

Imperatief programmeren

Jeroen Fokker
Departement Informatica
Universiteit Utrecht



augustus 2018

Korte inhoudsopgave

1	Programmeren	5
2	Hallo, C#!	15
3	Methoden om te tekenen	30
4	Objecten en methoden	46
5	Interactie	60
6	Herhaling	75
7	Keuze	84
8	Objecten en klassen	96
9	Arrays en strings	116
10	Libraries / taalconcepten / toepassingen	135
11	Algoritmen	187
12	Klassen	208
A	Gebruik van Visual Studio	214
B	Werkcollege-opgaven	218
C	Practicumopdrachten	234
D	Class library Imperatief programmeren	244
E	Syntax	256

Inhoudsopgave

1	Programmeren	5
1.1	Computers en programma's	5
1.2	Orde in de chaos	6
1.3	Programmeerparadigma's	8
1.4	Programmeertalen	9
1.5	Vertalen van programma's	11
1.6	Programmeren	12
2	Hallo, C#!	15
2.1	Soorten programma's	15
2.2	Opbouw van een programma	16
2.3	Syntax-diagrammen	17
2.4	Console-applicaties	19
2.5	Windows-applicaties: componenten op een Form	22
2.6	Static en this	26
2.7	Windows-applicatie: zelf tekenen van een Form	27
3	Methoden om te tekenen	30
3.1	De klasse Graphics	30
3.2	Variabelen	33
3.3	Berekeningen	34
3.4	Programma-layout	38
3.5	Declaraties met initialisatie	39
3.6	Methode-definities	40
3.7	Op zoek naar parameters	43
4	Objecten en methoden	46
4.1	Variabelen	46
4.2	Objecten	48
4.3	Object-variabelen	50
4.4	Typering	56
4.5	Constanten	58
5	Interactie	60
5.1	Controls op een Form	60
5.2	Methoden met een resultaat	64
5.3	Forms ontwerpen met Visual Studio	66
6	Herhaling	75
6.1	De while-opdracht	75
6.2	bool waarden	77
6.3	De for-opdracht	78
6.4	Bijzondere herhalingen	80
6.5	Toepassing: renteberekening	81

7	Keuze	84
7.1	De if-opdracht	84
7.2	Voorbeelden van if-opdrachten	85
7.3	Exceptions	89
7.4	Toepassing: grafiek en nulpunten van een parabool	91
8	Objecten en klassen	96
8.1	Klasse: beschrijving van een object	96
8.2	Toepassing: Bewegende deeltjes	98
8.3	Animatie	104
8.4	Klasse-ontwerp en -gebruik	106
8.5	Subklassen	110
8.6	Klasse-hiërarchieën	113
9	Arrays en strings	116
9.1	Array: rij variabelen van hetzelfde type	116
9.2	Syntax van arrays	123
9.3	String: een immutable array characters	125
9.4	Arrays als turftabel	128
10	Libraries / taalconcepten / toepassingen	135
10.1	Menu's / lambda-expressies / Bitmap-editor	135
10.2	Panels / switch&break / Calculator	144
10.3	Files / abstracte klassen / Tekst-editor	150
10.4	Collections / interfaces	163
10.5	Het tekenprogramma: "Schets"	170
11	Algoritmen	187
11.1	Een console-tool om tekst te zoeken	187
11.2	Automatische taalherkenning	190
11.3	Zoeken in een netwerk	198
12	Klassen	208
12.1	Klassen	208
A	Gebruik van Visual Studio	214
A.1	Installatie van de software	214
A.2	De eerste keer Visual Studio starten	214
A.3	De werking van de Visual Studio IDE	214
B	Werkcollege-opgaven	218
C	Practicumopdrachten	234
C.0	Oefenpracticum	234
C.1	Mandelbrot	236
C.2	Reversi-spel	239
C.3	SchetsPlus	241
D	Class library Imperatief programmeren	244
D.1	System	244
D.2	System.Drawing	246
D.3	System.Windows.Forms	248
D.4	System.Threading	251
D.5	System.IO	252
D.6	System.Collections.Generic	254
E	Syntax	256

Hoofdstuk 1

Programmeren

1.1 Computers en programma's

Computer: processor plus geheugen

Een computer bestaat uit tientallen verschillende componenten, en het is een vak apart om dat allemaal te beschrijven. Maar als je het heel globaal bekijkt, kun je het eigenlijk met twee woorden zeggen: een computer bestaat uit een *processor* en uit *geheugen*. Dat geheugen kan allerlei vormen aannemen, voornamelijk verschillend in de snelheid van gegevensoverdracht en de toegangssnelheid. Sommig geheugen kun je lezen en schrijven, sommig geheugen alleen lezen of alleen met wat meer moeite beschrijven, en er is geheugen dat je alleen kunt beschrijven.

Invoer- en uitvoer-apparatuur (toetsenbord, muis, monitor, printer enz.) lijken op het eerste gezicht buiten de categorieën processor en geheugen te vallen, maar als je ze maar abstract genoeg beschouwt vallen ze in de categorie “geheugen”: een toetsenbord is “read only” geheugen, en een monitor is “write only” geheugen. Ook de netwerkkaart, en met een beetje goede wil de geluidkaart, zijn een vorm van geheugen.

De processor, daarentegen, is een wezenlijk ander onderdeel. Taak van de processor is het uitvoeren van *opdrachten*. Die opdrachten hebben als effect dat het geheugen wordt veranderd. Zeker met onze ruime definitie van “geheugen” verandert of inspecteert praktisch elke opdracht die de processor uitvoert het geheugen.

Opdracht: voorschrift om geheugen te veranderen

Een opdracht is dus een voorschrift om het geheugen te veranderen. De opdrachten staan zelf ook in het geheugen (eerst op een disk, en terwijl ze worden uitgevoerd ook in het interne geheugen). In principe zou het programma opdrachten kunnen bevatten om een ander deel van het programma te veranderen. Dat idee is een tijdje erg in de mode geweest (en de verwachtingen voor de kunstmatige intelligentie waren hooggespannen), maar dat soort programma's bleken wel erg lastig te schrijven: ze veranderen waar je bij staat!

We houden het er dus maar op dat het programma in een afzonderlijk deel van het geheugen staat, apart van het deel van het geheugen dat door het programma wordt veranderd. Het programma wordt, alvorens het uit te voeren, natuurlijk wel in het geheugen geplaatst. Dat is de taak van een gespecialiseerd programma, dat we een *operating system* noemen (of anders een *virus*).

Programma: lange reeks opdrachten

Ondertussen zijn we aan een definitie van een programma gekomen: een programma is een (lange) reeks opdrachten, die –als ze door de processor worden uitgevoerd– het doel hebben om het geheugen te veranderen.

Programmeren is de activiteit om dat programma op te stellen. Dat vergt het nodige voorstellingsvermogen, want je moet je de hele tijd bewust zijn wat er met het geheugen zal gebeuren, later, als het programma zal worden uitgevoerd.

Voorbeelden van “programma's” in het dagelijks leven zijn talloos, als je bereid bent om het begrip “geheugen” nog wat ruimer op te vatten: kookrecepten, routebeschrijvingen, bevoorradingsstrategieën van een supermarktketen, ambtelijke procedures, het protocol voor de troonswisseling: het zijn allemaal reeksen opdrachten, die als ze worden uitgevoerd, een bepaald effect hebben.

Programmeertaal: notatie voor programma's

De opdrachten die samen het programma vormen moeten op een of andere manier worden geformuleerd. Dat zou met schema's of handbewegingen kunnen, maar in de praktijk gebeurt het vrijwel

altijd door de opdrachten in tekst-vorm te coderen. Er zijn vele verschillende notaties in gebruik om het programma mee te formuleren. Zo'n verzameling notatie-afspraken heet een *programmeertaal*. Daar zijn er in de recente geschiedenis nogal veel van bedacht, want telkens als iemand een nóg handigere notatie bedenkt om een bepaald soort opdrachten op te schrijven wordt dat al gauw een nieuwe programmeertaal.

Hoeveel programmeertalen er bestaan is moeilijk te zeggen, want het ligt er maar aan wat je meetelt: versies, dialecten enz. In Wikipedia (en.wikipedia.org/wiki/List_of_programming_languages) staat een overzicht van bijna 1000 talen, naar keuze alfabetisch, historisch, of naar afkomst gesorteerd.

Het heeft weinig zin om die talen allemaal te gaan leren, en dat hoeft ook niet, want er is veel overeenkomst tussen talen. Wel is het zo dat er in de afgelopen 60 jaar een ontwikkeling heeft plaatsgevonden in programmeertalen. Ging het er eerst om om steeds meer nieuwe mogelijkheden van computers te gebruiken, tegenwoordig ligt de nadruk er op om een beetje orde te scheppen in de chaos die het programmeren anders dreigt te veroorzaken.

1.2 Orde in de chaos

Omvang van het geheugen

Weinig zaken hebben zo'n spectaculaire groei doorgemaakt als de omvang van het geheugen van computers. In 1948 werd een voorstel van Alan Turing om een (één) computer te bouwen met een geheugencapaciteit van 6 kilobyte nog afgekeurd (te ambitieus, te duur!). Tegenwoordig zit dat geheugen al op de klantenkaart van de kruidenier. Maar ook recent is de groei er nog niet uit: tien jaar geleden had de modale PC een geheugen van 256 megabyte, en niet van 8192 megabyte (8 gigabyte) zoals nu. Voor disks geldt een zelfde ontwikkeling: tien jaar geleden was 40 gigabyte best acceptabel, nu is dat eerder 1024 gigabyte (1 terabyte). En wat zouden we over tien jaar denken van onze huidige 4 gigabyte DVD'tjes?

Variabele: geheugenplaats met een naam

Het geheugen is voor programma's aanspreekbaar in de vorm van *variabelen*. Een variabele is een plaats in het geheugen met een naam. Een opdracht in het programma kan dan voorschrijven om een bepaalde, bij naam genoemde, variabele te veranderen. Voor kleine programma's gaat dat prima: enkele tientallen variabelen zijn nog wel uit elkaar te houden. Maar als we al die nieuw verworven megabytes met aparte variabelen gaan vullen, worden dat er zoveel dat we daar het overzicht totaal over verliezen.

In wat oudere programmeertalen is het om die reden dan ook vrijwel niet mogelijk te voldoen aan de eisen die tegenwoordig aan programmatuur wordt gesteld (windowinterface, geheel configureerbaar, what-you-see-is-what-you-get, gebruik van alle denkbare rand- en communicatieapparatuur, onafhankelijk van taal, cultuur en schriftsoort, geïntegreerde online help en zelfdenkende wizards voor alle klusjes...).

Object: groepje variabelen

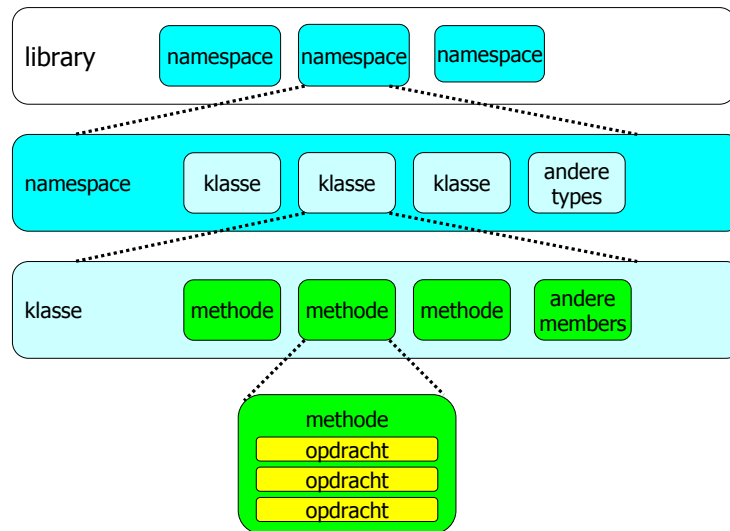
Er is een bekende oplossing die je kunt gebruiken als, door het grote aantal, dingen onoverzichtelijk dreigen te worden: groeperen, en de groepjes een naam geven. Dat werkt voor personen in verenigingen, verenigingen in bonden, en bonden in federaties; het werkt voor gemeenten in provincies, provincies in deelstaten, deelstaten in landen, en landen in unies; het werkt voor werknemers in afdelingen, afdelingen in divisies, divisies in bedrijven, bedrijven in holdings; en voor de opleidingen, in departementen in faculteiten in de universiteit.

Dat moet voor variabelen ook kunnen werken. Een groepje variabelen die bij elkaar horen en als geheel met een naam kan worden aangeduid, staat bekend als een *object*. In de zogenaamde object-georiënteerde programmeertalen kunnen objecten ook weer in een variabele worden opgeslagen, en als zodanig deel uitmaken van grotere objecten. Zo kun je in programma's steeds grotere gehelen manipuleren, zonder dat je steeds met een overweldigende hoeveelheid details wordt geconfronteerd.

Omvang van programma's

Programma's staan ook in het geheugen, en omdat daar zo veel van beschikbaar is, worden programma's steeds groter. Twintig jaar geleden pasten operating system, programmeertaal en tekstverwerker samen in een ROM van 256 kilobyte; de nieuwste tekstverwerkers worden geleverd op meerdere DVD's à 4 gigabyte.

In een programma staan een enorme hoeveelheid opdrachten, en het is voor één persoon niet meer te bevatten wat die opdrachten precies doen. Erger is, dat ook met een team er moeilijk uit te komen is: steeds moet zo'n team weer vergaderen over de precieze taakverdeling.



Figuur 1: Terminologie voor hiërarchische structurering van programma's

Methode: groepje opdrachten met een naam

Het recept is bekend: we moeten wat orde in de chaos scheppen door de opdrachten te groeperen, en van een naam te voorzien. We kunnen dan door het noemen van de naam nonchalant grote hoeveelheden opdrachten aanduiden, zonder ons steeds in alle details te verdiepen. Dat is de enige manier om de complexiteit van grote programma's nog te kunnen overzien.

Dit principe is al vrij oud, al wordt zo'n groepje opdrachten door de geschiedenis heen steeds anders genoemd (de naam van elk apart groepje wordt uiteraard door de programmeur bepaald, maar het gaat hier om de naam van de naamgevings-activiteit...). In de vijftiger jaren van de vorige eeuw heette een van naam voorzien groepje opdrachten een *subroutine*. In de zestiger jaren ging men spreken van een *procedure*. In de tachtiger jaren was de *functie* in de mode, en in de jaren negentig moest je van een *methode* spreken om er nog bij te horen.

We houden het nog steeds maar op "methode", maar hoe je het ook noemt: het gaat er om dat de complexiteit van lange reeksen opdrachten nog een beetje te beheersen blijft door ze in groepjes in te delen, en het groepje van een naam te voorzien.

Klasse: groepje methoden met een naam

Decennia lang kon men heel redelijk uit de voeten met hun procedures. Maar met de steeds maar groeiende programma's onstond er een nieuw probleem: het grote aantal procedures werd te onoverzichtelijk om nog goed hanteerbaar te zijn.

Het recept is bekend: zet de procedures in samenhangende groepjes bij elkaar en behandel ze waar mogelijk als één geheel. Zo'n groepje heet een *klasse*. (Overigens zitten er in een klasse ook nog andere dingen dan alleen methodes; een methode is slechts één van de mogelijk *members* van een klasse).

Namespace: groepje klassen met een naam

Niet iedereen hoeft opnieuw het wiel uit te vinden. Door de jaren heen zijn er vele klassen geschreven, die in andere situaties opnieuw bruikbaar zijn. Vroeger heette dat de *standard library*, maar naarmate het er meer werden, en er ook alternatieve libraries ontstonden, werd het handig om ook klassen weer in groepjes te bundelen. Zo'n groepje klassen (bijvoorbeeld: alles wat met file-input/output te maken heeft, of alles wat met interactieve interfaces te maken heeft) wordt een *namespace* genoemd. (Overigens zitten er in een namespace ook nog andere dingen dan alleen klassen; een klasse is slechts één van de mogelijk *types* die in een namespace zitten).

1.3 Programmeerparadigma's

Imperatief programmeren: gebaseerd op opdrachten

Ook in de wereld van de programmeertalen kunnen we wel wat orde in de chaos gebruiken. Programmeertalen die bepaalde eigenschappen gemeen hebben behoren tot hetzelfde programmeerparadigma. (Het woord “paradigma” is gestolen van de wetenschapsfilosofie, waar het een gemeenschappelijk kader van theorievorming in een bepaalde periode aanduidt; heel toepasselijk dus.)

Een grote groep programmeertalen behoort tot het *imperatieve paradigma*; dit zijn dus *imperatieve programmeertalen*. In het woord “imperatief” herken je de “gebiedende wijs”; imperatieve programmeertalen zijn dan ook talen die gebaseerd zijn op opdrachten om het geheugen te veranderen. Imperatieve talen sluiten dus direct aan op het besproken computermodel met processor en geheugen. In deze cursus staat de imperatieve programmeertaal C# centraal, en dat verklaart dan ook de naam van de cursus.

Declaratief programmeren: gebaseerd op functies

Het feit dat we de moeite nemen om de imperatieve talen als zodanig te benoemen doet vermoeden dat er nog andere paradigma's zijn, waarin geen opdrachten gebruikt worden. Kan dat dan? Wat doet de processor, als hij geen opdrachten uitvoert?

Het antwoord is, dat de processor weliswaar altijd opdrachten uitvoert, maar dat je dat in de programmeertaal niet noodzakelijk hoeft terug te zien. Denk bijvoorbeeld aan het maken van een ingewikkeld spreadsheet, waarbij je allerlei verbanden legt tussen de cellen op het werkblad. Dit is een activiteit die je “programmeren” kunt noemen, en het nog-niet-ingevulde spreadsheet is het “programma”, klaar om actuele gegevens te verwerken.

Het “programma” is niet op het geven van opdrachten gebaseerd, maar veeleer op het leggen functionele verbanden tussen de diverse cellen.

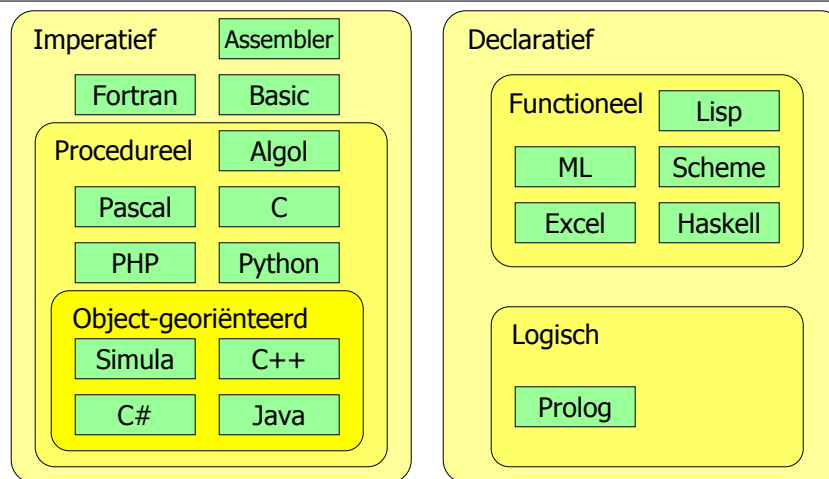
Naast dit soort *functionele programmeertalen* zijn er nog talen die op de propositiologica zijn gebaseerd: de *logische programmeertalen*. Samen staan deze bekend als het *declaratieve paradigma*. Maar daar gaat deze cursus dus niet over.

Procedureel programmeren: imperatief + methoden

Programmeertalen waarin procedures (of methoden, zoals we tegenwoordig zouden zeggen) een prominente rol spelen, behoren tot het *procedurele* paradigma. Alle procedurele talen zijn bovendien imperatief: in die procedures staan immers opdrachten, en de aanwezigheid daarvan maakt een taal imperatief.

Object-georiënteerd programmeren: procedureel + objecten

Weer een uitbreiding van procedurele talen vormen de object-georiënteerde talen. Hierin kunnen niet alleen opdrachten gebundeld worden in procedures (of liever: methoden), maar kunnen bovendien variabelen gebundeld worden in objecten.



Figuur 2: Programmeerparadigma's

1.4 Programmeertalen

Imperatieve talen: Assembler, Fortran, Basic

De allereerste computers werden geprogrammeerd door de instructies voor de processor direct, in getalvorm, in het geheugen neer te zetten. Al snel kreeg men door dat het handig was om voor die instructies gemakkelijk te onthouden afkortingen te gebruiken, in plaats van getallen. Daarmee was rond 1950 de eerste echte programmeertaal ontstaan, die *Assembler* werd genoemd, omdat je er gemakkelijk programma's mee kon bouwen ("to assemble"). Elke processor heeft echter zijn eigen instructies, dus een programma in Assembler is specifiek voor een bepaalde processor. Je moet dus eigenlijk niet spreken van "de taal Assembler", maar liever van "Assembler-talen".

Dat was natuurlijk niet handig, want als er een nieuwe type processor wordt ontwikkeld zijn al je oude programma's waardeloos geworden. Een nieuwe doorbraak was rond 1955 de taal Fortran (een afkorting van "formula translator"). De opdrachten in deze taal waren niet specifiek geënt op een bepaalde processor, maar konden (met een speciaal programma) worden vertaald naar diverse processoren. De taal werd veel gebruikt voor technisch-wetenschappelijke toepassingen. Nog steeds trouwens; niet dat modernere talen daar niet geschikt voor zouden zijn, maar omdat er in de loop der jaren nu eenmaal veel programmatuur is ontwikkeld, en ook omdat mensen niet zo gemakkelijk van een eenmaal aangeleerde taal afstappen.

Voor beginners was Fortran een niet zo toegankelijke taal. Dat was aanvankelijk niet zo erg, want zo'n dure computer gaf je natuurlijk niet in handen van beginners. Maar na verloop van tijd (omstreeks 1965) kwam er toch de behoefte aan een taal die wat gemakkelijker in gebruik was, en zo ontstond Basic ("Beginner's All-purpose Symbolic Instruction Code"). De taal is later vooral populair geworden doordat het de standaard-taal werd van "personal" computers: de Apple II in 1978, de IBM-PC in 1979, en al hun opvolgers. Helaas was de taal niet gestandaardiseerd, zodat op elk merk computer een apart dialect werd gebruikt, dat niet uitwisselbaar was met de andere.

Procedurele talen: Algol, Pascal, C, PHP, Python

Ondertussen was het inzicht doorgebroken dat voor wat grotere programma's het gebruik van procedures onontbeerlijk was. De eerste echte procedurele taal was Algol (een wat merkwaardige afkorting van "Algorithmic Language"). De taal werd in 1960 gelanceerd, met als bijzonderheid dat de taal een officiële definitie had, wat voor de uitwisselbaarheid van programma's erg belangrijk was. Er werd voor de gelegenheid zelfs een speciale notatie (BNF) gebruikt om de opbouw van programma's te beschrijven, die (anders dan Algol zelf) nog steeds gebruikt wordt.

