het object vanaf dat punt een geldige status heeft en houdt; u kunt immers de interne status van het object niet meer wijzigen. Daarmee kan ook de foutcontrole achterwege blijven omdat er geen statuswijzigingen meer mogelijk zijn nadat het object is gebouwd. De constante typen zijn inherent veilig te gebruiken in threads: meerdere lezers kunnen dezelfde inhoud benaderen. Als de interne status niet kan veranderen, is het onmogelijk dat verschillende threads incon-

sistente weergaven van de gegevens zien. Constante typen kunt u veilig exporteren vanuit uw objecten, omdat de aanroeper de interne status van uw objecten niet kan wijzigen.

Constante typen werken beter in verzamelingen met hashcode. De waarde die wordt geretourneerd door `Object.GetHashCode()` moet een constante instantie zijn (zie advies 10); dat is altijd het geval voor constante typen.

In de praktijk is het heel moeilijk om elk type constant te maken. Daarom is dit advies van toepassing op zowel atomaire als constante value-typen. Ontbindt uw typen in structuren die op natuurlijke wijze een enkele entiteit vormen. Het type Adres vormt een zo'n entiteit. Een adres is één item dat bestaat uit verschillende gerelateerde velden, en een wijziging in één veld betekent waarschijnlijk ook wijzigingen in andere velden. Het type Klant is daarentegen geen atomair type. Een klanttype bevat waarschijnlijk veel gegevens, zoals een adres, een naam en een of meer telefoonnummers, en elk van deze onafhankelijke gegevens kan worden gewijzigd. Zo kan een klant van telefoonnummer veranderen zonder te verhuizen. Als een klant verhuist, kan deze wel hetzelfde telefoonnummer behouden. En een klant kan zijn of haar naam wijzigen met behoud van telefoonnummer en zonder van adres te veranderen. Een klantobject is niet atomair; het is samengesteld uit veel verschillende constante typen – een adres, een naam, of een verzameling van telefoonnummer en meer. Atomaire typen zijn enkele entiteiten: bij een wijziging is het logisch om de gehele inhoud te vervangen. De uitzondering hierop is wanneer u een van de samenstellende velden wijzigt.

Hier is een typische implementatie van een variabel (*mutable*) adres:

```
// Mutable Address structure
public struct Address
{
    private string state;
    private int zipCode;

    // Rely on the default system-generated
    // constructor

    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string State
    {
        get => state;
        set
```

```
        {
            ValidateState(value);
            state = value;
        }
    }

    public int ZipCode
    {
        get => zipCode;
        set
        {
            ValidateZip(value);
            zipCode = value;
        }
    }

    // Other details omitted
}

// Example usage:
Address a1 = new Address();
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111;
// Modify:
a1.City = "Ann Arbor"; // Zip, State invalid now
a1.ZipCode = 48103; // State still invalid now
a1.State = "MI"; // Now fine
```

Interne statusveranderingen betekenen dat het mogelijk is om de onveranderlijkheid van het object te schenden, althans tijdelijk. Nadat u het veld City hebt gewijzigd, krijgt het object a1 een ongeldige status. De plaats is veranderd en komt niet meer overeen met de inhoud van de velden State en ZipCode. De code ziet er onschuldig genoeg uit, maar stel dat dit fragment onderdeel is van een multithreaded programma: elke wijziging in de context na de wijziging in City en voor de aanpassing van State of ZipCode kan leiden tot een inconsistente weergave van de gegevens in een andere thread.

Zelfs als u geen multithreaded programma schrijft, dan kunt u toch in de problemen komen bij verandering van de interne status. Stel dat de inhoud van het veld ZipCode ongeldig is, zodat set een exception genereert. U hebt nog niet alle wijzigingen aangebracht, maar nu heeft het systeem een ongeldige status. Om dit probleem te verhelpen, moet u aanzienlijke hoeveelheid interne

valideringscode toevoegen aan de adresstructuur. Die valideringscode verhoogt op zijn beurt de omvang en de complexiteit van uw totale code. Als u volledige beveiliging voor exceptions wilt implementeren, dan zou u beveiligingscode moeten schrijven voor elk codeblok waarin u meer dan één veld verandert. Threadsafety vereist het toevoegen van significante threadsynchronisatiecontroles voor elke eigenschapsaccessor, zowel set als get. Al met al is dat veel werk – en dat wordt waarschijnlijk meer wanneer u later nieuwe functies toevoegt.

Als u het adresobject nodig hebt als `struct`, dan is een betere aanpak om het constant of *immutable* te maken. Maak eerst alle instantievelden read-only voor extern gebruik.

```
public struct Address
{
    // Remaining details elided
    public string Line1 { get; }
    public string Line2 { get; }
    public string City { get; }
    public string State { get; }
    public int ZipCode { get; }

    public Address(string line1,
        string line2,
        string city,
        string state,
        int zipCode) :
        this()
    {
        Line1 = line1;
        Line2 = line2;
        City = city;
        ValidateState(state);
        State = state;
        ValidateZip(zipCode);
        ZipCode = zipCode;
    }
}
```

Nu hebt u een constant type, gebaseerd op de public interface. Schrijf nu alle benodigde constructors voor de initialisering van `Address`-structuur. Deze structuur heeft slechts één extra constructor nodig, waarbij elk veld wordt opgegeven. Een kopieerconstructor is niet nodig omdat de toewijzingsoperator net zo efficiënt is. Onthoud dat de standaardconstructor nog steeds toeganke-

lijk is. Er is een standaardadres waar alle strings de waarde null hebben en de zipcode de waarde 0.

```
public Address(string line1,
    string line2,
    string city,
    string state,
    int zipCode) :
    this()
{
    Line1 = line1;
    Line2 = line2;
    City = city;
    ValidateState(state);
    State = state;
    ValidateZip(zipCode);
    ZipCode = zipCode;
}
```