In het vooruitgangsgeloof van de zestiger jaren was in 1968 de tijd rijp voor een nieuwe versie: Algol68. Een grote commissie ging er eens goed voor zitten en voorzag de taal van allerlei nieuwe ideeën. Zo veel ideeën dat het erg lastig was om vertalers te maken voor Algol68-programma's. Die kwamen er dan ook nauwelijks, en dat maakt dat Algol68 de dinosauriërs achterna is gegaan: uitgestorven vanwege zijn complexiteit.

Het was wel een leerzame ervaring voor taal-ontwerpers: je moest niet willen streven naar een taal met eindeloos veel toeters en bellen, maar juist naar een compact en simpel taaltje. De eerste simpele, maar wel procedurele, taal werd als éénmansactie bedacht in 1971: Pascal (geen afkorting, maar een vernoeming naar de filosoof Blaise Pascal). Voornaamste doel van ontwerper Wirth was het onderwijs aan de universiteit van Zürich te voorzien van een gemakkelijk te leren, maar toch verantwoorde (procedurele) taal. Al gauw werd de taal ook voor serieuze toepassingen gebruikt; allicht, want mensen stappen niet zo gauw af van een eenmaal aangeleerde taal.

Voor echt grote projecten was Pascal echter toch te beperkt. Zo'n groot project was de ontwikkeling van het operating system *Unix* eind jaren zeventig bij Bell Labs. Het was sowieso nieuw om een operating system in een procedurele taal te schrijven (tot die tijd gebeurde dat in Assembler-talen), en voor deze gelegenheid werd een nieuwe taal ontworpen: C (geen afkorting, maar de opvolger van eerdere prototypes genaamd A en B). Het paste in de filosofie van Unix dat iedereen zijn eigen uitbreidingen kon schrijven (nieuwe editors en dergelijke). Het lag voor de hand dat die programma's ook in C werden geschreven, en zo werd C de belangrijkste imperatieve taal van de jaren tachtig, ook buiten de Unix-wereld.

Ook recente talen om snel en makkelijk een web-pagina te genereren (PHP) of data te manipuleren (Perl, Python) zijn procedureel.

Oudere Object-georiënteerde talen: Simula, Smalltalk, C++

In 1967 was de Noorse onderzoeker Dahl geïnteresseerd in programma's die simulaties uit konden voeren (van het gedrag van rijen in een postkantoor, de doorstroming van verkeer, enz.). Het was in die tijd al niet zo raar meer om je eigen taal te ontwerpen, en zo ontstond de taal Simula als een uitbreiding van Algol60. Een van die uitbreidingen was het *object* als zelfstandige eenheid. Dat kwam handig uit, want een persoon in het postkantoor of een auto in het verkeer kon dan mooi als object worden beschreven. Simula was daarmee de eerste object-georiënteerde taal.

Simula zelf leidde een marginaal bestaan, maar het object-idee werd in 1972 opgepikt door onderzoekers van Xerox in Palo Alto, die (eerder dan Apple en Microsoft) experimenteerden met window-systemen en een heuse muis. Hun taaltje (genaamd "Smalltalk") gebruikte objecten voor het modelleren van windows, buttons, scrollbars en dergelijke: allemaal min of meer zelfstandige objecten.

Maar Smalltalk was wel erg apart: werkelijk alles moest een object worden, tot aan getallen toe. Dat werd niet geaccepteerd door de massa. Toch was duidelijk dat objecten op zich wel handig waren. Er zou dus een C-achtige taal moeten komen, waarin objecten gebruikt konden worden. Die taal werd C++ (de twee plustekens betekenen in C "de opvolger van", en elke C-programmeur begreep dus dat C++ bedoeld was als opvolger van de taal C). De eerste versie is van 1978, en de officiële standaard verscheen in 1981.

De taal is erg geschikt voor het schrijven van window-gebaseerde programma's, en dat begon in die tijd net populair te worden. Maar het succes van C++ is ook toe te schrijven aan het feit dat het echt een uitbreiding is van C: de oude constructies uit C bleven bruikbaar. Dat kwam goed uit, want mensen stappen nu eenmaal niet zo gemakkelijk af van een eenmaal aangeleerde taal.

De taal C++ is weliswaar standaard, maar de methode-bibliotheken die nodig zijn om window-systemen te maken zijn dat niet. Het programmeren van een window op een Apple-computer, een Windows-computer of een Unix-computer moet dan ook totaal verschillend worden aangepakt, en dat maakt de interessantere C++-programma's niet uitwisselbaar met andere operating systems. Oorspronkelijk vond men dat nog niet eens zo heel erg, maar dat werd anders toen rond 1995 het Internet populair werd: het was toch jammer dat de programma's die je via het Internet verspreidde slechts door een deel van het publiek gebruikt kon worden (mensen met hetzelfde operating system als jij).

Java

Tijd dus voor een nieuwe programmeertaal, ditmaal eentje die gestandaardiseerd is voor gebruik onder diverse operating systems. De taal zou moeten lijken op C++, want mensen stappen nu eenmaal niet zo gemakkelijk af van een eenmaal aangeleerde taal, maar het zou een mooie gelegenheid zijn om de nog uit C afkomstige en minder handige ideeën overboord te zetten.

De taal Java vervulde deze rol (geen afkorting, geen filosoof, maar de naam van het favoriete koffiemerk van de ontwerpers). Java is in 1995 gelanceerd door hardwarefabrikant Sun, die daarbij gebruikmaakte van een toen revolutionair business model: de software is gratis, en verdiend moest er worden op de ondersteuning. Ook niet onbelangrijk voor Sun was het om tegenwicht te bieden voor de groeiende populariteit van Microsoft-software, die niet werkte op de Unix-computers die Sun maakte.

Een vernieuwing in Java was verder dat de taal zo was ingericht dat programma's niet per ongeluk konden interfereren met andere programma's die op dezelfde computer draaien. In C++ was dat een groeiend probleem aan het worden: als zo'n fout per ongeluk optrad kon het de hele computer platleggen, en erger nog: kwaadwillende programmeurs konden op deze manier virussen en spyware introduceren. Met het downloaden van programma's via het Internet werd dit een steeds groter probleem. Java is, anders dan C++, *sterk getypeerd*: de programmeur legt het type van variabelen vast (getal, tekst, object met een bepaalde opbouw) en kan daarna niet een object ineens als getal gaan behandelen. Bovendien wordt het programma niet direct op de processor uitgevoerd, maar onder controle van een *virtuele machine*, die controleert of het geheugen echt gebruikt wordt zoals dat door de typering is aangegeven.

C#

Ondertussen zat Microsoft natuurlijk ook niet stil: rond 2000 lanceerde dit bedrijf ook een nieuwe object-georiënteerde, sterk getypeerde programmeertaal die gebruik maakt van een virtuele machine (Microsoft noemt dit *managed code*). De naam van deze taal, C#, geeft al aan dat deze taal in de traditie van C en C++ verder gaat. Het hekje lijkt typografisch zelfs een beetje op aan elkaar

Java-versies			C#-versies		
JDK	1.0	jan 1996			
JDK	1.1	feb 1997			
J2SE	1.2	dec 1998			
J2SE	1.3	mei 2000	C#	1	2000
J2SE	1.4	feb 2002	C#	1.2	jan 2002
J2SE	5.0	sep 2004	C#	2.0	nov 2005
Java SE	6	dec 2006	C#	3.0	nov 2006
Java SE	7	juli 2011	C#	4.0	apr 2010
Java SE	8	mrt 2014	C#	5.0	aug 2012
			C#	6.0	jul 2015
Java SE	9	sep 2017	C#	7.0	mrt 2017
Java SE	10	mrt 2018	C#	7.3	mei 2018

Figuur 3: Versiegeschiedenis van Java en C#

gegroeide ++ tekens. In de muziekwereld symboliseert zo'n hekje een verhoging van een noot, en het wordt in het Engels uitgesproken als 'sharp'; het is mooi meegenomen dat 'sharp' in het Engels ook nog 'slim' betekent. De suggestie is: C# is een slimme vorm van C. (In het Nederlands gaat die woordspeling niet op, want Nederlandse musici noemen # een 'kruis'.)

Zowel Java als C# maakten een ontwikkeling door: elke paar jaar onstond er wel weer een nieuwe versie met nieuwe features, al dan niet geïnspireerd door de nieuwe features in de vorige versie van de concurrent (zie figuur 3). In de recente versies van C# sluipen ondertussen ook features uit het functionele paradigma binnen. Java heeft een gratis 'Standard Edition' (SE), en een 'Enterprise Edition' (EE) voor bedrijven die willen betalen voor extra ondersteuning en libraries. C# heeft een gratis 'Community' editie (voor individuen, organisaties tot 5 personen, onderwijs, en open source software ontwikkeling), en een 'Enterprise' editie voor bedrijven.

Waar dit alles toe moet leiden is lastig te voorspellen. Java en C# leven al vijftien jaar naast elkaar en er is nog geen winnaar aan te wijzen. Ook C++ is nog volop in gebruik, maar hoe lang nog? Gaan nog in dit decennium hippe geïnterpreteerde scripttalen zoals PHP en Python de markt overnemen van de klassieke gecompileerde object-georiënteerde talen?

In ieder geval is C# eenvoudiger te leren dan C++ (dat door de compatibiliteit met C een nogal complexe taal is), en is het in C# iets gemakkelijker om interactieve programma's te schrijven dan in Java. Je kunt er dus sneller interessante programma's mee schrijven. Object-georiënteerde ideeën zijn in C# prominent aanwezig, en het kan zeker geen kwaad om die te leren. Andere object-georiënteerde talen (C++, Java, of nog weer andere) zijn, met C# als basiskennis, relatief gemakkelijk bij te leren. En dat kan nooit kwaad, want er is geen enkele reden nooit meer af te stappen van een eenmaal geleerde taal...

1.5 Vertalen van programma's

Een computerprogramma wordt door een speciaal programma "vertaald" voor gebruik op een bepaalde computer. Afhankelijk van de omstandigheden heet zo'n vertaalprogramma een assembler, een compiler, of een interpreter. We bespreken de verschillende mogelijkheden hieronder; zie figuur 4 voor een overzicht.

Assembler

Een *assembler* wordt gebruikt voor het vertalen van Assembler-programma's naar machinecode. Omdat een Assembler-programma specifiek is voor een bepaalde processor, heb je voor verschillende computers verschillende programma's nodig, die elk door een overeenkomstige assembler worden vertaald.

Compiler

Het voordeel van alle talen behalve Assembler is dat ze, in principe althans, geschreven kunnen worden onafhankelijk van de computer. Er is dus maar één programma nodig, dat op een computer naar keuze kan worden vertaald naar de betreffende machinecode. Zo'n vertaalprogramma heet

een *compiler*. De compiler zelf is wel machine-specifiek; die moet immers de machinecode van de betreffende computer kennen. Het door de programmeur geschreven programma (de *source code*, of kortweg *source*, of in het Nederlands: *broncode*) is echter machine-onafhankelijk. Vertalen met behulp van een compiler is gebruikelijk voor de meeste procedurele talen, zoals C en C++.

Interpreter

Een directere manier om programma's te vertalen is met behulp van een *interpreter*. Dat is een programma dat de broncode leest, en de opdrachten daarin direct uitvoert, dus zonder deze eerst te vertalen naar machinecode. De interpreter is specifiek voor de machine, maar de broncode is machine-onafhankelijk.

Het woord “interpreter” betekent letterlijk “tolk”, dit naar analogie van het vertalen van mensentaal: een compiler kan worden vergeleken met schriftelijk vertalen van een tekst, een interpreter vertaalt de uitgesproken zinnen direct mondeling.

Het voordeel van een interpreter boven een compiler is dat er geen aparte vertaalslag nodig is. Het nadeel is echter dat het uitvoeren van het programma langzamer gaat, en dat eventuele fouten in het programma niet in een vroeg stadium door de compiler gemeld kunnen worden.

Vertalen met behulp van een interpreter is gebruikelijk voor de wat eenvoudigere talen, in de recente historie vooral de talen die bedoeld zijn om flexibel data te manipuleren (bijvoorbeeld Perl, PHP, Python).

Compiler+interpreter

Bij Java is voor een gemengde aanpak gekozen. Java-programma's zijn bedoeld om via het Internet te verspreiden. Het verspreiden van de gecompileerde versie van het programma is echter niet handig: de machinecode is immers machine-specifiek, en dan zou je voor elke denkbare computer aparte versies moeten verspreiden. Maar het verspreiden van broncode is ook niet altijd wenselijk; dan ligt de tekst van het programma immers voor het oprapen, en dat is om redenen van auteursrecht niet altijd de bedoeling. Het komt veel voor dat gebruikers het programma wel mogen gebruiken, maar niet mogen inzien of wijzigen; machinecode is voor dat doel heel geschikt.

De aanpak die daarom voor Java wordt gehanteerd is een compiler die de broncode vertaalt: maar niet naar machinecode, maar naar een nog machine-onafhankelijke tussenliggende taal, die *bytecode* wordt genoemd. Die bytecode kan via het Internet worden verspreid, en wordt op de computer van de gebruiker vervolgens met behulp van een interpreter uitgevoerd. De bytecode is dusdanig eenvoudig, dat de interpreter erg simpel kan zijn; interpreters kunnen dus worden ingebouwd in Internet-browsers. Omdat het meeste vertaalwerk al door de compiler is gedaan, kan het interpreteren van de bytecode relatief snel gebeuren, al zal een naar “echte” machinecode gecompileerd programma altijd sneller kunnen worden uitgevoerd.

Compiler+compiler

Platform-onafhankelijkheid is bij Microsoft nooit een prioriteit geweest. Toch wordt ook in C# een gemengde aanpak gebruikt, waarbij een tussenliggende taal een rol speelt die hier *intermediate language* wordt genoemd. Ditmaal is de bijzonderheid dat ook vanuit andere programmeertalen dezelfde intermediate code kan worden gegenereerd. Grotere projecten kunnen dus programma's in verschillende programmeertalen integreren.

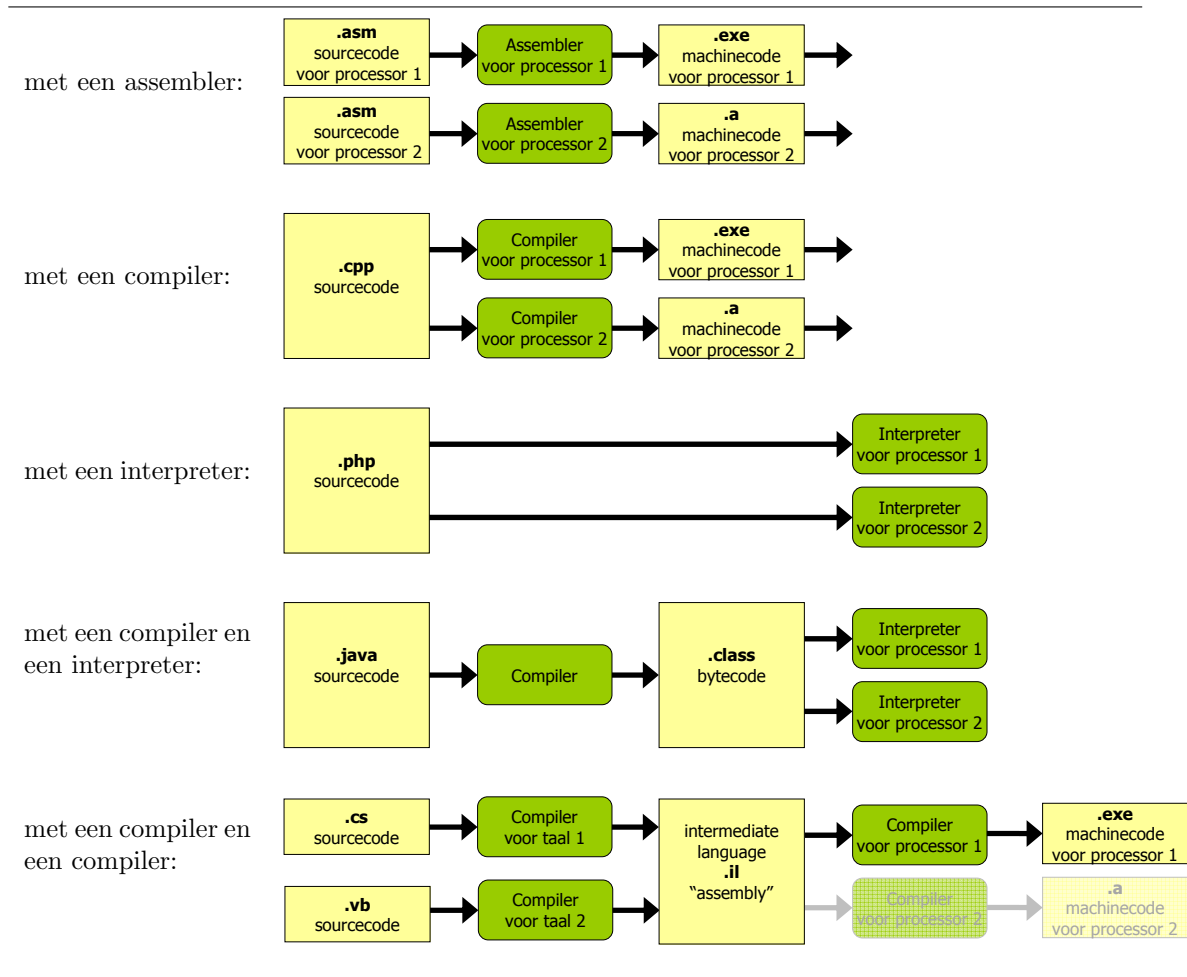
Uiteindelijk wordt de intermediate language toch weer naar machinecode vertaald, en anders dan bij Java gebeurt dit met een compiler. Soms gebeurt dat pas in een laat stadium, namelijk op het moment dat blijkt dat een deel van het programma echt nodig is — de scheidslijn met een interpreter begint dan wat onduidelijk te worden. De compiler wordt dan een *just-in-time compiler* of *jitter* genoemd.

Verwarrend is dat een bestand met intermediate code een *assembly* wordt genoemd (letterlijk: een ‘samengesteld ding’). Dit heeft echter niets te maken met de hierboven besproken ‘assembler-talen’.

1.6 Programmeren

In het klein: Edit-Compile-Run

Omdat een programma een tekst is, begint het implementeren over het algemeen met het tikken van de programmatekst met behulp van een editor. Is het programma compleet, dan wordt het bestand met de broncode aangeboden aan de compiler. Als het goed is, maakt de compiler de bijbehorende intermediate code en daarvan weer een uitvoerbaar bestand, dat we vervolgens kunnen runnen.



Figuur 4: Vijf manieren om een programma te vertalen

Zo ideaal verloopt het meestal echter niet. Het bestand dat je aan de compiler aanbiedt moet wel geldige C#-code bevatten: je kunt moeilijk verwachten dat de compiler van willekeurige onzin een uitvoerbaar bestand kan maken. De compiler controleert dan ook of de broncode aan de vereisten voldoet; zo niet, dan volgt er een foutmelding, en weigert de compiler om code te genereren.

Nu doe je over het algemeen wel je best om een echt C#-programma te compileren, maar een tikfout is snel gemaakt, en de vorm-vereisten voor programma's zijn nogal streng. Reken er dus maar op dat je een paar keer door de compiler wordt terugverwezen naar de editor.

Vroeg of laat zal de compiler echter wel tevreden zijn, en een uitvoerbaar bestand produceren. Dan kun je de volgende fase in: het *uitvoeren* van het programma, in het Engels *run* of *execute* genoemd, en in het Nederlands dus ook wel *runnen* of *executeren*. In veel gevallen merk je dan dat het programma toch net niet (of helemaal niet) doet wat je bedoeld had. Natuurlijk heb je je best gedaan om de bedoeling goed te formuleren, maar een denkfout is snel gemaakt.

Er zit dan niets anders op om weer terug te keren naar de editor, en het programma te veranderen. Dan weer compileren (en hopen dat je geen nieuwe tikfouten gemaakt hebt), en dan weer runnen. Om tot de conclusie te komen dat er nu wel iets anders gebeurt, maar toch *nét* niet wat je bedoelde. Terug naar de editor. . .

In het groot: Modelleer-Specificeer-Implementeer

Zodra de doelstelling van een programma iets ambitieuzer wordt, kun je niet direct achter de editor plaatsnemen en het programma beginnen te tikken. Aan het *implementeren* (het daadwerkelijk schrijven en testen van het programma) gaan nog twee fasen vooraf.

Als eerste zul je een praktijkprobleem dat je met behulp van een computer wilt oplossen moeten formuleren in termen van een programma dat invoer krijgt van een gebruiker en bepaalde resultaten te zien zal geven. Deze fase, het *modelleren* van het probleem, is misschien wel het moeilijkste.

Is het eenmaal duidelijk wat de taken zijn die het programma moet uitvoeren, dan is de volgende stap om een overzicht te maken van de klassen die er nodig zijn, en de methoden die daarin ondergebracht gaan worden. In deze fase hoeft van de methoden alleen maar beschreven te worden *wat* ze moeten doen, nog niet *hoe* dat precies gebeurt. Bij dit specificeren zul je wel in de gaten moeten houden dat je niet het onmogelijke van de methoden verwacht: ze zullen later immers geïmplementeerd moeten worden.

Als de specificatie van de methoden duidelijk is, kun je beginnen met het *implementeren*. Daarbij zal de edit-compile-run cyclus waarschijnlijk meermalen doorlopen worden. Is dat allemaal af, dan kun je het programma overdragen aan de opdrachtgever. In veel gevallen zal die dan opmerken dat het weliswaar een interessant programma is, maar dat er toch eigenlijk een net iets ander probleem opgelost moest worden. Dan begint het weer van voren af aan met het herzien van de modellering, gevolgd door aanpassing van de specificatie en een nieuwe implementatie, en dan. . .

Hoofdstuk 2

Hallo, C#!

2.1 Soorten programma's

C# is opgezet als universele programmeertaal. De taalconstructies zijn hetzelfde, of het nu om een fotobewerkingsprogramma, een database-server, of een game gaat. De algemene structuur van een programma, die we in dit hoofdstuk bespreken, is voor al deze doeleinden hetzelfde.

Voor de specifieke invulling van het programma maakt het natuurlijk wel uit om wat voor soort programma het gaat: de opzet van verschillende games lijkt meer op elkaar dan op die van kantoor-applicaties, en omgekeerd.

Op detailniveau maakt het gek genoeg weer minder uit om wat voor programma het gaat: van dichtbij bekeken is een C#-programma voor elke C#-programmeur herkenbaar; je hoeft geen specialist op een bepaald soort applicaties te zijn om een programma te kunnen begrijpen.

Bij de ontwikkeling van een programma moet je meteen al een keuze maken hoe dat programma communiceert met de gebruiker. Deze keuze drukt een zware stempel op de opbouw van het programma. Enkele veelgebruikte vormen zijn:

- *Console-applicatie*: er is alleen een simpel tekstschermd voor boodschappen aan de gebruiker, en meestal kan de gebruiker via een toetsenbord ook iets intikken. Communicatie met de gebruiker heeft noodgedwongen de vorm van een vraag-antwoord dialoog. Soms geeft de gebruiker alleen aan het begin wat input, waarna het programma voor langere tijd aan het werk gaat, en pas aan het eind de resultaten presenteert.
- *Windows-applicatie*: er is een grafisch scherm beschikbaar waarop meerdere windows zichtbaar zijn. Elk programma heeft een eigen window (dat eventueel ook verdeeld kan zijn in sub-windows). Het programma heeft een *grafische user-interface (GUI)*: de gebruiker kan met (meestal) een muis en/of het toetsenbord de inhoud van het window manipuleren, en verwacht daarbij directe grafische feedback (bij het aanklikken van een getekende button moet een verandering van de schaduwrand suggereren dat de button wordt ingedrukt). De communicatie wordt *event-driven* genoemd: de gebruiker (of andere externe invloeden) veroorzaakt gebeurtenissen (muiskliks, menukeuzes enz.) en het programma moet daarop reageren.
- *Game*: ook hier is een grafisch scherm aanwezig, met een veelal snel veranderend beeld. Het scherm kan een window zijn, maar heeft vaak ook een vaste afmeting op speciale hardware. De gebruiker kan de gepresenteerde grafische wereld direct manipuleren met muis, joystick, game-controller, nunchuck enz., of zelfs met z'n vingers op een aanraakschermd. Het toetsenbord speelt een ondergeschikte rol en kan zelfs afwezig zijn.
- *Web-applicatie (server side script)*: het programma is verantwoordelijk voor de opbouw van een web-pagina, die wordt gepresenteerd in een web-browser. Er is alleen aan het begin input van de gebruiker, in de vorm van keuzes die gemaakt zijn op de vorige pagina (aangeklikte link, ingevuld web-formulier). Door het achtereenvolgens tonen van meerdere pagina's kan er voor de gebruiker toch een illusie van interactie ontstaan.
- *Applet*: een kleine applicatie, die uitgevoerd wordt binnen de context van een web-browser, maar nu wordt het programma uitgevoerd op de *client*, dus op de computer van de gebruiker en niet op de web-server. De naam, die er door de suffix '-plet' uitziet als verkleinwoord en daarmee aangeeft dat het typisch om kleine programma's gaat, is bedacht door Sun voor client-side web-applicaties in de programmeertaal Java.
- *Mobiele applicatie* of kortweg *App*: een (nog kleinere?) applicatie, die uitgevoerd wordt op de mobiele telefoon van de gebruiker. Schermruimte is beperkt, de gebruiker kan wel dingen op het scherm aanwijzen maar niet veel tekst-invoer doen. Nieuwe mogelijkheden ontstaan

```

using System;

class Hallo1
{
    5     static void Main()
        {
            Console.WriteLine("Hallo!");
            Console.ReadLine();
        }
    10 }

```

Listing 1: Hallo1/Hallo1.cs

daarentegen als met GPS de locatie van het apparaat beschikbaar is, en/of er sensoren zijn voor de ruimtelijke oriëntatie. De naam is bedacht door Apple voor programma's op de iPhone, maar wordt ook wel gebruikt voor andere telefoons.

In dit hoofdstuk bespreken we eerst de universele opbouw van een C#-programma, met een console-applicatie als voorbeeld. Daarna komen achtereenvolgens de specifieke opbouw van een windows-applicatie (in drie varianten) en die van een game aan de orde.

In de rest van de cursus concentreren we ons grotendeels op windows-applicaties. Daarbij komt de specifieke opbouw voor dat soort toepassingen ter sprake, maar vooral ook veel van de taalconstructies in C#. Met die taalconstructies kun je ook uit de voeten in toepassingen uit een andere categorie (bijvoorbeeld console-applicaties of web-applicaties), ook al komen die in deze cursus niet uitgebreid aan de orde.

2.2 Opbouw van een programma

Opdracht: bouwsteen van een imperatief programma

In een imperatief programma doen de *opdrachten* het eigenlijke werk: de opdrachten worden één voor één uitgevoerd. Daarmee verandert het geheugen en/of het scherm, zodat de gebruiker er ook wat van merkt als het programma draait.

blz. 16

In listing 1 staat een van de kortst denkbare C#-programma's. Het is een console-applicatie die de tekst **Hallo!** op het scherm zet. De opdracht die het eigenlijke werk doet is

```
Console.WriteLine("Hallo!");
```

Als we dit programma in een permanente console uitvoeren kunnen we het hierbij laten. Maar omdat een console-applicatie soms in een tijdelijk console-window wordt uitgevoerd, dat wegfloept als het programma is afgelopen, zetten we er voor de zekerheid nog een tweede opdracht achter:

```
Console.ReadLine();
```

Deze opdracht blijft wachten totdat de gebruiker een regel tekst intikt en die afsluit met de Enter-toets. Met de intgetikte tekst doen we verder niets, maar zo heeft de gebruiker in ieder geval de tijd om de tekst die het programma toonde te lezen.

Methode: groepje opdrachten met een naam

Omdat C# een procedurele taal is, zijn de opdrachten gebundeld in *methoden*. Ook al zijn er in dit programma maar twee opdrachten, het is verplicht ze te bundelen in een methode. Opdrachten kunnen niet 'los' in een programma staan.

Het bundelen gebeurt met behulp van accolades { en }. Zo'n blok met opdrachten vormt de *body* van een methode. Daar boven staat de *header* van de methode, in dit geval:

```
static void Main()
```

waarin onder andere de naam van de methode geïntroduceerd wordt. De naam van de methode mag je als programmeur vrij kiezen, met dien verstande dat er in elk programma één methode moet zijn met de naam **Main**. Omdat dit onze enige methode is, hebben we in dit geval geen keus.

Klasse: groepje methoden met een naam

Omdat C# een object-georiënteerde taal is, zijn de methoden gebundeld in *klassen*. Ook al is er in dit programma maar één methode, het is verplicht hem te bundelen in een klasse. Methoden kunnen niet ‘los’ in een programma staan.

Ook het bundelen van methoden gebeurt met accolades. Rondom onze enige methode komt dus nog een stel accolades, met daar boven de header van de klasse:

```
class Hallo1
```

De klasse-header bestaat simpelweg uit het woord `class` met daarachter de naam van de klasse. De naam van de klasse mag je als programmeur echt vrij kiezen; in dit geval is dat dus `Hallo1`. Je ziet dat een naam behalve letters ook cijfers mag bevatten. De naam moet wel met een letter beginnen, en moet helemaal zonder spaties aan elkaar worden geschreven.

Compilatie-eenheid: groepje klassen in een file

De programmatekst staat opgeslagen in een tekstbestand. In een bestand kunnen meerdere klassen staan: de klasse-headers en de accolades geven duidelijk aan waar de grenzen liggen. Een tekstfile wordt in zijn geheel door de compiler gecompileerd, en vormt dus een zogeheten *compilatie-eenheid*. In het voorbeeld is er maar één klasse in de compilatie-eenheid.

De klassen van een programma mogen gespreid worden over meerdere files, dus over meerdere compilatie-eenheden, maar dat is in dit voorbeeld niet nodig.

Using: gebruik van libraries

De bovenste regel van onze compilatie-eenheid is geen onderdeel van de klasse:

```
using System;
```

Met deze regel geven we aan dat in het programma klassen gebruikt mogen worden die beschikbaar zijn in een *library* genaamd `System`. Eén van die klassen is de klasse `Console`, die op zijn beurt de methode `WriteLine` bevat.

Aanroep van een methode

De opdracht

```
Console.WriteLine("Hallo!");
```

is een *aanroep* van die methode. Dat wil zeggen: met deze opdracht wordt de methode `WriteLine` aan het werk gezet, die het kunstje kan doen dat we hier nodig hebben: het schrijven van een regel tekst op de console.

Bij een aanroep kan er nog wat extra informatie worden doorgespeeld aan de methode. In het geval van `WriteLine` is dat de tekst die getoond moet worden. Zulke extra informatie wordt een *parameter* van de methode genoemd. Parameters van een methode staan altijd tussen haakjes.

Bij de aanroep van de methode `ReadLine` is er geen extra informatie nodig. Toch moeten de haakjes er staan, maar in dit geval dus met niets er tussen.

2.3 Syntax-diagrammen

Syntax: grammatica van de taal

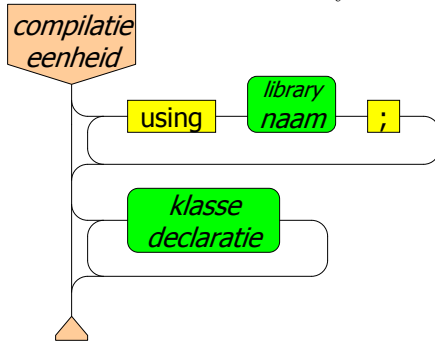
Het is lastig om in woorden te beschrijven hoe een C#-programma precies is opgebouwd. Een voorbeeld zoals in de vorige sectie maakt veel duidelijk, maar met een paar voorbeelden weet je nog steeds niet wat er nou precies wel en niet mag in de taal.

Daarom gaan we de ‘grammatica’ van C# (de zogeheten *syntax*) beschrijven met diagrammen: *syntax-diagrammen*. Volg de route van links naar rechts door het ‘rangeerterrein’, en je ziet precies wat er allemaal nodig is. Woorden in een gele/lichtgekleurde rechthoek moet je letterlijk opschrijven; cursieve woorden in een groene/donkergekleurde ovaal verwijzen naar een ander syntax-diagram voor de details van een bepaalde deelconstructie. Bij elke splitsing is er een keuze; bochten moeten vloeiend genomen worden en je mag niet achteruitrijden. (In sommige diagrammen staan als toelichting nog vertikaal geschreven woorden op licht/blauw vlak; voor het ‘rangeren’ zijn die niet van belang).

We geven hier de syntax-diagrammen voor de constructies die in de vorige sectie werden besproken: *compilatie-eenheid*, de daarin gebruikte *klasse-declaratie*, en de daarin op zijn beurt gebruikte *member*. Deze schema’s bevatten een iets versimpelde versie van de werkelijkheid. De volledige schema’s worden later besproken; een overzicht staat in appendix E.

Syntax van compilatie-eenheid

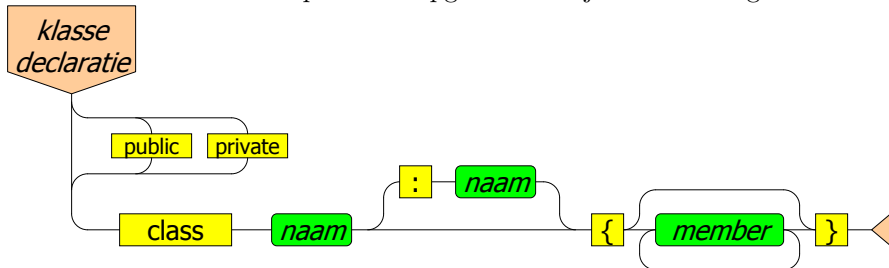
Hier is het schema voor de syntax van een *compilatie-eenheid*:



Uit dit schema wordt duidelijk dat zowel de **using** regels als de klasse-declaraties herhaald mogen worden. Desgewenst mag je ze overslaan, en in het meest extreme geval kom je helemaal niets tegen tussen startpunt en eindpunt. Inderdaad is een leeg bestand een geldige compilatie-eenheid: niet erg nuttig, maar wel toegestaan. Verder kun je zien dat aan het eind van de **using** regel een puntkomma moet staan.

Syntax van klasse-declaratie

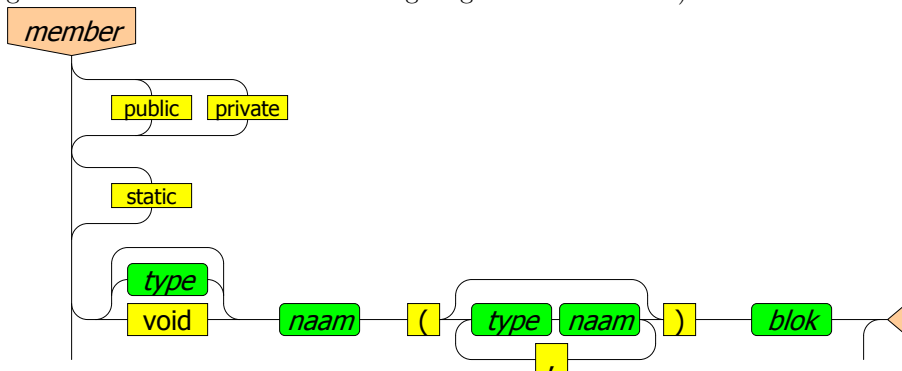
Hoe een *klasse-declaratie* precies is opgebouwd blijkt uit het volgende schema:



Duidelijk is dat aan het woord **class** en de naam daarachter niet valt te ontkomen. Ook de accolades zijn verplicht. De *member* tussen de accolades kun je desgewenst passeren, maar in de praktijk zal het juist vaker voorkomen dat je meer dan één member in de klasse wilt schrijven. Ook dat is mogelijk. Je zult je allicht afvragen waar dat zijspoor met de dubbele punt voor dient; die gaan we gebruiken in sectie 2.5.

Syntax van member

Er zijn verschillende soorten *members* mogelijk in een klasse, maar de belangrijkste is voorlopig de methode-definitie. De syntax daarvan is voorlopig als volgt (de doodlopende einden onderaan geven aan dat het schema later nog uitgebreid zal worden):

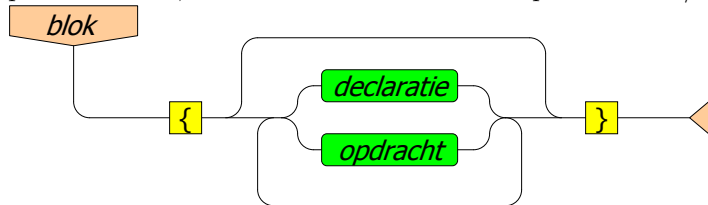


Je kunt dit schema gebruiken om je er van te overtuigen dat de methode-header uit het voorbeeld

```
static void Main()
```

gevolgd door het blok met de methode-body een geldige *member* vormt. Het woord **static** mag volgens dit schema desgewenst ook worden weggelaten; in plaats van **void** staat er ook wel eens een *type* (wat dat ook moge zijn), of soms juist helemaal niets, de haakjes zijn weer wel verplicht en daar staat soms ook nog iets ingewikkelds tussen.

Het aparte syntax-diagram van *blok* maakt duidelijk dat de body van een methode bestaat uit een paar accolades, met daartussen nul of meer opdrachten en/of declaraties.

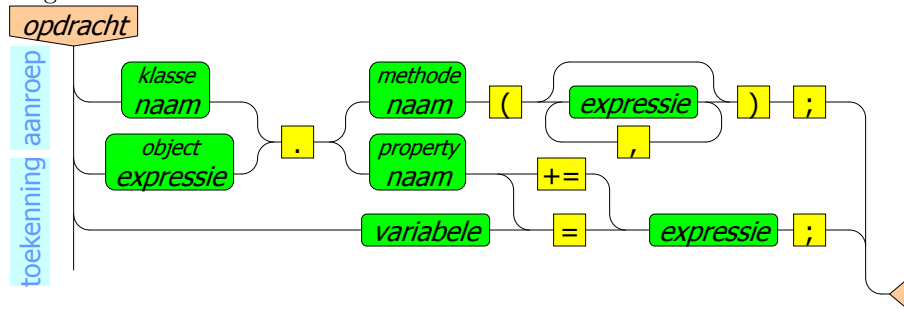


De syntax van begrip *declaratie* bespreken we in de volgende sectie, de syntax van een *opdracht* hieronder.

Syntax van opdracht

Opdrachten vormen de kern van elk imperatief programma, dus ook van een C#-programma: ze worden één voor één door de computer uitgevoerd. Het syntax-diagram van het begrip *opdracht* is dan ook het grootste van de grammatica van C#; er zijn een tiental verschillende soorten opdrachten.

We beginnen met de syntax van twee soorten opdracht, die worden beschreven door het volgende diagram:



Van de verschillende routes door dit schema hebben we de bovenste nodig gehad voor het construeren van de opdrachten in het voorbeeldprogramma. De onderste route gaan we gebruiken in de volgende sectie, en de slingeroute middendoor in sectie 2.7.

Syntax en semantiek

Weten hoe een opdracht (althans één van de tien mogelijke vormen) is opgebouwd is één ding, maar het is natuurlijk ook van belang om te weten wat er gebeurt als zo'n opdracht wordt uitgevoerd. Dat heet de *betekenis* of *semantiek* van de opdracht.

Semantiek van een methode-aanroep

Als een methode-aanroep door de processor wordt uitgevoerd, dan zal de processor op dat moment de opdrachten gaan uitvoeren die in de body van die methode staan. Pas als die allemaal zijn uitgevoerd, gaat de processor verder met de opdracht die volgt op de methode-aanroep.

Het aardige is dat de opdrachten in de body van de aangeroepen methode ook weer methode-aanroepen mogen zijn, van weer andere methoden. Beschouw het maar als een soort uitbesteden van werk aan anderen: als een methode geen zin heeft om het werk zelf uit te voeren, wordt een andere methode aangeroepen om het vuile werk op te knappen.

In het voorbeeldprogramma zijn beide opdrachten in de body van de methode `Main` een methode-aanroep, van respectievelijk de methode `WriteLine` en `ReadLine`.

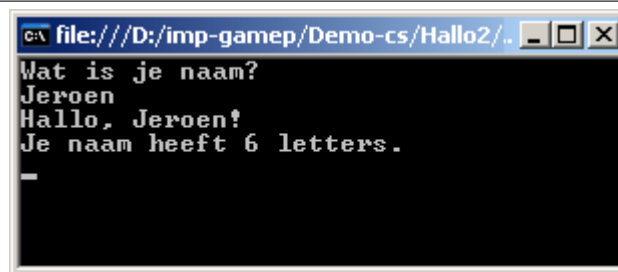
2.4 Console-applicaties

In deze sectie breiden we het voorbeeld uit tot een console-applicatie waarin de gebruiker echt input kan geven die door het programma verwerkt wordt. Het programma staat in listing 2; in figuur 5 is het programma in werking te zien. Het nodigt de gebruiker uit om zijn/haar naam in te tikken, en kan daarmee een persoonlijke groet construeren. En omdat de naam 'computer' suggereert dat het apparaat kan rekenen, berekent het programma meteen ook maar het aantal letters van de naam.

```
using System;

class Hallo2
{
5   static void Main()
    {
        string naam;
        Console.WriteLine("Wat is je naam?");
        naam = Console.ReadLine();
10   Console.WriteLine("Hallo, " + naam + "!");
        Console.WriteLine("Je naam heeft " + naam.Length + " letters." );
        Console.ReadLine();
    }
}
```

Listing 2: Hallo2/Hallo2.cs



Figuur 5: Het programma Hallo2 in werking

Variabele: geheugenplaats met een naam

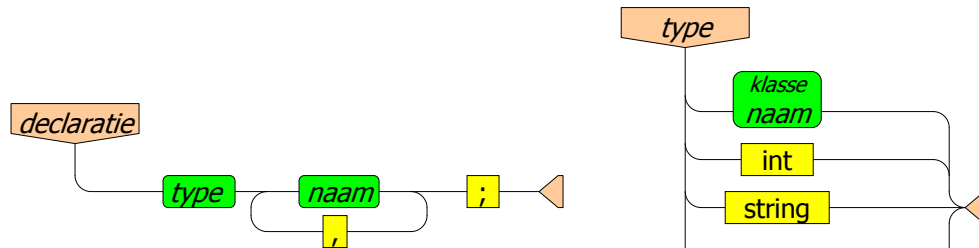
De gebruiker is uitgenodigd om zijn/haar naam in te tikken. Met de methode-aanroep `Console.ReadLine()` wordt die naam opgevraagd. Om die naam verderop in het programma ook te kunnen gebruiken, moet hij worden opgeslagen in het geheugen. Daarom hebben we een *variabele* nodig: een plek in het geheugen met een naam, waarin de waarde kan worden opgeslagen, en waarvan we de waarde later weer kunnen opvragen.

Declaratie: reserveren van geheugenruimte voor een variabele

Elke variabele die nodig is in een programma moet worden aangekondigd. Dat gebeurt door middel van een *declaratie*. In dit geval is dat:

```
string naam;
```

Dit is de (vereenvoudigde) syntax van een declaratie:



De declaratie bestaat uit twee onderdelen:

- Het *type* van de variabele. In dit geval is dat **string**, want dat is het type van een variabele waarin teksten kunnen worden opgeslagen. Een ander mogelijk type is **int**, het type van gehele getallen, maar ook de naam van een klasse kan als type worden gebruikt (we zullen dat in de volgende sectie ook gaan gebruiken).
- De *naam* van de variabele. Deze naam mag de programmeur zelf kiezen. In dit geval kozen we **naam**, omdat deze variabele gebruikt gaat worden om de naam van de gebruiker in op te slaan. Desgewenst kun je met één declaratie meerdere variabelen van hetzelfde type declareren.

Een declaratie is *geen* opdracht: er valt immers, tijdens het runnen, niets “uit te voeren” aan een declaratie. Een declaratie is veeleer een aanwijzing voor de compiler dat bepaalde variabelen gebruikt gaan worden en van een bepaald type zijn.

Declaraties staan in een programma tussen de opdrachten in het blok dat de body van een methode vormt. De declaratie moet eerder in de methode staan dan de opdracht waarin de variabele voor het eerst gebruikt wordt. Sommige programmeurs zetten alle declaraties helemaal aan het begin van de methode, andere wisselen de declaraties en opdrachten af, en declareren een variabele pas op het punt waar hij voor het eerst nodig is.

Syntax van de toekenningsopdracht

Je kunt variabelen een waarde geven met een zogeheten *toekennings-opdracht* (engels: *assignment statement*). Dat gebeurt in het programma als volgt:

```
naam = Console.ReadLine();
```

Een toekenningsopdracht bestaat dus uit:

- de naam van de variabele die een waarde moet krijgen
- het teken =
- de nieuwe waarde van de variabele
- een puntkomma.