Het gebruik van het constante type vereist een iets andere aanroep voor het wijzigen van zijn status. U maakt namelijk een nieuw object in plaats van het bestaande exemplaar aan te passen:

```
// Create an address:
Address a2 = new Address("111 S. Main",
    "", "Anytown", "IL", 61111);

// To change, re-initialize:
a2 = new Address(a1.Line1,
    a1.Line, "Ann Arbor", "MI", 48103);
```

De waarde van a1 vindt u op een van twee locaties: Anytown op de originele locatie of Ann Arbor op de bijgewerkte locatie. U wijzigt niet het bestaande adres, daarmee genereert u de ongeldige tijdelijke status zoals in het vorige voorbeeld. De tussentijdse locaties bestaan slechts tijdens de uitvoering van de adresconstructor en zijn niet zichtbaar buiten die constructor. Zodra een nieuw adresobject is gemaakt, wordt de waarde ervan voor altijd vastgelegd. Deze code is ook *exception safe*: a1 heeft ofwel de oorspronkelijke waarde of de nieuwe waarde. Als tijdens de bouw van het nieuwe adresobject een exception wordt gegenereerd, is de oorspronkelijke waarde van a1 ongewijzigd.

Als u een constant type creëert, moet u ervoor zorgen dat uw code geen gaten heeft die cliënten toestaan om de interne status te veranderen. Value-typen bieden geen ondersteuning voor afgeleide typen, dus u hoeft geen beveiliging

in te bouwen voor afgeleide typen die velden wijzigen. Echter, in een constant type moet u wel oppassen met elk veld dat verwijst naar een variabel reference-type. Wanneer u constructors voor deze typen implementeert, moet u een beveiligingskopie van dat variabele type maken. In de volgende voorbeelden wordt ervan uitgegaan dat `Phone` een constant value-type is, omdat we ons hier alleen bezighouden met constante value-typen:

```
// Almost immutable: There are holes that would
// allow state changes.
public struct PhoneList
{
    private readonly Phone[] phones;

    public PhoneList(Phone[] ph)
    {
        phones = ph;
    }

    public IEnumerable<Phone> Phones
    {
        get { return phones; }
    }
}

Phone[] phones = new Phone[10];
// Initialize phones
PhoneList pl = new PhoneList(phones);

// Modify the phone list:
// also modifies the internals of the (supposedly)
// immutable object.
phones[5] = Phone.GeneratePhoneNumber();
```

De arrayklasse is een reference-type. In dit voorbeeld verwijst de array binnen de `struct` PhoneList naar dezelfde geheugenlocatie van de array Phones die is toegewezen buiten het object. Dat wil zeggen dat ontwikkelaars nu uw constante structuur kunnen wijzigen met een andere variabele die naar dezelfde opslaglocatie verwijst. U elimineert deze mogelijkheid met een veiligheidskopie van de array. `Array` is een variabel type, dus de klasse `ImmutableArray` van de namespace `System.Collections.Immutable` zou een alternatief kunnen zijn. Het vorige voorbeeld toont de valkuilen van een variabele verzameling. Als het type `Phone` een variabel reference-type is, zijn er nog meer mogelijkheden voor ellende. Een client kan dan de waarden van de verzameling wijzigen, zelfs

ls de verzameling is beveiligd tegen wijziging. Dit probleem is eenvoudig te verhelpen als u het type `ImmutableList` gebruikt:

```
public struct PhoneList
{
    private readonly ImmutableList<Phone> phones;

    public PhoneList(Phone[] ph)
    {
        phones = ph.ToImmutableList();
    }

    public IEnumerable<Phone> Phones => phones;
}
```

Er zijn drie strategieën die u kunt gebruiken. De complexiteit van het type bepaalt welke van de drie u het beste kunt gebruiken voor het initialiseren van uw constante type. Ten eerste, u kunt één constructor definiëren waarmee clients een object kunnen initialiseren. Neem bijvoorbeeld de adresstructuur waarbij een constructor is gedefinieerd waarmee clients een adres kunnen initialiseren. Het opzetten van de benodigde constructeurs is vaak de eenvoudigste aanpak.

Ten tweede, u kunt factorymethoden maken om de structuur te initialiseren. Factory's maken het makkelijker om gemeenschappelijke waarden te creëren. Het .NET Frameworktype `Color` gebruikt deze strategie om systeemkleuren te initialiseren. De statische methoden `Color.FromKnownColor()` en `Color.FromName()` retourneren een kopie van een kleurwaarde die de huidige waarde voor een bepaalde systeemkleur vertegenwoordigt.

Ten derde, u kunt een variabele companionklasse maken wanneer meerstapsoperaties nodig zijn voor de volledige constructie van een constant type. De stringklasse van .NET gebruikt deze strategie met de klasse `System.Text.StringBuilder`. Met de klasse `StringBuilder` vult u een string in opeenvolgende stappen. Nadat alle bewerkingen die nodig zijn om de string te bouwen klaar zijn, haalt u de constante string op in `StringBuilder`.

De constante typen zijn eenvoudiger te schrijven en gemakkelijker te onderhouden. Schrijf niet blindelings get- en set-accessors voor elke eigenschap van een type. Implementeer In plaats daarvan, wanneer een type gegevens opslaat, deze dan als constante, atomaire value-typen. U bouwt met deze entiteiten eenvoudig ingewikkelde structuren.