Na de methode-aanroep in is de toekenningsopdracht de tweede vorm van opdrachten die we zijn tegengekomen. De syntax van deze opdracht-vorm stond al in het syntaxdiagram in de vorige sectie.

Kenmerkend voor de toekennings-opdracht is het =-teken in het midden. Je kunt dit teken beter uitspreken als ‘wordt’ dan als ‘is’. De variabele *is* immers nog niet gelijk aan de waarde rechts van het =-teken, hij *wordt* het door het uitvoeren van de opdracht.

Rechts van het =-teken staat volgens dat diagram een *expressie*. In dit geval is dat die expressie `Console.ReadLine()`. Blijkbaar is een methode-aanroep (maar dan zonder de puntkomma) een mogelijke invulling van een expressie. Maar er zijn nog vele andere mogelijkheden, die we in latere programma's zullen tegenkomen.

Declaraties zijn lokaal

De gedeclareerde variabelen mogen alleen worden gebruikt in de resterende opdrachten van de methode waar de declaratie in staat. Buiten de methode is de variabele niet bekend. Declaraties van variabelen worden daarom *lokale* declaraties genoemd. Een andere methode mag een variabele declareren die dezelfde naam heeft, maar dat is dan gewoon een andere variabele, die geen enkele relatie heeft met zijn naamgenoten in andere methoden.

Dit mechanisme is van groot belang in grotere programma's, waar verschillende methoden worden geschreven door verschillende programmeurs. Die programmeurs hoeven dan onderling niet af te spreken welke variabele-namen ze mogen gebruiken, en hoeven ook niet bang te zijn dat hun variabelen tijdens het aanroepen van een andere methode ineens van waarde veranderen.

Strings samenvoegen met +

Nu we de naam van de gebruiker hebben opgeslagen in de variabele `naam`, kunnen we de persoonlijke groet construeren. Die bestaat uit de vaste tekst 'hallo', dan de naam van de gebruiker, en om het geheel kracht bij te zetten nog een uitroepteken er achter. Vaste teksten staan in het programma met aanhalingstekens er omheen. De persoonlijke groet wordt geconstrueerd door de twee vaste teksten te combineren met de tekst in de variabele `naam`.

Samenvoegen van teksten tot één geheel gebeurt met de `+`-operator. De totale tekst wordt vervolgens meegegeven bij de aanroep van de methode `Console.WriteLine`, zodat hij op het scherm verschijnt:

```
Console.WriteLine( "Hallo, " + naam + "!" );
```

De tekst die tussen de aanhalingstekens staat wordt letterlijk zo gebruikt, dus inclusief de komma en de spatie achter de komma. Staan er geen aanhalingstekens, zoals bij `naam`, dan wordt de de tekst die in de variabele is opgeslagen gebruikt.

Spaties die buiten de aanhalingstekens staan, zoals rondom de plus-teken, komen niet in de tekst terecht. Ze staat er puur om het programma een beetje overzichtelijk te houden voor de menselijke lezer. De betekenis van het programma verandert er niet door. We hadden dus ook kunnen schrijven:

```
Console.WriteLine("Hallo, "+naam+"!");
```

Maar dat is dus minder overzichtelijk...

Property: eigenschap van een object

Als een variabele eenmaal van een waarde voorzien is, kun je er *properties* (oftewel *eigenschappen*) van opvragen. Welke eigenschappen er beschikbaar zijn verschilt per type. In het geval van het string-object dat in de variabele `naam` is opgeslagen, is er één eigenschap beschikbaar: `Length`, de lengte van de tekst.

Je kunt de eigenschap van een object opvragen door achter het object een punt te schrijven, gevolgd door de gewenste eigenschap. De lengte van de tekst die is opgeslagen in de variabele `naam` krijg je dus met `naam.Length` te pakken.

Voor de punt hoeft niet per se een variabele te staan. Je kunt ook de lengte van een vast tekst-object opvragen, zoals in `"Hallo".Length`, al is dat misschien niet zo zinvol omdat je dan net zo goed meteen 5 in het programma kunt opschrijven.

In het voorbeeldprogramma wordt de opgevraagde lengte met wat toelichtende tekst eromheen teruggemeld aan de gebruiker.

2.5 Windows-applicaties: componenten op een Form

In deze sectie maken we twee programma's die de opbouw van een windows-applicatie demonstren. Een minimaal voorbeeld staat in listing 3, een wat serieuzere aanpak in listing 4. In figuur 6 zijn de programma's, en de derde versie uit sectie 2.7 in werking te zien.

Form: het type van een window-object

Zoals `string` het type is van een tekst-object, zo is er ook een type voor een object dat een window voorstelt. Je zou misschien verwachten dat dat type 'window' heet, maar de ontwerpers van de bewuste library zaten blijkbaar meer te denken aan het feit dat een window vaak als een soort invul-formulier dient, want ze hebben het type `Form` genoemd. Dit type is beschikbaar in de library `System.Windows.Forms`, wat in feite een sub-sub-library is van `System`. Niettemin moet

```
using System.Windows.Forms;
using System.Drawing;

class HalloWin1
5 {
    static void Main()
    {
        Form scherm;
        scherm = new Form();
10     scherm.Text = "Hallo";
        scherm.BackColor = Color.Yellow;
        scherm.Size = new Size(200,100);
        Application.Run(scherm);
    }
15 }
```

Listing 3: HalloWin1/HalloWin1.cs

```
using System.Windows.Forms;
using System.Drawing;

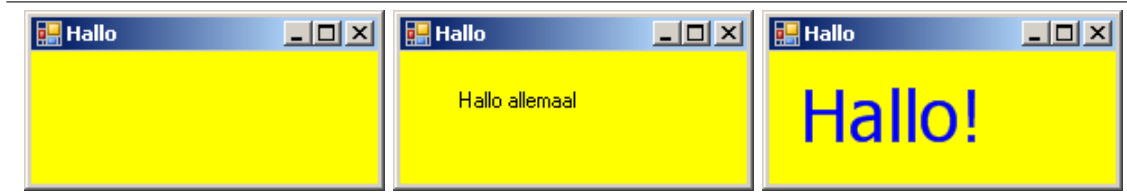
class HalloForm : Form
5 {
    public HalloForm()
    {
        this.Text = "Hallo";
        this.BackColor = Color.Yellow;
10     this.Size = new Size(200, 100);

        Label groet;
        groet = new Label();
        groet.Text = "Hallo allemaal";
15     groet.Location = new Point(30, 20);

        this.Controls.Add(groet);
    }
}
20

class HalloWin2
{
    static void Main()
    {
25     HalloForm scherm;
        scherm = new HalloForm();
        Application.Run(scherm);
    }
}
```

Listing 4: HalloWin2/HalloWin2.cs



Figuur 6: De drie versies van het programma HalloWin in werking

deze library apart vermeld worden in een `using`-directief. De library `System` hebben we in dit programma trouwens niet nodig.

new: aanmaken van een nieuw object

De eerste regel in `Main` is de declaratie van een variabele met het type `Form`:

```
Form scherm;
```

Zo'n declaratie geeft je nog niet een `Form`-object in handen. Het enige wat je krijgt is een variabele die mogelijk later in het programma naar een `Form`-object kan verwijzen. Dat gebeurt echter pas als de variabele met een toekennings-opdracht een waarde krijgt.

Dat roept de vraag op: hoe kom je aan een nieuw `Form`-object? Het antwoord is bijna letterlijk datzelfde, want er is een `C#` taalconstructie waarmee je nieuwe objecten van een bepaald type kunt maken. Zo kun je een nieuw `Form`-object maken met `new Form()`. Dat gebruiken we aan de rechterkant van een toekenningsopdracht, die een waarde geeft aan de zojuist gedeclareerde variabele:

```
scherm = new Form() ;
```

Nu hebben we in de variabele `scherm` een `Form`-object beschikbaar, maar dat is daarmee nog niet automatisch in beeld. Daarvoor is er een speciale methode `Run` in de klasse `Application`. Als parameter wil die methode een `Form`-object hebben, dus dat komt goed uit:

```
Application.Run(scherm);
```

Propertes veranderen

Zoals een string-object de property `Length` bezit, zo heeft ook een form-object een aantal eigenschappen: de titel van het window, de achtergrondkleur, de afmetingen, en nog veel meer. Een net `new` gemaakt form-object heeft standaardwaarden voor z'n eigenschappen (geen titel, grijze achtergrond, afmetingen 300 bij 300 pixels, enz.).

Het leuke is dat je deze eigenschappen met toekenningsopdrachten kunt veranderen (bij de lengte van een string kon dat natuurlijk niet). De titel van het window (de property `Text`) is een string, die we met aanhalingstekens ter plaatse neerzetten:

```
scherm.Text = "Hallo";
```

De achtergrondkleur (property `BackColor`) is geen string maar een `Color`-object. In principe kunnen we hier elke mengkleur opgeven, maar circa 200 standaardkleuren zijn direct beschikbaar in de klasse `Color`:

```
scherm.BackColor = Color.Yellow;
```

De afmetingen van het window (de property `Size`) is een object die als type ook `Size` heeft (de eerdere twee voorbeelden geven al aan dat het type niet altijd samenvalt met de naam van de property, maar hier is dat toevallig wel zo). Anders dan bij de kleuren zijn er geen standaardwaarden voor sizes, maar we kunnen de gewenste `Size` als `new` object creëren:

```
scherm.Size = new Size(200,100);
```

Subklasse: gespecialiseerde versie van een bestaande klasse

Een iets andere aanpak van de windows-applicatie staat in listing 4. Dit is de aanpak die in de praktijk het meest gebruikt wordt. Er zijn nu twee klassen in het programma. Eentje dient om de methode `Main` onderdak te bieden. In die methode wordt net als in het vorige programma een object gedeclareerd, `new` aangemaakt, en meegegeven aan methode `Run`.

Het verschil is dat het object ditmaal niet een `Form`-object is, maar een `HalloForm`. Dat is natuurlijk niet een bestaande klasse uit een library, maar een zelfbedacht type. Zo'n zelfbedacht type kun je aanmaken met een klasse-declaratie, en dat is dan ook de tweede klasse in het programma.

De header van de klasse `HalloForm` is:

```
class HalloForm : Form
```

Achter de naam van de klasse staat een dubbele punt en de naam van een bestaande klasse. Dit betekent dat `HalloForm` een gespecialiseerde versie is van de bestaande klasse `Form`. Daarom is de variabele `scherm`, die nu het type `HalloForm` heeft, toch acceptabel als parameter van `Run`: die verwacht eigenlijk een `Form`, maar gespecialiseerde versies daarvan mogen ook.

Constructormethode

Tijdens het maken van een nieuw object met `new` wordt er automatisch een speciale methode aangeroepen. Deze methode heeft dezelfde naam als het type van het nieuwe object, en wordt de *constructormethode* genoemd. In de klasse `HalloForm` is de enige methode zo'n constructormethode. Die wordt dus aangeroepen bij het `new` maken van het object, en dat is een goed moment om de properties van het object aan te passen. De toekenningen aan de properties `Text`, `BackColor` en `Size` staan nu dus in de constructormethode van `HalloForm`.

this: het object dat door de methode bewerkt wordt

De vraag is nu nog: van welk object moeten we in de constructormethode de properties veranderen? We kunnen hier niet, zoals in het vorige programma, schrijven

```
scherm.Text = "Hallo";
```

want de variabele `scherm` staat in een hele andere methode (en zelfs in een andere klasse) gedeclareerd.

Gelukkig is er toch een manier waarop de constructormethode kan refereren aan het kersverse object: er is in elke methode een speciaal object genaamd `this` beschikbaar, dat refereert aan 'het' object dat door de methode bewerkt wordt. In het geval van de constructormethode is dat het object dat zojuist nieuw gemaakt is. Aldus kunnen we in de constructormethode de properties wijzigen met

```
this.Text = "Hallo";
this.BackColor = Color.Yellow;
this.Size = new Size(200,100);
```

Controls op een formulier

Nu we toch bezig zijn om het formulier mooi te maken, kunnen we ook iets op het formulier gaan zetten. Tot nu toe was het window immers helemaal leeg; de tekst 'Hallo' stond in de titelbalk van het window – dat telt niet echt.

Een formulier kan gevuld worden met zogeheten *controls*. Denk daarbij aan buttons, tekstvelden, schuifregelaars, maar ook simpelweg aan een stukje tekst. Dat laatste heet een `Label`, en dat wordt in het voorbeeldprogramma gebruikt. Het aanmaken van een `Label` gebeurt op dezelfde manier als het aanmaken van het hele formulier: declareren, `new` maken, en properties naar keuze aanpassen:

```
Label groet;
groet = new Label();
groet.Text = "Hallo allemaal!";
groet.Location = new Point(30,20);
```

Tenslotte wordt het label toegevoegd aan de controls van het formulier met de opdracht:

```
this.Controls.Add(groet);
```

public: bruikbaar in andere klassen

Uit de syntax-diagrammen in sectie 2.3 was al duidelijk dat er in de header van een methode desgewenst het woord `public` of `private` mag staan. Daarmee wordt de *zichtbaarheid* van de methoden vastgelegd. Methoden die `private` zijn, mogen alleen vanuit de eigen klasse worden aangeroepen. Methoden die `public` zijn, kunnen ook vanuit andere klassen worden aangeroepen. Als je niets opgeeft, zijn de methoden automatisch `private`.

In het voorbeeldprogramma wordt de constructormethode `HalloForm` vanuit een andere klasse aangeroepen. Om dat mogelijk te maken moet de zichtbaarheid van deze methode `public` zijn.

2.6 Static en this

De punt-notatie

In de programma's tot nu toe is op verschillende plaatsen een notatie `A.B` gebruikt:

- bij het gebruiken van een sub-library: `using System.Windows.Forms;`
- bij het opvragen/wijzigen van een property: `naam.Length` of `scherm.BackColor`
- bij het opvragen van een standaardkleur: `Color.Yellow`
- bij het aanroepen van een methode: `Console.ReadLine()` of `Application.Run(scherm)`

De library-notatie laten we even buiten beschouwing, maar de andere drie hebben meer met elkaar te maken dan op het eerste gezicht lijkt.

Properties hebben betrekking op een object

Als je een property opvraagt, is dat de eigenschap van een bepaald object. Bijvoorbeeld: `naam.Length` is de lengte van een string-object dat de gebruiker heeft ingetikt en dat is opgeslagen in de variabele `naam`. En: `scherm.Size` is de afmeting van een form-object dat eerder nieuw is gemaakt en dat is opgeslagen in de variabele `scherm`. Voor de punt staat in deze gevallen een object: vaak een variabele, als je wilt ook een constante zoals in `"Hallo".Length`, en soms het speciale object `this`.

Static properties hebben geen betrekking op een object

Als je in de ontwikkelomgeving help-informatie opvraagt over `Color.Yellow` staat daar enigszins verrassend bij dat het hier ook om een property gaat. Hoe kan dat nu? Waarvan wordt er dan een eigenschap bepaald? Kun je 'de geel' ergens van bepalen, zoals je ook 'de lengte' en 'de afmeting' ergens van kunt bepalen? Nee natuurlijk.

Er staat in `Color.Yellow` dan ook niet een object voor de punt, want `Color` is geen object maar een klasse. Je bepaalt dan ook niet zozeer 'de geel van de klasse `Color`', als wel 'geel, zoals gedefinieerd in de klasse `Color`'.

Dit soort properties heten *static* properties. In overzichten zoals op de help-pagina's staat het er altijd duidelijk bij als een property static is. Bij het opvragen van static properties staat er dus altijd de naam van een *klasse* voor de punt.

Omgekeerd staat er bij 'gewone' (niet-static) properties altijd een *object* voor de punt. Je kunt immers niet de lengte van `String` bepalen, wel van *een bepaalde* string zoals `naam`. Netzomin kun je de afmeting van `Form` bepalen, maar wel van een bepaald formulier zoals `scherm`.

Static methoden

Bij de aanroep van de methodes `Console.WriteLine`, `Console.ReadLine`, en `Application.Run` staat er ook steeds de naam van een klasse voor de punt. Genoemde voorbeelden zijn dan ook alledrie static methodes.

Dit roept de vraag op of er dan ook niet-static methodes bestaan. Het antwoord is: jazekeer! Niet-static methodes komen eigenlijk veel vaker voor dan static methodes.

Niet-static methoden

Bij de aanroep van een niet-static methode, een 'gewone' methode dus eigenlijk, staat er niet een klasse maar een object links van de punt. Zo'n methode hebben we al gebruikt: de methode `Add` in

```
this.Controls.Add(groet);
```

is niet-static. Maar dit is een beetje ingewikkelde situatie, want hier staan *twee* punten in de opdracht.

Je moet zo'n keten van punten van links naar rechts lezen. Het begint met `this`, dat is in ieder geval een object. Van dat object wordt de property `Controls` bepaald (dat is blijkbaar een niet-static property, en dat is ook wel logisch, want je bepaalt de controls van *een bepaald* formulier).

Die controls nu zijn zelf een object (om precies te zijn, een object van het type `Collection`). Aan dat object wordt door `Add` iets toegevoegd. De methode bewerkt dus *een bepaald* collection-object, en het is dus een niet-static methode.

Object-georiënteerd programmeren

Het programmeer-paradigma van C# heet niet voor niets *object-georiënteerd*: de objecten staan centraal. Je maakt ze met `new`, bepaalt/verandert er properties van, en bewerkt ze met methoden.

Bijna altijd is er wel een relevant object om te bewerken: je zoekt in een tekst, tekent op een scherm, slaat iets op in een database, leest of schrijft een file, voegt iets toe aan een collectie, enz. Daarom zijn methoden standaard niet-static. In de uitzonderlijke gevallen waarin methoden *geen* object bewerken, en dus *wel* static zijn, staat dat er in de header van de methode speciaal bij.

Main is altijd static, constructor-methoden nooit

De methode **Main**, die in elk programma aanwezig is en die bij de start van het programma wordt aangeroepen, is static. Op het moment dat het programma start zijn er immers nog geen objecten gemaakt die **Main** zou kunnen bewerken.

Constructor-methoden worden automatisch aangeroepen als je een **new** object maakt. Ze zijn bedoeld om het net nieuwe object meteen aan het begin te bewerken, bijvoorbeeld om de properties een zinvolle startwaarde te geven. Daarbij is er dus altijd een object dat bewerkt wordt, en constructormethodes zijn dus nooit static.

Static methoden kennen geen ‘this’

Het speciale woord **this** duidt in een methode het object aan dat door de methode wordt bewerkt. Bij constructormethoden is dat het zojuist nieuw aangemaakte object, bij andere methoden verwijst **this** naar het object dat bij aanroep voor de punt werd geschreven. Static methoden bewerken niet een specifiek object, en in static methoden mag je dus geen **this** gebruiken.

2.7 Windows-applicatie: zelf tekenen van een Form

In sectie 2.5 zagen we een windows-applicatie, waar de tekst ‘Hallo’ getoond werd op een **Label**-control, die door aanroep van de methode **Add** was toegevoegd aan de **Control**-property van een **Form**.

In deze sectie volgen we een andere aanpak: we maken geen gebruik van labels of andere controls, maar schrijven de tekst ‘Hallo’ direct op het **Form**. Het programma staat in listing 5.

blz. 28

De opzet van het programma is in grote lijnen hetzelfde als de vorige versie: er is een statische methode **Main**, die een object aanmaakt van een zelfgemaakte subklasse van een library-klasse **Form**. Dat object wordt vervolgens meegegeven aan de statische methode **Run** in de klasse **Application**. De subklasse van **Form** heeft een constructormethode, waarin een aantal properties van het object **this** (dat in elke niet-statische methode beschikbaar is) worden gewijzigd: de titeltekst, de achtergrondkleur, en de afmetingen.

Reageren op een event

De vierde property die in de constructormethode wordt aangepast, **Paint**, is nieuw. Het gaat hier om een zogeheten *event*, een ‘gebeurtenis’ die op een zeker moment zal optreden. In dit geval is die gebeurtenis: het tekenen van het window.

Je kunt in een programma aangeven dat je het wilt weten als zo’n event in de toekomst zal optreden: ‘laat het me weten als het window getekend moet worden’. Dat gebeurt in het programma met de opdracht:

```
this.Paint += this.tekenScherm;
```

Anders dan bij de andere properties staat er hier niet **=** maar **+=**, omdat je de property niet verandert, maar je je er als het ware op ‘abonneert’. Aan de rechterkant van **+=** staat de naam van een methode: **tekenScherm**, en het object dat deze methode onder handen zal krijgen bij aanroep: **this**.

Maar **this.tekenScherm** is niet de aanroep van een methode; er staan immers geen haakjes achter de methodenaam. De opdrachten in de methode worden op dit moment dus nog niet uitgevoerd. Pas op het moment dat het event optreedt (in dit geval dus: de noodzaak tot tekenen van het window ontstaat) wordt de methode aangeroepen. Ook eventuele andere abonnees van dit event krijgen dan een seintje, door middel van het aanroepen van de methode die zij hebben opgegeven toen zij zich abonneerden.

Event-handlers

De methode die kan reageren op een event heet een *event-handler*. In dit geval is dat de methode **tekenScherm**, die verderop in de klasse wordt gedefinieerd. De benodigde parameters van een event-handler liggen vast. Dat kan per event verschillen; in het geval van het **Paint**-event zijn er twee parameters nodig: eentje van het type **object** en eentje van het type **PaintEventArgs**.

```
using System.Windows.Forms;
using System.Drawing;

class HalloForm : Form
5 {
    public HalloForm()
    {
        this.Text = "Hallo";
        this.BackColor = Color.Yellow;
10    this.Size = new Size(200, 100);
        this.Paint += this.tekenScherm;
    }

    void tekenScherm(object obj, PaintEventArgs pea)
15    {
        pea.Graphics.DrawString( "Hallo!"
                                , new Font("Tahoma", 30)
                                , Brushes.Blue
                                , 10, 10
20    );
    }
}

class HalloWin3
25 {
    static void Main()
    {
        HalloForm scherm;
        scherm = new HalloForm();
30    Application.Run(scherm);
    }
}
```

Listing 5: HalloWin3/HalloWin3.cs

Als het `Paint`-event optreedt worden alle methoden die er op zijn geabonneerd aangeroepen. Daarbij krijgen die via de twee parameters nadere details met betrekking op het event beschikbaar.

De parameters van de `Paint`-eventhandler

De eerste parameter van de eventhandler is het object dat het event heeft veroorzaakt. Dat is in dit geval niet zo belangrijk, en in de body van de methode gebruiken we deze parameter ook helemaal niet. Toch moet hij gedeclareerd worden, want als je je ergens op abonneert heb je je te conformeren aan de eisen die zo'n event aan zijn abonnees stelt.

De tweede parameter is wel zinvol. De naam van het type ervan, `PaintEventArgs`, geeft al aan dat het hier om extra informatie ('argumenten') behorend bij een paint-event gaat. We gebruiken hier de property `Graphics` van dat `PaintEventArgs`-object.

Die property `Graphics` is zelf een object van het type dat ook `Graphics` heet (verwarrend? handig juist?). Dat object kunnen we vervolgens onder handen nemen met de methode `DrawString`, waarmee je een tekst op het window kunt tekenen.

De parameters van `DrawString`

Bij de aanroep van de methode `DrawString` moeten we hem de parameters geven die hij nodig heeft. Er zijn een aantal varianten mogelijk, maar een daarvan heeft vijf parameters nodig: de tekst die getekend moet worden (eerste parameter), de positie op het scherm waar dat moet gebeuren (vierde en vijfde parameter), het lettertype (tweede parameter), en een zogeheten *brush* (derde parameter).

Voor het lettertype is een `Font`-object nodig. Dat construeren we ter plaatse met behulp van `new`, waarbij de constructor van `Font` de naam en de grootte van het gewenste lettertype meekrijgt.

Resteert nog het benodigde brush-object: dit is de 'kwast' waarmee de letters getekend worden. Meestal wordt een *solid brush* gebruikt, waarmee de tekst egaal gekeurd kan worden. Als alternatief (en dat is eigenlijk alleen voor hele grote letters zinvol) kun je een gearceerde, gestippelde of in kleur verlopende kwast gebruiken. We zouden hier met `new SolidBrush` een brush-object kunnen aanmaken, maar het is makkelijker om een kant-en-klare brush te gebruiken. Voor dit doel zijn er in de klasse `Brushes` een hele reeks statische properties beschikbaar, eentje voor elk van de standaardkleuren. Met `Brushes.Blue` pakken we de blauwe kwast.

Hoofdstuk 3

Methoden om te tekenen

3.1 De klasse Graphics

Grafische uitvoer

Een programma dat alleen maar "Hallo!" op het scherm schrijft is niet geweldig interessant. Bovendien konden we dat ook al door een `Label`-object aan het `Form` toe te voegen.

Gelukkig kennen objecten van type `Graphics` nog andere methoden dan `DrawString`. Door in de body van de event-handler van het `Paint`-event allerlei andere methoden aan te roepen, kunnen we een gecompliceerdere tekening opbouwen. Het programma in listing 6 bijvoorbeeld, maakt op deze manier een schilderij in de Stijl van Mondriaans "compositie met rood en blauw". In figuur 7 is dit programma in werking te zien.

blz. 31

Methoden van de klasse Graphics

Een event-handler van het `Paint`-event krijgt altijd als tweede parameter een `PaintEventArgs`-object mee. Dat object heeft een property `Graphics` die het type `Graphics` heeft. (Let op: het type van deze property heeft dus dezelfde naam als de property zelf; dat kan gebeuren, maar dat is lang niet altijd het geval).

Met dat object van type `Graphics` kunnen we nu methoden uit de klasse `Graphics` aanroepen. Daarbij worden als parameter nadere details over de positie en/of de afmeting van de te tekenen figuur meegegeven. Bovendien moeten we meegeven:

- voor teksten: een *font*-object die het lettertype specificeert
- voor opgevulde vlakken: een *brush*-object (kleur en opvulpatroon)
- voor lijn-tekeningen: een *pen*-object (kleur en lijndikte)

Enkele voorbeelden van methoden uit de klasse `Graphics` zijn:

- `DrawString(tekst, font, brush, x, y)` tekent een tekst
- `DrawLine(pen, x1, y1, x2, y2)` tekent een lijn tussen twee punten
- `DrawRectangle(pen, x, y, b, h)` tekent een rechthoek met breedte `b` en hoogte `h`
- `DrawEllipse(pen, x, y, b, h)` tekent een ellips binnen de genoemde rechthoek
- `FillRectangle(brush, x, y, b, h)` als `DrawRectangle`, maar nu ingekleurd
- `DrawImage(bitmap, x, y)` tekent een plaatje

Alle afmetingen en posities worden geteld in beeldscherm-punten, en worden gerekend vanaf de linkerbovenhoek. De *x*-coördinaat loopt dus van links naar rechts, de *y*-coördinaat loopt van boven naar beneden (en dat is dus anders dan in wiskunde-grafieken gebruikelijk is); zie figuur 8.

In listing 6 gebruiken we de methode `FillRectangle` om een aantal rechthoeken in te kleuren. Door de juiste kwast mee te geven worden sommige rechthoeken zwart, en andere gekleurd.

blz. 31

Klassen beschrijven de mogelijkheden van objecten

Alle methoden uit de klasse `Graphics` kun je aanroepen, als je tenminste de beschikking hebt over een object met object-type `Graphics`. Dat is in de body van de teken-methode geen probleem, want die methode heeft een `PaintEventArgs`-object als parameter, die op zijn beurt een `Graphics`-object als property heeft. Die kunnen we bij het tekenen dus gebruiken.

Dit illustreert de rol van klasse-definities. Het is niet zomaar een opsomming van methoden: de methoden kunnen gebruikt worden om een object uit die klasse te bewerken. In zekere zin beschrijft de lijst van methoden de mogelijkheden van een object: een `Graphics`-object "kan" teksten, lijnen, rechthoeken en ovaal tekenen.

Je kunt zien dat objecten "geheugen hebben". Ze hebben immers properties die je kunt opvragen, en soms ook kunt wijzigen. Als je een gewijzigde property later weer opvraagt, heeft het object

```
/* dit programma tekent een Mondriaan-achtige
 * "compositie met rood en blauw"
 */

5 using System.Windows.Forms;
using System.Drawing;

class Mondriaan : Form
{
10   Mondriaan()
   {
       this.Text = "Mondriaan";
       this.BackColor = Color.White;
       this.ClientSize = new Size(200, 100);
15       this.Paint += this.tekenScherm;
   }

   void tekenScherm(object obj, PaintEventArgs pea)
   {
20       int breedte, hoogte, balk, x1, x2, x3, y1, y2;

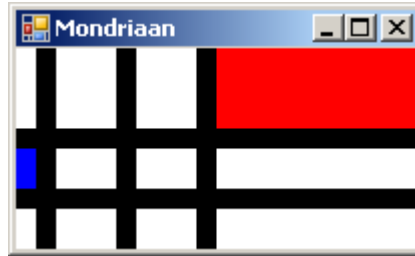
       breedte = pea.ClipRectangle.Width;
       hoogte = pea.ClipRectangle.Height;
       x1 = 10; x2 = 50; x3 = 90;
25       y1 = 40; y2 = 70;
       balk = 10;

       // zwarte balken
       pea.Graphics.FillRectangle(Brushes.Black, x1, 0, balk, hoogte);
30       pea.Graphics.FillRectangle(Brushes.Black, x2, 0, balk, hoogte);
       pea.Graphics.FillRectangle(Brushes.Black, x3, 0, balk, hoogte);
       pea.Graphics.FillRectangle(Brushes.Black, 0, y1, breedte, balk);
       pea.Graphics.FillRectangle(Brushes.Black, 0, y2, breedte, balk);

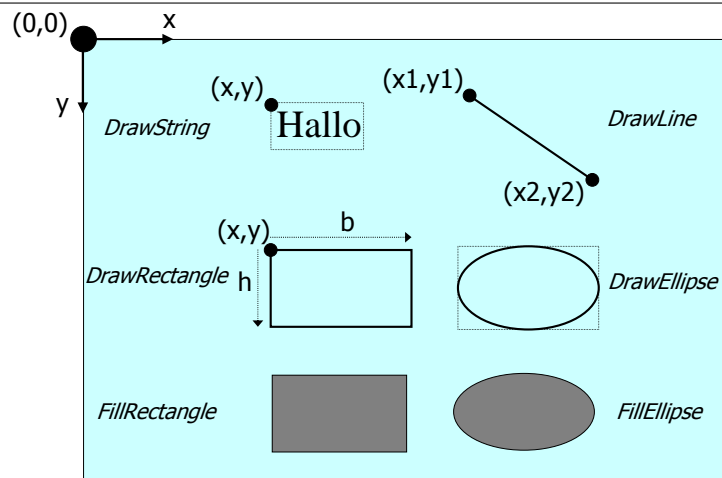
35       // gekleurde vlakken
       pea.Graphics.FillRectangle(Brushes.Blue, 0, y1 + balk, x1, y2 - (y1 + balk));
       pea.Graphics.FillRectangle(Brushes.Red, x3+balk, 0, breedte-(x3+balk), y1);
   }

40   static void Main()
   {
       Application.Run( new Mondriaan());
   }
}
```

Listing 6: Mondriaan/Mondriaan.cs



Figuur 7: Het programma Mondriaan in werking



Figuur 8: Enkele methoden uit de klasse Graphics

blijbaar onthouden wat de waarde van die property was. Dat klopt ook wel met de manier waarop in sectie 1.2 over objecten werd gesproken: een object is een groepje variabelen. Inmiddels hebben we gezien dat een klasse (sectie 1.2: groepje methoden met een naam) beschrijft wat je met zo'n object kunt doen. Het “gedrag” dat het object door aanroep van de methoden kan vertonen is veel interessanter dan een beschrijving van welke variabelen nou precies deel uitmaken van een object. Je ziet dit duidelijk aan de manier waarop we het **Graphics**-object gebruiken: uit welke variabelen het object precies is opgebouwd hoeven we helemaal niet te weten, als we maar weten welke methoden aangeroepen kunnen worden, en welke properties opgevraagd en/of veranderd. Het gebruik van bibliotheek-klassen gebeurt onder het motto: “vraag niet hoe het kan, maar profiteer ervan!”.

blz. 6
blz. 6

3.2 Variabelen

Variabele: declaratie, toekenning, gebruik

In eerdere programma's declareerden we een variabele, gaven die een waarde met een toekennings-opdracht, en gebruikten de variabele in latere opdrachten.

In sectie 2.4 was er een variabele om de ingetikte naam op te slaan, zodat we er later de lengte van konden bepalen:

blz. 19

```
string naam;
naam = Console.ReadLine();
Console.WriteLine( naam.Length + " letters");
```

In sectie 2.5 was er een variabele om een label-object in aan te maken, om er vervolgens properties van te veranderen en hem in beeld te brengen:

blz. 22

```
Label groet;
groet = new Label();
groet.Text = "Hallo";
this.Controls.Add(groet);
```

Het type van de variabele was in het eerste geval het ingebouwde type **string**, in het tweede geval de klasse **Label**.

Variabelen van type **int**

In het voorbeeldprogramma in listing 6 worden drie verticale zwarte balken getekend. Dat had gekund met de volgende opdrachten:

blz. 31

```
pea.Graphics.FillRectangle(Brushes.Black, 10, 0, 10, 100);
pea.Graphics.FillRectangle(Brushes.Black, 50, 0, 10, 100);
pea.Graphics.FillRectangle(Brushes.Black, 90, 0, 10, 100);
```

De eerste twee getallen geven de plaats aan van de balken: 10, 50 en 90 beeldpunten vanaf de linkerrand, tegen de bovenrand aan. De laatste twee getallen stellen de breedte (10) en hoogte (100) voor van de balken.

Nu zou het kunnen zijn dat we er na enig experimenteren achter komen dat het mooier is als de breedte van de balken niet 10, maar 12 is. Bij dat experimenteren moeten we dan in alle aanroepen de 10 vervangen door de 12. Het vervelende is, dat dat niet met zoek- en vervang-opdracht van de editor kan, want dan wordt ook de *x*-coördinaat van de eerste balk veranderd, en verandert de 100 in 120. (Wie denkt dat de drie getalletjes ook wel met de hand veranderd kunnen worden, stelle zich een experiment met Mondriaans ‘Victory Boogie Woogie’ voor).

Een oplossing is het gebruik van variabelen. In plaats van de constanten 10 en 100 gebruiken we twee variabelen, laten we zeggen **balk** en **hoogte**:

```
pea.Graphics.FillRectangle(Brushes.Black, 10, 0, balk, hoogte);
pea.Graphics.FillRectangle(Brushes.Black, 50, 0, balk, hoogte);
pea.Graphics.FillRectangle(Brushes.Black, 90, 0, balk, hoogte);
```

Voorafgaand aan deze opdrachten zorgen we er met een toekenningsopdracht voor dat deze variabelen een waarde hebben:

```
balk = 10;
hoogte = 100;
```

In dit geval bevatten de variabelen dus niet een tekst of een object, maar een getal. Zulke variabelen zijn van het type **int**, en moeten dus gedeclareerd worden met

```
int balk, hoogte;
```

Declaraties versus parameters

Declaraties van variabelen lijken veel op de parameters, die in de methode-header zijn opgesomd. In feite zijn dat óók declaraties. Maar er zijn een paar belangrijke verschillen:

- variabelen worden gedeclareerd in de body van de methode, parameters worden gedeclareerd tussen de haakjes in de methode-header;
- variabelen krijgen een waarde door een toekenning-opdracht, parameters krijgen automatisch een waarde bij de aanroep van de methode;
- in een variabele-declaratie kun je meerdere variabelen tegelijk declareren en het type maar één keer opschrijven, in parameter-declaraties moet bij elke parameter opnieuw het type worden opgeschreven (zelfs als dat hetzelfde is);
- variabele-declaraties eindigen met een puntkomma, parameter-declaraties niet.

Het type int

Variabelen (en parameters) met het type `int` zijn getallen. Hun waarde moet geheel zijn; er kunnen in `int`-waarden dus geen cijfers achter de komma staan. De waarde kan positief of negatief zijn. De grootst mogelijk `int`-waarde is 2147483647, en de kleinst mogelijke waarde is -2147483648; het bereik ligt dus ruwweg tussen min en plus twee miljard.

Net als `string` is `int` een ingebouwd type. Er zijn maar een handjevol ingebouwde typen. Andere ingebouwde typen die we nog zullen tegenkomen zijn `double` (getallen die wel cijfers achter de komma kunnen hebben), `char` (lettertekens) en `bool` (waarheidswaarden). De meeste andere typen zijn object-typen; hun mogelijkheden worden beschreven in een klasse.

Nut van declaraties

Declaraties zijn nuttig om meerdere redenen:

- de compiler weet door de declaraties van elke variabele wat het type is; daardoor kan de compiler controleren of methode-aanroepen wel zinvol zijn (aanroep van `FillRectangle` is zinvol met een `Graphics`-object onder handen, maar onmogelijk met waarden van andere object-typen of met `int`-waarden);
- de compiler kan bij aanroep van methoden controleren of de parameters wel van het juiste type zijn; zou je bijvoorbeeld bij aanroep van `DrawString` de tekst en de positie omwisselen, dan kan de compiler daarvoor waarschuwen;
- als je een tikfout maakt in de naam van een variabele (bijvoorbeeld `hootge` in plaats van `hoogte`), dan komt dat aan het licht doordat de compiler klaagt dat deze variabele niet is gedeclareerd.

3.3 Berekeningen

Expressies met een int-waarde

Op verschillende plaatsen in het programma kan het nodig zijn om een `int`-waarde op te schrijven, bijvoorbeeld:

- als parameter in een methode-aanroep van een methode met `int`-parameters
- aan de rechterkant van een toekenning-opdracht aan een `int`-variabele

Op deze plaatsen kun je een constante getalwaarde schrijven, zoals 37, of de naam van een `int`-variabele, zoals `hoogte`. Maar het is ook mogelijk om op deze plaats een formule te schrijven waarin bijvoorbeeld optelling en vermenigvuldiging een rol spelen, bijvoorbeeld `hoogte+5`. In dat geval wordt, op het moment dat de opdracht waarin de formule staat wordt uitgevoerd, de waarde uitgerekend (gebruikmakend van de op dat moment geldende waarden van variabelen). De uitkomst wordt gebruikt in de opdracht.

Zo'n formule wordt een *expressie* genoemd: het is een “uitdrukking” waarvan de waarde kan worden bepaald.

Gebruik van variabelen en expressies

In het voorbeeldprogramma in listing 6 komen variabelen en expressies goed van pas. Om het programma gemakkelijk aanpasbaar te maken, zijn er niet alleen variabelen gebruikt voor de breedte en hoogte van het schilderij en voor de breedte van de zwarte balken daarin, maar ook voor de positie van de zwarte balken. De x -posities van de drie verticale balken worden opgeslagen in de

drie variabelen `x1`, `x2` en `x3`, en de y -posities van de twee horizontale balken in de twee variabelen `y1` en `y2` (er mogen cijfers voorkomen in variabele-namen, als die maar met een letter begint). Met toekenningsopdrachten krijgen deze variabelen een waarde toegekend:

```
breedte = 200;
x1 = 10;
x2 = 50;
x3 = 90;
```

enzovoorts. Bij het tekenen van de balken komt er, behalve het getal 0, geen enkele constante meer aan te pas:

```
pea.Graphics.FillRectangle(Brushes.Black, x1, 0, balk, hoogte);
pea.Graphics.FillRectangle(Brushes.Black, 0, y1, breedte, balk);
```

Met behulp van expressies kunnen we ook de positie van de gekleurde vlakken in termen van deze variabelen aanduiden. Het blauwe vlak aan de linkerkant ligt direct onder de eerste zwarte balk; dit vlak heeft dus een y -coördinaat die één balkbreedte groter is dan de y -coördinaat van de eerste balk:

```
pea.Graphics.FillRectangle(Brushes.Blue, 0, y1+balk, iets , iets );
```

Omdat het vlak tegen de linkerkant aanligt, is de breedte van het vlak gelijk aan de x -positie van de eerste verticale balk. Zelfs de hoogte van het vlak kunnen we met een expressie aangeven: het is het verschil van de y -posities van de twee horizontale balken, minus één extra balkbreedte. Het vlak kan dus getekend worden met de methode-aanroep:

```
pea.Graphics.FillRectangle(Brushes.Blue, 0, y1+balk, x1, y2-(y1+balk) );
```

Ook het rode vlak tegen de bovenrand kan op zo'n manier beschreven worden.

Operatoren

In `int`-expressies kun je de volgende rekenkundige operatoren gebruiken:

- + optellen
- - aftrekken
- * vermenigvuldigen
- / delen
- % bepalen van de rest bij deling (uit te spreken als 'modulo')

Voor vermenigvuldigen wordt een sterretje gebruikt, omdat de in de wiskunde gebruikelijke tekens (\cdot of \times) nou eenmaal niet op het toetsenbord zitten. Helemaal weglaten van de operator, zoals in de wiskunde ook wel wordt gedaan is niet toegestaan, omdat dat verwarring zou geven met meer-letterige variabelen.

Bij gebruik van de delings-operator `/` wordt het resultaat afgerond, omdat het resultaat van een bewerking van twee `int`-waarden in `C#` weer een `int`-waarde oplevert. De afronding gebeurt door de cijfers achter de komma weg te laten; positieve waarden worden dus nooit "naar boven" afgerond (en negatieve waarden nooit "naar beneden"). De uitkomst van de expressie `14/3` is dus 4.

De bijzondere operator `%` geeft de rest die overblijft bij de deling. De uitkomst van `14%3` is bijvoorbeeld 2, en de uitkomst van `456%10` is 6. De uitkomst zal altijd liggen tussen 0 en de waarde rechts van de operator. De uitkomst is 0 als de deling precies op gaat.

Prioriteit van operatoren

Als er in één expressie meerdere operatoren voorkomen, dan geldt de gebruikelijke prioriteit van de operatoren: "vermenigvuldigen gaat voor optellen". De uitkomst van `1+2*3` is dus 7, en niet 9. Optellen en aftrekken hebben onderling dezelfde prioriteit, en vermenigvuldigen en de twee delings-operatoren ook.

Komen in een expressie operatoren van dezelfde prioriteit naast elkaar voor, dan wordt de expressie van links naar rechts uitgerekend. De uitkomst van `10-5-2` is dus 3, en niet 7.

Als je wilt afwijken van deze twee prioriteitsregels, dan kun je haakjes gebruiken in een expressie, zoals in `(1+2)*3` en `3+(6-5)`. In de praktijk komen in dit soort expressies natuurlijk variabelen voor, anders had je de waarde (9 en 4) meteen zelf wel kunnen uitrekenen.

Een overbodig extra paar haakjes is niet verboden: `1+(2*3)`, en wat de compiler betreft mag je naar hartelust overdrijven: `((1)+(((2)*3)))`. Dat laatste maakt het programma er voor de menselijke lezer echter niet duidelijker op.

Expressie: programmafragment met een waarde

Een expressie is een stukje programma waarvan de *waarde* kan worden bepaald. Bij expressies waar getallen en operatoren in voorkomen is dat een duidelijke zaak: de waarde van de expressie `2+3` is 5. Er kunnen ook variabelen in een expressie voorkomen, en dan wordt bij het bepalen van de waarde de op dat moment geldende waarde van de variabelen gebruikt. De waarde van de expressie `y1+balk` is 50, als eerder met toekenningsoopdrachten de variabele `y1` de waarde 40 en de variabele `balk` de waarde 10 heeft gekregen.

Het opvragen van een property van een object is ook een expressie: een property heeft immers een waarde. Het programmafragment `naam.Length` is een expressie, en kan (afhankelijk van de waarde van `naam`) bijvoorbeeld de waarde 6 hebben.

Expressies met een string-waarde

Het begrip ‘waarde’ van een expressie is niet beperkt tot getal-waarden. Ook een tekst, oftewel een string, geldt als een waarde. Er zijn constante strings, zoals `"Hallo"`, en je kunt strings opslaan in een variabele. Later gebruik van zo’n variabele in een expressie geeft dan weer de opgeslagen string. Ook kun je strings gebruiken in operator-expressies, zoals `"Hallo "+naam`. ‘Optellen’ is hier niet het juiste woord; de `+`-operator op strings betekent veeleer ‘samenvoegen’. Niet alle operatoren kun je op waarden van alle types gebruiken: tussen twee `int`-waarden kun je onder meer de operator `+` of `*` gebruiken, maar tussen twee `string`-expressies alleen de operator `+`.

Sommige properties hebben een string als waarde, bijvoorbeeld `f.Text` als `f` een `Form` is. Dus ook hier vormt het opvragen van een property een expressie.

Expressies met een object-waarde

Het begrip ‘waarde’ van een expressie is niet beperkt tot getal- en string-waarden. Expressies kunnen van elk type zijn waarvan je ook variabelen kunt declareren, dus naast de ingebouwde typen `int` en `string` kunnen dat ook object-typen zijn, zoals `Color`, `Form`, of `Pen`.

Weliswaar zijn er geen constanten met een object-waarde, maar een variabele of een property blijft een expressie met als waarde een object. Een derde expressievorm met een object-waarde is de constructie van een nieuw object met `new`. De expressie `new Form()` heeft een `Form`-object als waarde, de expressie `new Font("Tahoma",30)` heeft een `Font`-object als waarde.

Syntax van expressies

De syntax van expressies tot nu toe wordt samengevat in het syntax-diagram in figuur 9. Er is speciale syntax voor een constant getal en een constante string (tussen aanhalingstekens). Een losse variabele is ook een geldige expressie: een variabele heeft immers een waarde.

Uit expressies kun je weer grotere expressies bouwen: twee expressies met een operator ertussen vormt in zijn geheel weer een expressie, en een expressie met een paar haakjes eromheen ook.

Verder zijn er in het syntax-diagram aparte routes voor de expressie-vorm waarin een nieuw object wordt geconstrueerd met `new`, voor de aanroep van een methode, en voor het opvragen van een property.

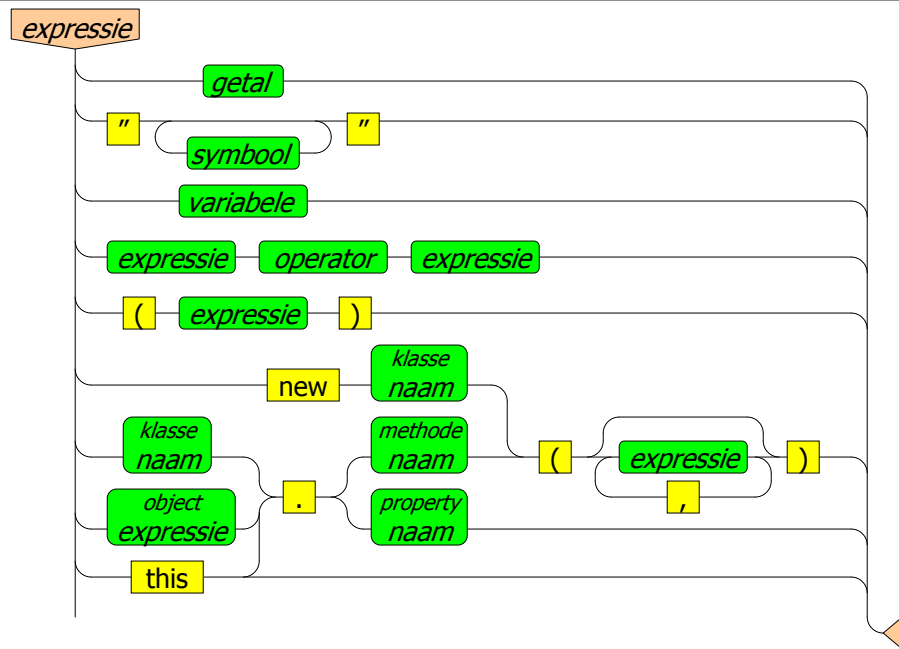
Voor de punt van een methode-aanroep of opvragen van een property kan een klasse-naam staan (als het om een statische methode of property gaat), of een object (als de methode of property een object onder handen neemt). In het syntax-diagram kun je zien dat er in het niet-statische geval in feite een *expressie* voor de punt staat.

In veel gevallen is de expressie voor de punt simpelweg een variabele (zoals in de property `naam.Length`), maar het is ook mogelijk om er een constante te gebruiken (zoals in `"Hallo".Length`) of een property van een ander object (zoals in `scherm.Text.Length`).

Soms staat er voor de punt het keyword `this`. Dit speciale object heeft als waarde het object dat de methode onder handen heeft, en dat kan natuurlijk ook gebruikt worden voor het opvragen van properties of het aanroepen van methoden. Omdat `this` een waarde heeft, vormt het zelf een volwaardige expressie. Meestal zul je die expressie aantreffen links van een punt in een grotere expressie, maar `this` kan ook op andere plaatsen staan waar een (object-)waarde nodig is.

Expressies versus opdrachten

De syntactische begrippen ‘expressie’ en ‘opdracht’ hebben allebei een groot syntax-diagram; van allebei zijn er een tiental verschillende vormen (die we nog niet allemaal hebben gezien). Houd deze twee begrippen goed uit elkaar: het zijn verschillende dingen. Dit is het belangrijkste verschil:



Figuur 9: (Vereenvoudigde) syntax van een expressie

een *expressie* kun je *uitrekenen* (en heeft dan een waarde)

een *opdracht* kun je *uitvoeren* (en heeft dan een effect)

Het zijn uiteindelijk de opdrachten die (samen met declaraties) in de body van een methode staan. Losse expressies kunnen niet in een methode staan. Expressies kunnen wel een deel uitmaken van een opdracht:

- er staat een expressie rechts van het `=`-teken in een toekenningsopdracht;
- er staan expressies tussen de haakjes van een methode-aanroep;
- er staat een expressie voor de punt van een (niet-statische) methode-aanroep en property-bepaling.

Als je het syntaxdiagram van 'expressie' vergelijkt met dat van 'opdracht' dan valt het op dat in beide schema's de methode-aanroep voorkomt, met als enige verschil dat er bij een opdracht nog een puntkomma achter staat.

Een voorbeeld van een methode-aanroep die een opdracht vormt is `Console.WriteLine("Hoi");`. Een methode-aanroep die een expressie vormt is `Console.ReadLine()`. In dit geval staat er dus geen puntkomma achter! Deze expressie moet deel uitmaken van een groter geheel, bijvoorbeeld als rechterkant van een toekenningsopdracht:

```
naam = Console.ReadLine();
```

Nu staat er wel een puntkomma achter, maar dat is niet vanwege de methode-aanroep, maar omdat de toekenningsopdracht moet eindigen met een puntkomma.

Of een methode bedoeld is om aan te roepen als opdracht of als expressie, wordt bepaald door de auteur van de methode. Bij `ReadLine` is het duidelijk de bedoeling dat de methode-aanroep een string als waarde heeft, en deze aanroep is dan ook een expressie. De methode `WriteLine` heeft geen waarde, en de aanroep vormt dan ook een opdracht. Het verschil wordt door de auteur van de methode in de header aangegeven: staat er aan het begin van de header een type, dan is dat het type van de waarde van de aanroep; staat er in plaats van het type het woord `void`, dan heeft de aanroep geen waarde.

Void-methodes moeten dus altijd als opdracht worden aangeroepen. Alle andere methode worden meestal als expressie aangeroepen. Als je wilt kun je ze toch als opdracht aanroepen; de waarde van de methode wordt dan genegeerd. Dit hebben we in sectie 2.2 gedaan bij de aanroep van `Console.ReadLine`.

3.4 Programma-layout

Commentaar

Voor de menselijke lezer van een programma (een collega-programmeur, of jijzelf over een paar maanden, als je de details van de werking van het programma vergeten bent) is het heel nuttig als er wat toelichting bij het programma staat geschreven. Dit zogenaamde *commentaar* wordt door de compiler geheel genegeerd, maar zorgt ervoor dat het programma beter te begrijpen is.

Er zijn in C# twee manieren om commentaar te markeren:

- alles tussen de tekencombinatie `/*` en de eerstvolgende teken-combinatie `*/` (mogelijk pas een paar regels verderop)
- alles tussen de tekencombinatie `//` en het einde van de regel

Dingen waarbij het zinvol is om commentaar te zetten zijn: groepjes opdrachten die bij elkaar horen, methoden en de betekenis van de parameters daarvan, en complete klassen.

Het is de kunst om in het commentaar niet de opdracht nog eens in woorden weer te geven; je mag er van uitgaan dat de lezer C# kent. In het voorbeeld-programma staat daarom bijvoorbeeld het commentaar

```
// posities van de lijnen
x1 = 10; x2 = 50;
```

en niet

```
// maak de variabele x1 gelijk aan 10, en x2 aan 50
x1 = 10; x2 = 50;
```

Tijdens het testen van het programma kunnen de commentaar-tekenen ook gebruikt worden om een of meerdere opdrachten tijdelijk uit te schakelen. Het staat echter niet zo verzorgd om dat soort “uitgecommentarieerde” opdrachten in het definitieve programma te laten staan.

Regel-indeling

Er zijn geen voorschriften voor de verdeling van de tekst van een C#-programma over de regels van de file. Hoewel het gebruikelijk is om elke opdracht op een aparte regel te schrijven, worden hier door de compiler geen eisen aan gesteld. Als dat de overzichtelijkheid van het programma ten goede komt, kan een programmeur dus meerdere opdrachten op één regel schrijven (in het voorbeeldprogramma is dat gedaan met de relatief korte toekenningsopdrachten). Bij hele lange opdrachten (bijvoorbeeld methode-aanroepen met veel of ingewikkelde parameters) is het een goed idee om de tekst over meerdere regels te verspreiden.

Verder is het een goed idee om af en toe een regel over te slaan: tussen verschillende methoden, en tussen groepjes opdrachten (en het bijbehorende commentaar) die bij elkaar horen.

Witruimte

Ook voor de plaatsing van spaties zijn er nauwelijks voorschriften. De enige plaats waar spaties vanzelfsprekend werkelijk van belang zijn, is tussen afzonderlijke woorden: `static void Main` mag niet worden geschreven als `staticvoidMain`. Omgekeerd, midden in een woord mag geen extra spatie worden toegevoegd.

In een tekst die letterlijk genomen wordt omdat er aanhalingstekens omheen staan, worden ook de spaties letterlijk genomen. Er is dus een verschil tussen

```
Console.WriteLine("hallo");
```

en

```
Console.WriteLine("h a l l o ");
```

Maar voor het overige zijn extra spaties overal toegestaan, zonder dat dat de betekenis van het programma verandert.

Goede plaatsen om extra spaties te schrijven zijn:

- achter elke komma en puntkomma (maar niet ervoor)
- links en rechts van het `=` teken in een toekenningsopdracht
- aan het begin van regels, zodat de body van methoden en klassen wordt ingesprongen (4 posities is gebruikelijk) ten opzichte van de accolades die de body begrenzen.

Declaratie op deze manier is echter niet aan te raden: expliciete vermelding van het type maakt het programma duidelijker voor de menselijke lezer, en maakt het de compiler mogelijk om foutmeldingen te geven in het geval dat het bedoelde type niet klopt met de initialisatie.

3.6 Methode-definities

Orde in de chaos

Als je een vierkant tekent met twee schuine lijntjes erbovenop heb je een simpel huisje getekend. Het voorbeeldprogramma in dit hoofdstuk tekent drie huisjes. Het complete programma staat in listing 7; in figuur 10 is het resultaat te zien.

blz. 41

Deze drie huisjes zouden getekend kunnen worden met de volgende methode als `Paint`-eventhandler:

```
void tekenScherm(object obj, PaintEventArgs pea)
{
    Pen pen = Pens.Black;
    // kleine huisje links
    pea.Graphics.DrawRectangle(pen, 20, 60, 40, 40);
    pea.Graphics.DrawLine(pen, 14, 66, 40, 40);
    pea.Graphics.DrawLine(pen, 40, 40, 66, 66);
    // kleine huisje midden
    pea.Graphics.DrawRectangle(pen, 80, 60, 40, 40);
    pea.Graphics.DrawLine(pen, 74, 66, 100, 40);
    pea.Graphics.DrawLine(pen, 100, 40, 126, 66);
    // grote huis rechts
    pea.Graphics.DrawRectangle(pen, 140, 40, 60, 60);
    pea.Graphics.DrawLine(pen, 130, 70, 170, 10);
    pea.Graphics.DrawLine(pen, 170, 10, 210, 66);
}
```

Ondanks het commentaar begint dit nogal onoverzichtelijk te worden. Wat zou je bijvoorbeeld in dit programma moeten veranderen als bij nader inzien niet het rechter, maar juist het linker huis groot getekend moet worden? Om het programma op die manier aan te passen zou je alle parameters van alle opdrachten weer moeten napuzzelen, en als je dat niet nauwkeurig doet loop je een goede kans dat in de nieuwe versie van het programma een van de daken in de lucht getekend wordt.

En dan is dit nog maar een programma dat drie huisjes tekent; dit programma uitbreiden zodat het niet drie maar tien huisjes tekent is ronduit vervelend.

We gaan wat orde scheppen in deze chaos met behulp van methoden.

Nieuwe methoden

Methoden zijn bedoeld om groepjes opdrachten die bij elkaar horen als één geheel te kunnen behandelen. Op het moment dat het groepje opdrachten moet worden uitgevoerd, kun je de dat laten gebeuren door de methode aan te roepen.

In het voorbeeld horen duidelijk steeds drie opdrachten bij elkaar die samen één huisje tekenen (de aanroep van `DrawRectangle` en de twee aanroepen van `DrawLine`). Die drie opdrachten zijn dus een goede kandidaat om in een methode te zetten; in de methode `tekenScherm` komen dan alleen nog maar drie aanroepen van deze nieuwe methode te staan. De opzet van het programma wordt dus als volgt:

```
public class Huizen : Form
{
    Huizen()
    {
        this.Paint += tekenScherm;
    }
    private void tekenHuis( iets )
    {
        iets .DrawRectangle( iets );
        iets .DrawLine( iets );
        iets .DrawLine( iets );
    }
    public void tekenScherm(object obj, PaintEventArgs pea)
```

```
// dit programma tekent drie huizen van divers formaat

using System.Windows.Forms;
using System.Drawing;
5 using System.Drawing.Drawing2D;

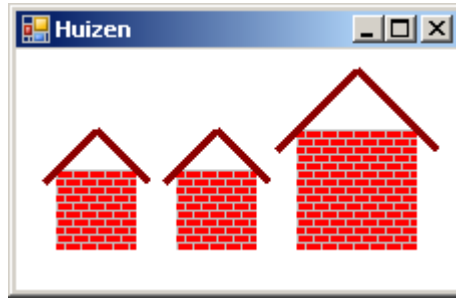
class Huizen : Form
{
    Huizen()
10 {
        this.Text = "Huizen";
        this.BackColor = Color.White;
        this.ClientSize = new Size(220, 120);
        this.Paint += this.tekenScherm;
15 }

    void tekenScherm(object obj, PaintEventArgs pea)
    {
        this.tekenHuis(pea.Graphics, 20, 100, 40);
20 this.tekenHuis(pea.Graphics, 80, 100, 40);
        this.tekenHuis(pea.Graphics, 140, 100, 60);
    }

    void tekenHuis(Graphics gr, int x, int y, int breedte)
25 {
        int topx, topy, afdak;
        topx = x + breedte / 2;
        topy = y - 3 * breedte / 2;
        afdak = breedte / 6;
30 Brush br = new HatchBrush(HatchStyle.HorizontalBrick, Color.Silver, Color.Red);
        gr.FillRectangle(br, x, y - breedte, breedte, breedte);
        Pen pen = new Pen(Color.DarkRed, 3);
        gr.DrawLine(pen, x - afdak, y - breedte + afdak, topx, topy);
        gr.DrawLine(pen, topx, topy, x + breedte+afdak, y - breedte+afdak);
35 }

    static void Main()
    {
        Application.Run(new Huizen());
40 }
}
```

Listing 7: Huizen/Huizen.cs



Figuur 10: Het programma Huizen in werking

```

    {   iets .tekenHuis( iets );
        iets .tekenHuis( iets );
        iets .tekenHuis( iets );
    }
}

```

Er zijn dus twee methoden: naast de event-handler `tekenScherm` is er een tweede methode die één huis tekent, en die we daarom `tekenHuis` noemen. De naam mag vrij worden gekozen, en het is een goed idee om die naam de taak van de methode te laten beschrijven.

De volgorde waarin de methoden in de klasse staan is niet van belang. De opdrachten in de body van een methode worden pas uitgevoerd als de methode wordt aangeroepen. De constructormethode `Huizen` wordt aangeroepen als er (in de methode `Main`) een nieuw `Huizen`-object wordt gemaakt. De methode `tekenScherm` wordt aangeroepen als het `Paint`-event optreedt. Pas als de methode `tekenScherm` een aanroep doet van de methode `tekenHuis`, worden de opdrachten in de body van de methode `tekenHuis` uitgevoerd. Als dat klaar is, gaat `tekenScherm` weer verder met de volgende opdracht. In dit geval is dat toevallig weer een aanroep van `tekenHuis`, dus wordt er een tweede huis getekend. Ook bij de derde aanroep in `tekenScherm` wordt er een huis getekend, en pas daarna gaat het weer verder op de plaats van waaruit `tekenScherm` zelf werd aangeroepen.

Methoden nemen een object onder handen

De opzet van het programma is nu klaar, maar er zijn nog de nodige details die ingevuld moeten worden (in de opzet aangegeven met *iets*). Als eerste bekijken we de vraag: wat komt er vóór de punt te staan bij de aanroep van de methode `DrawRectangle` in de body van `tekenHuis`?

Elke methode die je aanroept, krijgt een object “onder handen”; dit is het object dat je voor de punt in de methode-aanroep aangeeft. De methode `DrawRectangle` bijvoorbeeld, krijgt een `Graphics`-object onder handen.

Tot nu toe hebben we daar het `Graphics`-object voor gebruikt, dat we als property kunnen opvragen van de `PaintEventArgs` parameter van `tekenScherm`. De parameter van de methode `tekenScherm` is echter niet zomaar beschikbaar in de body van de methode `tekenHuis`.

Parameters van methoden

We moeten er dus voor zorgen dat ook in de body van `tekenHuis` een `Graphics`-object beschikbaar is, en dat kunnen we doen door `tekenHuis` een `Graphics`-object als parameter te geven. In de body van `tekenHuis` kunnen we die parameter dan mooi gebruiken voor de punt in de aanroep van `DrawRectangle` en `DrawLine`:

```

private void tekenHuis(Graphics gr, iets )
{   gr.DrawRectangle( iets );
    gr.DrawLine( iets );
    gr.DrawLine( iets );
}

```

Je mag als programmeur de naam van de parameter vrij kiezen; hier hebben we de naam `gr` gekozen. In de body van de methode moet je, als je de parameter wilt gebruiken, wel diezelfde naam gebruiken, dus bij de aanroep van methode `DrawRectangle` schrijven we nu het `Graphics`-object `gr`.

De naam van het *type* van de parameter mag je niet zomaar kiezen: het object-type **Graphics** is een bestaande bibliotheek-klasse, en die mogen we niet ineens **Grafiek** of iets dergelijks gaan noemen.

Nu we in de header van de methode **tekenHuis** gespecificeerd hebben dat de eerste parameter een **Graphics**-object is, moeten we er voor zorgen dat bij aanroep van **tekenHuis** ook inderdaad een **Graphics**-object wordt meegegeven. De aanroep van **tekenHuis** vindt plaats vanuit de methode **tekenScherm**, en daar hebben we gelukkig een **Graphics**-object beschikbaar: de property die ook **Graphics** heet die we kunnen opvragen van de **PaintEventArgs** parameter van **tekenScherm**. De aanroepen van **tekenHuis** komen er dus als volgt uit te zien:

```
void tekenScherm(object obj, PaintEventArgs pea)
{
    iets.tekenHuis(pea.Graphics, iets );
    iets.tekenHuis(pea.Graphics, iets );
    iets.tekenHuis(pea.Graphics, iets );
}
```

De methode **tekenHuis** wordt alleen maar door **tekenScherm** aangeroepen, en is niet bedoeld om van buiten de klasse te worden aangeroepen (althans niet direct). De methode **tekenHuis** is daarom als een **private** methode gedeclareerd: hij is alleen voor intern gebruik door andere methoden van de klasse.

Het object **this**

Een volgend detail dat we nog moeten invullen in het programma is het object vóór de punt bij de aanroep van **tekenHuis**. Welk object krijgt **tekenHuis** eigenlijk onder handen? En welk object heeft **tekenScherm** zelf eigenlijk onder handen?

Het object dat door methoden onder handen wordt genomen, is van het object-type zoals dat in de klasse-header staat waarin de methode staat. De methode **DrawRectangle** heeft een **Graphics**-object onder handen, omdat **DrawRectangle** in de klasse **Graphics** staat.

Welnu, de methoden **tekenScherm** en **tekenHuis** staan in de klasse **Huizen**, en hebben dus blijkbaar een **Huizen**-object onder handen. Zo'n **Huizen**-object is in **Main** gecreëerd, en de methode **tekenScherm** heeft dat object onder handen. In de body van **tekenScherm** zouden we datzelfde object wel willen gebruiken om door **tekenHuis** onder handen genomen te laten worden. Maar hoe moeten we “het” object dat we onder handen hebben, aanduiden? Dit object is immers geen parameter, dus we hebben het in de methode-header geen naam kunnen geven.

De oplossing van dit probleem is dat in **C#** het object dat een methode onder handen heeft gekregen, kan worden aangeduid met het woord **this**. Dit woord kan dus worden geschreven op elke plaats waar “het” object nodig is. Nu komt het dus goed van pas om in de body van de methode **tekenScherm** aan te geven dat bij de aanroep van **tekenHuis** hetzelfde object onder handen genomen moet worden als dat **tekenScherm** zelf al onder handen heeft:

```
void tekenScherm(object obj, PaintEventArgs pea)
{
    this.tekenHuis(pea.Graphics, iets );
    this.tekenHuis(pea.Graphics, iets );
    this.tekenHuis(pea.Graphics, iets );
}
```

Het woord **this** is in **C#** een voor dit speciale doel gereserveerd woord (net als **class**, **void**, **public** en dergelijke). Je mag het dus niet gebruiken als naam van een variabele of iets dergelijks. In elke methode duidt **this** een object aan. Dit object heeft als object-type dat wat in de header van de klasse staat waarin de methode is gedefinieerd.

3.7 Op zoek naar parameters

Parameters maken methoden flexibeler

Het administratieve werk –zorgen dat alle methoden over de benodigde **Graphics**- en **Huizen**-objecten kunnen beschikken– is nu gedaan, en het leuke werk kan beginnen: de jacht op de overige parameters.

Tot nu toe hebben we voor het gemak gezegd dat de huis-tekenende opdrachten (**DrawRectangle** en tweemaal **DrawLine**) in alle drie gevallen hetzelfde is, en dat ze daarom met drie aanroepen van **tekenHuis** kunnen worden uitgevoerd. Maar de opdrachten die de drie huizen tekenen zijn niet precies hetzelfde: per huisje verschillen de getallen die als parameter worden meegegeven aan

`DrawRectangle` en `DrawLine`.

We kijken eerst maar eens naar de aanroepen van `DrawRectangle` in de oorspronkelijke (chaotische) versie van het programma:

```
pea.Graphics.DrawRectangle(pen, 20, 60, 40, 40);
pea.Graphics.DrawRectangle(pen, 80, 60, 40, 40);
pea.Graphics.DrawRectangle(pen, 140, 40, 60, 60);
```

De eerste twee getallen zijn de coördinaten van de linkerbovenhoek van de rechthoek, de laatste twee getallen zijn de breedte en hoogte van de rechthoek. Omdat we vierkanten tekenen zijn de breedte en hoogte gelijk: 40 voor de kleine huisjes, en 60 voor het grote.

De breedte (tevens hoogte) is niet in alle gevallen dezelfde. Als we de gewenste breedte echter door een parameter aangeven, dan kunnen we bij elke aanroep een andere breedte specificeren.

Wat betreft de coördinaten geldt hetzelfde: aangezien deze verschillend zijn bij alle drie de aanroepen, laten we de aanroeper van `tekenHuis` ook deze waarden specificeren. Voor de aanroeper is het waarschijnlijk gemakkelijker om de coördinaten van de linker-onderhoek te specificeren: de coördinaten van de bovenhoek zijn verschillend voor huizen van verschillende grootte, terwijl de *y*-coördinaat van de onderhoek voor huizen op één rij hetzelfde zijn. Ook dit kan geregeld worden: we spreken af dat de *y*-coördinaat-parameter van de methode `tekenHuis` de basislijn van de huizen voorstelt, en de *y*-coördinaat van de bovenhoek, zoals `DrawRectangle` die nodig heeft, berekenen we met een expressie:

```
private void tekenHuis(Graphics gr, int x, int y, int br)
{
    Pen pen = Pens.Black;
    gr.DrawRectangle(pen, x, y-br, br, br);
    gr.DrawLine(pen, iets);
    gr.DrawLine(pen, iets);
}

public void tekenScherm(object obj, PaintEventArgs pea)
{
    this.tekenHuis(pea.Graphics, 20, 100, 40);
    this.tekenHuis(pea.Graphics, 80, 100, 40);
    this.tekenHuis(pea.Graphics, 140, 100, 60);
}
```

De parameters van de twee aanroepen van `DrawLine` (de coördinaten van begin- en eindpunt van de lijnen die het dak van het huis vormen) zijn ook in alle gevallen verschillend. Het is echter niet nodig om die apart als parameter aan `tekenHuis` mee te geven; deze coördinaten kunnen namelijk worden berekend uit de positie en de breedte van het vierkant, en die hebben we al als parameter. De coördinaten van de top van het dak zijn twee maal nodig: als het eindpunt van de eerste lijn, en als beginpunt van de tweede. Om de berekening van dit punt niet twee maal te hoeven doen, gebruiken we twee variabelen om deze coördinaten tijdelijk op te slaan. Deze variabelen zijn nodig in de methode `tekenHuis`, en worden dan ook lokaal in die methode gedeclareerd:

```
private void tekenHuis(Graphics gr, int x, int y, int br)
{
    int topx, topy;
    topx = x + br/2;
    topy = y - 3*br / 2;
    Pen pen = Pens.Black;
    gr.DrawRectangle(pen, x, y-br, br, br);
    gr.DrawLine(pen, x, y-br, topx, topy);
    gr.DrawLine(pen, topx, topy, x+br, y-br);
}
```

In de expressie $3*br/2$ zijn alle betrokken getallen een `int`: de constanten 3 en 2 omdat er geen punt of *E* in voorkomt, en `br` omdat die als `int` is gedeclareerd. Dat betekent dat de berekening ook een `int` oplevert, en dat het resultaat van de deling dus (naar beneden) wordt afgerond.

De prioriteit van vermenigvuldigen en delen is dezelfde, en dus wordt $3*br/2$ van links naar rechts uitgerekend: eerst $3*br$, en dan de uitkomst halveren. Als we hadden geschreven $3/2*br$ dan gebeuren er nare dingen: de berekening $3/2$ wordt uitgevoerd *en afgerond*. De uitkomst is dus niet anderhalf maar 1, en dat wordt vervolgens vermenigvuldigd met `br`. Dat is natuurlijk niet de bedoeling! Let dus op bij werken met `int`-waarden in dit soort situaties: zorg dat je eerst vermenigvuldigt, en dan pas deelt.

Om het helemaal mooi te maken, hebben we in listing 7 ook nog een variabele gedeclareerd die bepaalt hoe ver het dak uitsteekt naast het huis. Deze variabele `afdak` is afhankelijk van de breedte van het huis: een groter huis krijgt ook een groter afdak.

Ook wordt in de listing niet `DrawRectangle` gebruikt maar `FillRectangle`, met een voor huizen wel zeer toepasselijke brush.

blz. 41

Grenzen aan de flexibiliteit

Nu we besloten hebben om de linkeronderhoek van het huisje te specificeren (en niet de linkerbovenhoek van de gevel), blijkt de y -coördinaat in alle drie de aanroepen van `tekenHuis` hetzelfde te zijn (namelijk 100). Achteraf gezien was deze parameter dus niet nodig geweest: we hadden de waarde 100 in de body van `tekenHuis` kunnen schrijven op alle plaatsen waar nu een y staat.

Kwaad kan het echter ook niet om “te veel” parameters te gebruiken. Wie weet willen we later nog wel eens huisjes tekenen op een andere y -coördinaat dan 100, en dan is onze methode er alvast maar op voorbereid.

De vraag is wel hoe ver je moet gaan in het flexibeler maken van methoden, door het toevoegen van extra parameters. De methode `tekenHuis` zoals we die nu hebben geschreven kan alleen maar huisjes met een vierkante gevel tekenen. Het is ook denkbaar om de breedte en de hoogte apart als parameter mee te geven, want wie weet willen we later nog wel eens een niet-vierkant huisje tekenen, en dan is de methode er alvast maar op voorbereid. En je zou de hoogte van het dak apart als parameter mee kunnen geven, want wie weet willen we later nog wel eens een huisje met een extra schuin of extra plat dak tekenen. En je zou nog een brush-object apart als parameter kunnen meegeven, want wie weet willen we later nog wel eens een huisje met een ander patroon tekenen. En dan nog een pen-object, zodat het dak een andere kleur kan krijgen dan de gevel... Al die extra parameters hebben wel een prijs, want bij de aanroep moeten ze steeds maar meegegeven worden. En als de aanroeper helemaal niet van plan is om al die variatie te gaan gebruiken, zijn die overbodige parameters maar tot last.

De kunst is om een afweging te maken tussen de moeite die het kost om extra parameters te gebruiken (zowel voor de programmeur van de methode als voor de programmeur die de aanroepen schrijft) en de kans dat de extra flexibiliteit in de toekomst ooit nodig zal zijn.

Flexibiliteit in het groot

Hetzelfde dilemma doet zich voor bij programma's als geheel. Gebruikers willen graag flexibele software, die ze naar hun eigen wensen kunnen configureren. Maar ze zijn weer ontevreden als ze eindeloze lijsten met opties moeten instellen voordat ze aan het werk kunnen, en onnodige opties maken een programma maar complex en (daardoor) duur.

Achteraf heb je makkelijk praten, maar had men in het verleden kunnen voorzien dat er ooit behoefte zou ontstaan aan een 4-cijferig jaartal in plaats van een 2-cijferig? (Ja.) Maar moeten we er nu al rekening mee houden dat in de toekomst de jaarkalender misschien een dertiende maand krijgt, en alle maanden 28 dagen? (Nou, nee). Moet de gebruiker van financiële software zelf kunnen instellen wat het geldende BTW-tarief is? Of moet de gebruiker, als het tarief ooit zal veranderen, maar een nieuwe versie van het programma kopen? En moet de software er nu al in voorzien dat er behalve een laag en een hoog BTW-tarief ook een midden-tarief komt? En dat de munteenheid verandert? En het symbool daarvoor? Moet de gebruiker van een programma waarin tijden een rol spelen zelf kunnen instellen op welke datum de zomertijd eindigt? Of is het beter als de regel daarvoor (“laatste zondag van oktober”) in het programma is ingebouwd? En als de regel dan veranderd wordt? Of moet de gebruiker zelf de regel kunnen specificeren? En mag hij dan eerst kiezen in welke taal hij “oktober” mag spellen?

Hoofdstuk 4

Objecten en methoden

4.1 Variabelen

Declaratie: aangifte van het type van een variabele

blz. 19

In sectie 2.4 hebben we gezien dat je in je programma wilt gebruiken moet declareren. Dat gebeurt door middel van een zogeheten *declaratie*, waarin je de namen van de variabelen opsomt en hun type aangeeft. Een voorbeeld van een declaratie is

```
int x, y;
```

Je maakt daarmee ruimte in het geheugen voor twee variabelen, genaamd **x** en **y**, en geeft aan dat het type daarvan **int** is. Het type **int** staat voor *integer number*, oftewel geheel getal. Je kunt je de situatie in het geheugen als volgt voorstellen:

x  **y** 

De geheugenplaatsen zijn beschikbaar (in de tekening gesymboliseerd door het hok), maar ze hebben nog geen waarde. Een variabele krijgt een waarde door middel van een toekenningso opdracht, zoals

```
x = 20;
```


De situatie in het geheugen wordt daarmee:

x  **y** 

Met een tweede toekenningso opdracht kan ook aan de variabele **y** een waarde worden toegekend. In de expressie aan de rechterkant van het =-teken kan de variabele **x** worden gebruikt, omdat die inmiddels een waarde heeft. Bijvoorbeeld:

```
y = x+5;
```

Na het uitvoeren van deze opdracht is de situatie als volgt:

x  **y** 

Het kan gebeuren dat later een andere waarde aan een variabele wordt toegekend, bijvoorbeeld met

```
x = y*2;
```

De variabele **x** krijgt daarmee een nieuwe waarde, en de oude waarde gaat voor altijd verloren. De situatie die daardoor ontstaat is als volgt:

x  **y** 

Merk op dat met toekenningso opdrachten de waarde van een variabele kan veranderen. De naam wordt echter met de declaratie definitief vastgelegd. Om te zien wat er in ingewikkelde situaties gebeurt, kun je de situatie op papier ‘naspelen’. Teken daartoe voor elke declaratie met pen een hok met bijbehorende naam. De toekenningso opdrachten voer je uit door het hok van de variabelen met potlood in te vullen, waarbij je een eventuele oude inhoud van het hok eerst uitgumt.

Numerieke typen

Een ander numeriek type is het type `double`. Variabelen van dat type kunnen getallen met cijfers achter de decimale punt bevatten.

Na de declaratie

```
double d;
```

```
d
```

Kun je de variabele een waarde geven met een toekenningsopdracht

```
d = 3.141592653;
```

```
d
```

Overeenkomstig de angelsaksische gewoonte wordt in dit soort getallen een decimale punt gebruikt, en dus niet zoals in Nederland een decimale komma.

Variabelen van het type `double` kunnen ook gehele getallen bevatten; er komt dan automatisch 0 achter de decimale punt te staan:

```
d = 10;
```

```
d
```

Anders dan bij het type `int`, treden er bij deling van `double` variabelen slechts kleine afrondingsfouten op:

```
d = d / 3;
```

```
d
```

Naast `int` en `double` zijn er in C# nog negen andere types voor numerieke variabelen. Acht van de elf numerieke types kunnen worden gebruikt voor gehele getallen. Het verschil is het bereik van de waarden die kunnen worden gerepresenteerd. Bij sommige types is de maximale waarde die kan worden opgeslagen groter, maar variabelen van dat type kosten dan ook meer geheugen. Sommige typen kunnen zowel negatieve als positieve getallen bevatten, andere typen alleen positieve getallen (en nul).

type	ruimte	kleinst mogelijke waarde	grootst mogelijke waarde
<code>sbyte</code>	1 byte	-128	127
<code>short</code>	2 bytes	-32768	32767
<code>int</code>	4 bytes	-2147483648	2147483647
<code>long</code>	8 bytes	-9223372036854775808	9223372036854775807
<code>byte</code>	1 byte	0	255
<code>ushort</code>	2 bytes	0	65535
<code>uint</code>	4 bytes	0	4294967295
<code>ulong</code>	8 bytes	0	18446744073709551615

Het type `long` is alleen maar nodig als je van plan bent om extreem grote of kleine waarden te gebruiken. De types `byte` en `short` worden gebruikt als het bereik van de waarden beperkt blijft. De besparing in geheugengebruik die dit oplevert is eigenlijk alleen de moeite waard als er erg veel (duizenden of zelfs miljoenen) van dit soort variabelen nodig zijn.

De typen `short`, `int` en `long` hebben alle drie een 'unsigned' variant, waarvan de naam met een 'u' begint. Het type `byte` is juist al 'unsigned' van zichzelf, en heeft een 'signed' versie `sbyte`.

Voor getallen met cijfers achter de komma zijn er drie verschillende types beschikbaar. Ze verschillen behalve in de maximale waarde die kan worden opgeslagen ook in het aantal significante cijfers dat beschikbaar is.

type	ruimte	significante cijfers	grootst mogelijke waarde
<code>float</code>	4 bytes	7	3.4×10^{38}
<code>double</code>	8 bytes	15	1.7×10^{308}
<code>decimal</code>	16 bytes	28	7.9×10^{28}

Hier is het type `float` het zuinigst met geheugenruimte, het type `double` kan erg grote getallen opslaan en dat ook nog eens nauwkeuriger, het type `decimal` kan het nog nauwkeuriger, maar dan weer niet overdreven groot.

Ieder type heeft zijn eigen doelgroep: `decimal` voor financiële berekeningen, `double` voor technische of wiskundige, en `float` als nauwkeurigheid niet van groot belang is, maar zuinig geheugengebruik wel.

4.2 Objecten

Object: groepje variabelen dat bij elkaar hoort

Een variabele is een geheugenplaats met een naam, die je kunt veranderen met een toekenningso opdracht. Een variabele `x` kan bijvoorbeeld op een bepaald moment de waarde 7 bevatten, en een tijdje later de waarde 12.

In veel situaties is het handig om meerdere variabelen te groeperen en als één geheel te behandelen. Bijvoorbeeld, met twee variabelen, laten we zeggen `x` en `y`, kun je de positie van een punt in het platte vlak beschrijven. Die twee variabelen zou je dan samen als één ‘positie-object’ kunnen beschouwen. Een *object* is een groepje variabelen dat bij elkaar hoort. `C#` is een object-georiënteerde programmeertaal. Natuurlijk spelen in zo’n taal objecten een belangrijke rol.

Met twee getallen kun je een positie in het platte vlak (op het scherm, op papier) beschrijven: de *x*-coördinaat en de *y*-coördinaat. Twee variabelen die ieder een getal bevatten zijn dus samen als één ‘positie-object’ te beschouwen.

Met drie getallen kun je een kleur beschrijven: de hoeveelheid rood, groen en blauw licht die in de kleur gemengd zijn. Drie variabelen die ieder een getal bevatten zijn dus samen als één ‘kleur-object’ te beschouwen.

Objecten schermen hun opbouw af

Voor het beschrijven van complexere zaken zijn veel meer variabelen nodig. Voor het beheer van een window op het scherm zijn variabelen nodig om de positie van het window te bepalen, de afmetingen, de achtergrondkleur en de naam die in de titelbalk staat, het aanwezig zijn van scrollers en als die er zijn de positie daarvan, om nog maar niet te spreken van de inhoud van het window. Het is duidelijk dat het erg gemakkelijk is om in een programma een window in z’n geheel te kunnen manipuleren, in plaats van steeds opnieuw met al die losse variabelen te worden geconfronteerd.

Het is lang niet altijd nodig om precies te weten uit welke variabelen een bepaald object is opgebouwd. Het kan handig zijn om je er ongeveer een voorstelling van te maken, maar strikt noodzakelijk is dat niet. Om je een voorstelling te maken van een kleur-object kun je aan een groepje van drie variabelen denken, maar ook zonder die kennis kun je een kleur-object manipuleren. We hebben dat in sectie 2.5 gedaan: door het toekennen van een `Color`-object aan de property `BackColor` werd de achtergrondkleur van een `Form` bepaald. Dat kan, zonder dat we hoefden te weten dat een kleur-object in feite een groepje van drie variabelen is.

Het is eerder regel dan uitzondering dat je niet precies weet hoe een object is opgebouwd. In programma’s worden forms, labels, buttons, files, en allerlei andere objecten gebruikt, zonder dat de programmeur de opbouw van die objecten in detail kent. Die details worden (gelukkig) afgeschermd in de library-klassen.

Van de meeste standaard-objecten (forms, buttons enz.) is het zelfs zo dat je de opbouw niet te weten kan komen, zelfs als je dat uit nieuwsgierigheid zou willen. Dat is geen pesterij: de opbouw van objecten wordt geheim gehouden om de auteur van de standaard-bibliotheek de vrijheid te geven om in de toekomst een andere opbouw te kiezen (bijvoorbeeld omdat die efficiënter is), zonder dat bestaande programma’s daaronder te lijden hebben.

Als je zelf nieuwe object-typen samenstelt (we gaan dat doen in hoofdstuk 8) dan moet je natuurlijk wel weten hoe je eigenbedachte object is opgebouwd. Maar zelfs dan kan het een goed idee zijn om dat zo snel mogelijk weer te vergeten, en je eigen objecten waar mogelijk als ondeelbaar geheel te behandelen.

Objecten bieden properties en methoden aan

Voor het gebruik van objecten hoef je dus niet precies te weten uit welke variabelen ze bestaan. Wel moet je weten wat je er mee kunt doen: welke properties je ervan kunt bepalen (bijvoorbeeld de **Length** van een **string**-object) of veranderen (bijvoorbeeld de **BackColor** van een **Form**-object), en welke methoden je ermee kunt aanroepen (bijvoorbeeld de methode **DrawLine** van een **Graphics**-object). Daarnaast kun je het object natuurlijk ook als geheel gebruiken: door het mee te geven aan een methode (bijvoorbeeld een **Label**-object aan de methode **Add** van een **Form**-object), of aan de rechterkant van een toekenningsopdracht (bijvoorbeeld een **string**-object in een toekenning aan een variabele of property).

Soms is het gemakkelijk voor te stellen hoe een property van een object wordt bepaald. Zo kent een **Color**-object een property **R**, die de waarde van de rood-component van de kleur bepaalt, en properties **G** en **B** die de groen- en blauw-componenten bepalen. Helemaal zeker kun je het niet weten, maar deze properties corresponderen waarschijnlijk regelrecht met drie overeenkomstige variabelen die deel uitmaken van het object.

Soms is het minder goed voorstelbaar hoe een property van een object wordt bepaald. Zo kent een **string**-object een property **Length**, die de lengte van de tekst bepaalt. Zou er in het **string**-object een variabele zijn waar die lengte direct is te vinden? Of worden op het moment dat we de **Length**-property opvragen, de letters echt geteld? Het zou allebei kunnen, en er is misschien wel ergens op te zoeken hoe het ‘in het echt’ werkt, maar in feite doet het er niet toe, zolang die property maar te bepalen is. Het is ook mogelijk dat in een toekomstige versie van de library een andere keuze wordt gemaakt (omdat dat sneller is, of juist minder geheugen gebruikt). Als gebruiker van de library hoeven we daar niets van te merken, zolang onze vertrouwde properties er maar zijn.

Methoden hebben een object onder handen

Bij de aanroep van een methode vermeld je voor de punt een object (behalve bij statische methoden, daar staat een klasse-naam voor de punt). Dat object wordt door de methode onder handen genomen, dat wil zeggen: de methode heeft toegang tot de variabelen die onderdeel zijn van het object. Sommige methoden zullen (de variabelen van) het object alleen maar bekijken. Bijvoorbeeld: door de methode-aanroep

```
gr.DrawLine(Pens.Black, 10, 10, 30, 30);
```

worden de variabelen van het **Graphics**-object **gr** bekeken om te bepalen in welk window de lijn getekend moet worden. Maar het object **gr** zelf verandert niet als gevolg van deze aanroep.

Andere methoden zullen (de variabelen van) het object blijvend veranderen. Bijvoorbeeld, door de methode-aanroep

```
f.Controls.Add( new Label() );
```

zal er een nieuw label worden toegevoegd aan de controls van het form **f**. Daarbij wordt de controls-collection veranderd (hij wordt groter).

Sommige methoden veranderen objecten

Een ander voorbeeld van objecten die blijvend kunnen veranderen zijn objecten van het type **Rectangle**. Zo’n object kun je je voorstellen als de specificatie van een rechthoek: de positie en de afmetingen ervan. Waarschijnlijk wordt een **Rectangle**-object in het geheugen opgeslagen als vier getallen: de *x*- en *y*-coördinaat en de breedte en de hoogte, maar wie weet hebben de library-makers een andere keuze gemaakt (bijvoorbeeld de coördinaten van de linkerbovenhoek en de rechteronderhoek). Dat hoeven we niet te weten om **Rectangle**-objecten in een programma te gebruiken. Wat we wel moeten weten is dat er een constructor-methode bestaat die inderdaad de vier genoemde getallen als parameter krijgt. Zo kunnen we dus een **Rectangle**-object maken:

```
Rectangle r;  
r = new Rectangle(50, 40, 60, 20);
```

Later in het programma kun je het rectangle-object meegeven aan de methode van **Graphics**-objecten die een rechthoek tekent. In een eerder programma hebben we deze methode aangeroepen met vier losse parameters, maar nu krijgt hij de rechthoek als geheel mee:

```
gr.FillRectangle(Brushes.Blue, r);
```

Er zijn methoden die een **Rectangle**-object blijvend veranderen. Bijvoorbeeld: **Offset** verplaatst

de positie, en `Inflate` maakt hem groter. Schrijven we bijvoorbeeld:

```
r.Offset(220, 0);
r.Inflate(20, 0);
gr.DrawRectangle(Brushes.Red, r);
```

dan verschijnt er een rode rechthoek op het scherm, 220 beeldpunten verder naar rechts, en links en rechts 20 punten groter gegroeid. De x -coördinaat wordt dus $50 + 220 - 20 = 250$, de breedte $60 + 2 \times 20 = 100$. Door de aanroep van `Offset` en `Inflate` is het object `r` dus veranderd.

Een ander voorbeeld van objecten die kunnen veranderen door er methoden van aan te roepen zijn `Bitmap`-objecten, die plaatjes specificeren. Je kunt een `Bitmap`-object veranderen door hem onder handen te nemen met de methode `RotateFlip`. Eén van de constructoren van `Bitmap` kan een plaatje uit een file inlezen. Met het volgende programma-fragment kunnen we dus een plaatje inlezen, en kwart slag draaien, en vervolgens op het scherm tekenen:

```
Bitmap bm;
bm = new Bitmap("plaatje.jpg");
bm.RotateFlip(RotateFlipType.Rotate90FlipNone);
gr.DrawImage(bm, 10, 10);
```

De positie van de bitmap wordt niet opgeslagen in het object, en moet dus apart aan `DrawImage` worden meegegeven. De parameter van `RotateFlip` ziet er ingewikkeld uit, maar het komt er op neer dat we een keuze maken uit één van de statische properties van `RotateFlipType` waarmee we een combinatie van kwartslag/halve slag draaien en/of spiegelen in x - of y -richting kunnen specificeren.

Immutable objecten kunnen niet veranderen

Niet alle objecten hebben methoden waarmee we ze kunnen veranderen. Bij `Rectangle`- en `Bitmap`-objecten zijn dat soort methoden er wel, maar bijvoorbeeld `Color`- en `Pen`-objecten kunnen, nadat je ze eenmaal hebt gemaakt, niet meer veranderen. We spreken dan van *immutable objecten*.

Je kunt wel variabelen declareren voor immutable-objecttypen. Bijvoorbeeld:

```
Pen p;
p = new Pen(Color.Black, 5);
gr.DrawLine(p, ...);
p = new Pen(Color.Green, 3);
gr.DrawLine(p, ...);
```

kent twee verschillende `Pen`-objecten toe aan de variabele `p`, en tekent daarmee lijnen op het scherm. De variabele `p` wijzigt wel, maar het `Pen`-object dat in die variabele is opgeslagen wijzigt niet.

4.3 Object-variabelen

We gaan nu bekijken wat er precies in het geheugen gebeurt als je een variabele declareert met een object-type, en wat als je er waarden aan toekent met toekenningsoopdrachten. Als demonstratie dient het programma `CopyDemo`. In listing 8 staat de programmatekst, en in figuur 11 is het programma in werking te zien. Heel erg spannend is de output niet, want het tekent alleen drie rechthoeken en drie bitmaps. Maar het programma laat goed zien wat er gebeurt als je object-variabelen kopieert.

Twee soorten object-variabelen

In het programma worden drie variabelen van het type `Rectangle` gedeclareerd, en drie variabelen van het type `Bitmap`. Er worden `new` objecten gemaakt, de objecten worden onder handen genomen door methoden, en ze worden meegegeven aan `Graphics`-methoden die de informatie in de objecten gebruikt om dingen op het scherm te tekenen.

Als je de listing bekijkt, dan lijkt er een grote analogie te zijn tussen deze twee situaties. Toch gebeuren er wezenlijk andere dingen in het geheugen. De reden daarvoor is dat het type `Rectangle` in de library als `struct` is gedefinieerd, terwijl het type `Bitmap` in de library als `class` is gedefinieerd.

Rectangle-variabelen bevatten waardes

In het programma worden drie `Rectangle`-variabelen gedeclareerd:

```
Rectangle r1, r2, r3;
```

```
using System.Windows.Forms;
using System.Drawing;
using System.IO;

5 class CopyForm : Form
{
    public CopyForm()
    {
        this.Text = "CopyDemo";
        this.BackColor = Color.White;
10      this.Size = new Size(680, 340);
        this.Paint += this.tekenScherm;
    }

15  void tekenScherm(object obj, PaintEventArgs pea)
    {
        Graphics gr = pea.Graphics;
        Rectangle r1, r2, r3;
        Bitmap bm1, bm2, bm3;

20      // Maak en teken een rechthoek en een bitmap
        r1 = new Rectangle(50, 40, 60, 20);
        bm1 = BitmapFactory.DecodeResource(this.Resources, Resource.Drawable.telefoon);

25      gr.FillRectangle(Brushes.Blue, r1);
        gr.DrawImage(bm1, 20, 100);

        // Maak twee kopieen van de rechthoek, verander ze, en teken ze
        r2 = r1;
        r3 = r1;
30      r2.Offset(220, 0); r2.Inflate(20, 10);
        r3.Offset(440, 0); r3.Inflate(0, 30);
        gr.FillRectangle(Brushes.Red, r2);
        gr.FillRectangle(Brushes.Green, r3);

35      // Maak twee kopieen(?) van de bitmap, verander ze, en teken ze
        bm2 = bm1;
        bm3 = bm1;
        bm2.RotateFlip(RotateFlipType.Rotate90FlipNone); // draai bm2 rechtsom
        bm3.RotateFlip(RotateFlipType.RotateNoneFlipX); // spiegel bm3
40      gr.DrawImage(bm2, 240, 100);
        gr.DrawImage(bm3, 460, 100);
    }
}

45 class CopyDemo
{
    static void Main()
    {
50      Application.Run(new CopyForm());
    }
}
```

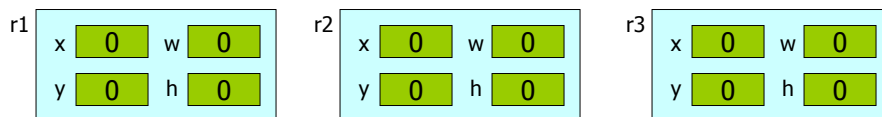
Listing 8: CopyDemo/CopyDemo.cs



Figuur 11: Het programma CopyDemo in werking

Er wordt daarmee ruimte in het geheugen gereserveerd voor drie objecten. Hoe zo'n object intern is opgebouwd kun je nooit helemaal zeker te weten komen, maar voor een `Rectangle` is het aannemelijk dat daarin de vier kenmerken van een rechthoek worden opgeslagen: de x - en y -coördinaten en de breedte en de hoogte. Alle variabelen van een object krijgen aan het begin een neutrale waarde; voor `int`-variabelen is dat de waarde 0.

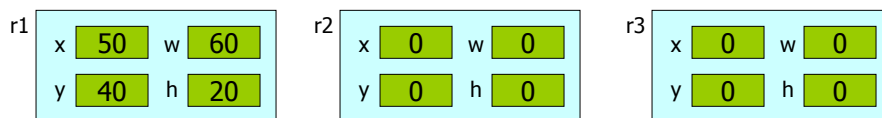
De situatie in het geheugen kun je nu dus als volgt voorstellen:



Door de toekenningsoopdracht

```
r1 = new Rectangle(50, 40, 60, 20);
```

krijgt de variabele `r1` een nieuwe waarde. De situatie in het geheugen wordt daarmee:



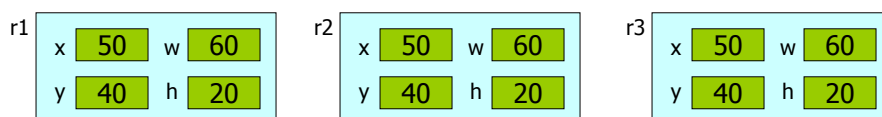
De rechthoek met deze kenmerken wordt met een blauwe kwast op het scherm getekend met

```
gr.FillRectangle(Brushes.Blue, r1);
```

Vervolgens worden in het programma twee toekenningsoopdrachten gedaan aan `r2` en `r3`:

```
r2 = r1;
r3 = r1;
```

Die twee variabelen worden daarmee een kopie van de variabele `r1`. Nadat de tekenningen zijn gedaan hebben de drie objecten dus allemaal dezelfde waarde:

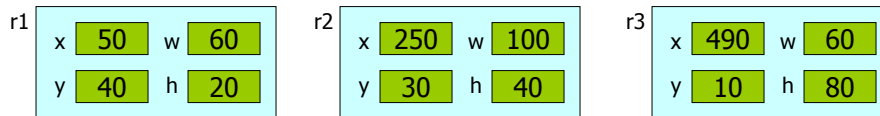


Dat gaat veranderen doordat we de objecten `r2` en `r3` onder handen nemen met methoden:

```
r2.Offset(220, 0); // x wordt 220 groter
r2.Inflate(20, 10); // x en y worden iets kleiner, maar w en h juist groter
```

```
r3.Offset(440, 0);
r3.Inflate(0, 30);
```

De methode `Offset` verandert de coördinaten van de rechthoek met de meegegeven hoeveelheden. Dus de x -coördinaat van `r2` verandert van 50 in 270. De methode `Inflate` maakt de rechthoek aan alle randen groter, terwijl het midden op dezelfde plaats blijft. Dus de x -coördinaat van `r2` wordt weer 20 kleiner, maar de breedte wordt tweemaal 20 groter; de y -coördinaat wordt 10 kleiner, maar de hoogte wordt tweemaal 10 groter. Op een soortgelijke manier nemen we het object `r3` onder handen. De eindsituatie van het geheugen is als volgt:



Let wel, de rechthoeken zijn nog niet getekend, er zijn alleen maar getallen in het geheugen veranderd. Het tekenen gebeurt daarna pas met de opdrachten

```
gr.FillRectangle(Brushes.Red, r2);
gr.FillRectangle(Brushes.Green, r3);
```

en daarmee krijgen we inderdaad een rode rechthoek die vooral in de breedte is opgerekt, en een groene rechthoek die alleen in de hoogte is opgerekt te zien.

Bitmap-variabelen bevatten verwijzingen

Heel anders gaat het in zijn werk bij declaratie van variabelen waarvan het type als `class` is gedefinieerd, zoals `Bitmap`. Met de declaratie

```
Bitmap bm1, bm2, bm3;
```

wordt niet ruimte gereserveerd voor de objecten zelf, maar voor *verwijzingen* naar de objecten. Direct na de declaratie wijzen deze verwijzings-variabelen nog nergens naar, en hebben ze een neutrale waarde: de speciaal voor dit doel bestaande waarde `null`.

Een verwijzing is veel kleiner dan het object zelf; in grootte is het vergelijkbaar met een getal. Je kunt je de situatie in het geheugen dus als volgt voorstellen:

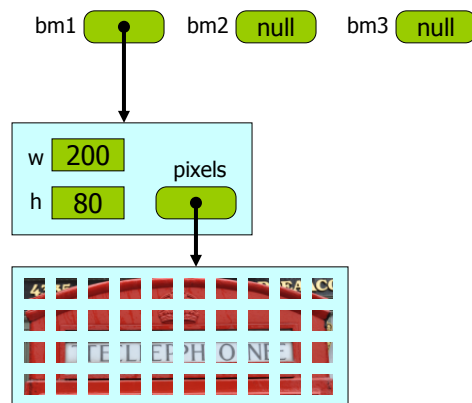


De verwijzing gaat pas naar een object wijzen met een toekenningsoopdracht:

```
bm1 = new Bitmap("telefoon.jpg");
```

Door de aanroep van de constructormethode wordt het object gemaakt. Dat object heeft zelf geen naam, maar via de verwijzingsvariabele is het toch toegankelijk.

De interne opbouw van een `Bitmap`-object is wat minder gemakkelijk te raden dan die van een `Rectangle`-object. Waarschijnlijk worden de breedte en de hoogte van het plaatje opgeslagen, eventueel nog wel wat meer informatie, maar in ieder geval ook de kleuren van de afzonderlijke beeldpunten (pixels). Misschien gebeurt dat laatste wel in een apart object, en slaat het `Bitmap`-object daar op zijn beurt een verwijzing naar op. De situatie in het geheugen is na de toekenningsoopdracht dan ongeveer als volgt:



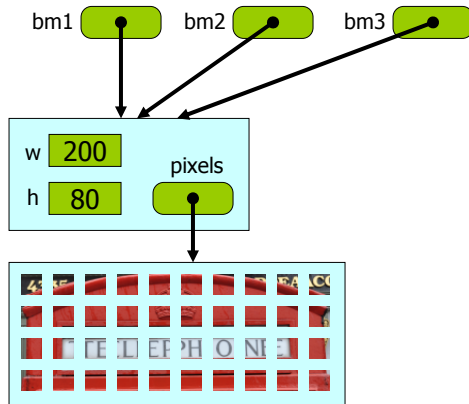
Voor de duidelijkheid zijn in dit schema de pixels van het plaatje als kleine mini-plaatjes getekend. In werkelijkheid zijn het er veel meer, en staan er in het geheugen niet zozeer gekleurde vlakjes, als wel getallen die de kleuren symboliseren.

Met een aanroep van **DrawImage** wordt het plaatje, zoals dat momenteel in het object (waarnaar verwezen wordt door) **bm1** is opgeslagen, op het scherm getekend.

Verderop in het programma staan twee toekenningsoverdrachten aan **bm2** en **bm3**:

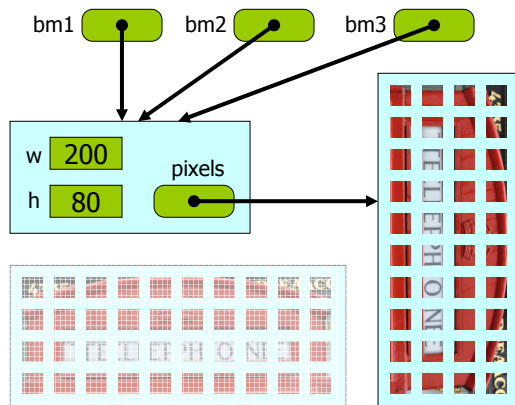
```
bm2 = bm1;
bm3 = bm1;
```

Het is hier dat er iets anders gebeurt dan in het geval van de **Rectangle**-objecten. Nu wordt namelijk niet het complete object gekopieerd, maar alleen de verwijzing erheen. Het effect op het geheugen is dus als volgt:



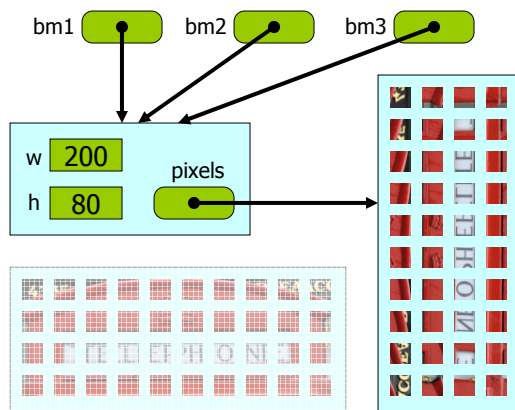
De volgende opdracht in het programma is een aanroep van **RotateFlip**, waarbij het object (waarnaar verwezen wordt door) **bm2** onder handen genomen wordt. Het effect is dat de breedte en de hoogte worden verwisseld, en de beeldpunten zodanig door elkaar worden gehusseld dat het plaatje geroteerd is. Waarschijnlijk wordt daarbij een nieuw object gemaakt om die beeldpunten te bevatten. Het oude object is er ook nog wel, maar als daar niemand meer naar wijst is het helemaal onbereikbaar geworden, en had het er dus net zo goed niet meer kunnen zijn.

Eindresultaat is dat **bm2** wijst naar een object dat het geroteerde plaatje beschrijft.



In het schema valt het op dat hiermee in feite ook het object **bm1** is veranderd. Niet de variabele **bm1**, want die bevat nog precies dezelfde pijl als voorheen, maar het object waarnaar **bm1** wijst is geroteerd, want dit is hetzelfde object als dat waar **bm2** naar wijst, en dat hebben we onder handen genomen.

Het beeld op het scherm wijzigt echter niet: **bm1** was al getekend voordat z'n object stiekem was veranderd, en **bm2** is nog niet getekend. Dat gebeurt later in het programma, maar eerst wordt **bm3** onder handen genomen: ook met **RotateFlip**, maar ditmaal met een parameter die aangeeft dat het plaatje gespiegeld ('flipped') moet worden. Het gevolg is dat de beeldpunten van het object waar **bm3** naar wijst weer anders worden gerangschikt. En door de manier waarop de drie verwijzings-variabelen naar hetzelfde object wijzen, zijn **bm1** en **bm2** in zekere zin mee-veranderd:



Als laatste opdrachten in het programma worden **bm2** en **bm3** op het scherm getekend. Het scherm ziet er dan uit zoals in figuur 11.

Hier is nu iets raars aan de hand: het tweede plaatje is niet alleen geroteerd maar ook geflipt, het derde plaatje is niet alleen geflipt maar ook geroteerd! Als je de situatie in het geheugen begrijpt is dat wel logisch, want er was in feite maar één object, waar alle drie de variabelen naar toe wijzen. Maar deze situatie was dus anders geweest als **Bitmap** net als **Rectangle** een **struct** was geweest.

Voorbeelden van objectverwijzingstypen

In de libraries die met C# worden meegeleverd zijn een groot aantal typen gedefinieerd. De verzameling wordt met elke versie van C# verder uitgebreid (en soms komen er ook te vervallen als er een beter alternatief beschikbaar komt). Daarnaast kun je ook nog extra libraries kopen/krijgen/maken, waarmee het aantal typen nog verder wordt uitgebreid.

Veel typen zijn klassen; als je objecten van zo'n type declareert krijgt je dus een verwijzing, die je naar **new** gemaakte objecten kunt laten wijzen. Sommige typen zijn structs; als je objecten van zo'n type declareert komt het object direct in het geheugen, en met **new** kun je nieuwe waarden voor zo'n object maken.

Om een idee te geven van welke object-typen er bestaan volgt hier een kleine selectie:

- typen van objecten waarin heel duidelijk een klein groepje variabelen is te herkennen: **Point** (twee gegroepeerde **int**-variabelen: een *x*- en een *y*-coördinaat), **Size** (twee gegroepeerde variabelen: een lengte en een breedte), **Color** (vier bytes: de hoeveelheid rood, groen en blauw, en de als 'alfa' bekend staande doorschijnendheid), **DateTime** (een groepje variabelen die een datum en een tijd kunnen bevatten), **TimeSpan** (een groepje variabelen die een tijdsduur kunnen bevatten).
- typen van objecten met een wat complexere structuur, die een zeer natuurlijke eenheid vormen: **String** (een tekst bestaande uit nul of meer lettertekens), **Font** (een lettertype), **Bitmap** (een afbeelding).
- typen van objecten die nodig zijn om een interactieve interface te maken: **Label** (tekstpaneeltje op het scherm), **Button** (een drukknop op het scherm), **TrackBar** (een schuifregelaar), **TextBox** (een invulveld voor de gebruiker), maar ook samengestelde componenten zoals **Form**
- typen van objecten waarin de details van een bepaalde gebeurtenis kunnen worden weergegeven, zoals **PaintEventArgs**
- typen van objecten die een bepaald kunstje kunnen uitvoeren, zoals **Graphics** (om een tekening te maken)
- typen van objecten waarmee files en internet-verbindingen gemanipuleerd kunnen worden: **File**, **URL**, **FileReader**, **FileWriter** en vele anderen.

Van al deze typen kunnen variabelen worden gedeclareerd. In het geval van **struct** typen bevat de variabele dan het object zelf, in het geval van **class** typen is de variabele een verwijzing die kan wijzen naar het eigenlijke object.

Klasse: groepje methoden en type van object

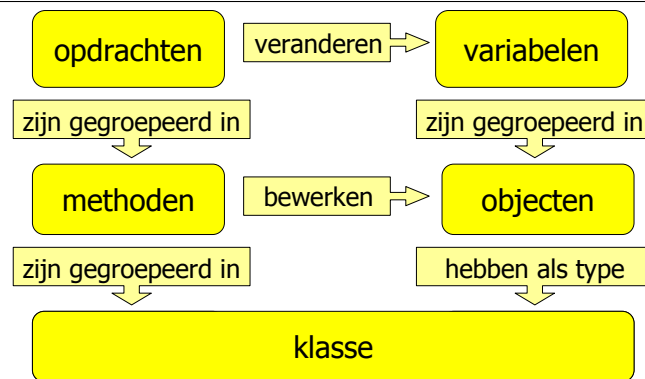
In sectie 1.2 hebben we het begrip 'klasse' gedefinieerd als 'een groepje methoden met een naam'. Als je zelf methoden schrijft moet je die onderbrengen in een klasse. Ook de standaardmethoden zijn ondergebracht in een klasse: zo is de methode **DrawRectangle** bijvoorbeeld beschikbaar in de klasse **Graphics**.

Maar klassen spelen nog een andere rol: ze zijn het type van objecten. Je gebruikt de naam van de klasse dan als type van een (verwijzings-)variabele, zoals je ook de ingebouwde types zoals `int` kunt gebruiken. Vergelijk:

```
int x; Color geel;
```

De twee rollen die een klasse kan spelen zijn sterk met elkaar verbonden. Methoden hebben immers een object onder handen (het object dat voor de punt staat in de methode-aanroep). Dat object bestaat uit variabelen, die kunnen worden veranderd door de opdrachten in de methode. Objecten die een bepaalde klasse als type hebben, kunnen onder handen worden genomen door de methoden uit die klasse. Of anders gezegd: de methoden van een klasse kunnen objecten onder handen nemen, die die klasse als type hebben.

In figuur 12 staat de samenhang tussen de begrippen opdracht, variabele, methode, object en klasse, waarbij de dubbele rol van klassen duidelijk naar voren komt.



Figuur 12: De twee rollen van het begrip “klasse”

4.4 Typering

Typering voorkomt fouten

Elke variabele heeft een type, die door de declaratie van de variabele is vastgelegd. Het type kan een van de elf numerieke basistypen zijn (de variabele kan dan een getal van dat type bevatten), een klasse (de variabele kan dan een verwijzing naar een object van die klasse bevatten), of een struct (de variabele kan dan het object zelf bevatten),

Declaraties worden verwerkt door de compiler. Dat is wat ze onderscheidt van opdrachten, die tijdens het runnen van het programma worden uitgevoerd. Door de declaraties ‘kent’ de compiler de typen van alle variabelen.

De compiler is daardoor in staat om te controleren of aanroepen van methoden wel zinvol zijn. Methoden uit een bepaalde klasse kunnen immers alleen worden aangeroepen met een object onder handen dat die klasse als type heeft. Klopt de typering niet, dan geeft de compiler een foutmelding. De compiler genereert dan geen uitvoerbaar programma, en het programma kan niet worden gerund.

Hoewel het in de praktijk een heel gedoe kan zijn om de compiler helemaal tevreden te stellen wat betreft de typering van het programma, is dat verre te prefereren boven de situatie waar vergissingen met de typering pas aan het licht zouden komen bij het uitvoeren van het programma. In programmeertalen waarin geen of een minder strenge vorm van typering wordt gebruikt kunnen er verborgen fouten in een programma zitten. Als de bewuste opdrachten bij het testen toevallig niet aan bod zijn gekomen, blijft de fout in de code sluimeren totdat een ongelukkige gebruiker in een onwaarschijnlijke samenloop van omstandigheden de foute opdracht eens tegenkomt. Voor de programmeur is het een onrustbarende gedachte dat dat zou kunnen gebeuren – daarom is het goed dat de C#-compiler de typering zo streng controleert.

Als de compiler geen foutmeldingen meer geeft, betekent dat niet dat het programma ook gegarandeerd foutvrij is. Een compiler kan natuurlijk niet de bedoeling van de programmeur raden, en waarschuwen voor het feit dat er een rode cirkel wordt getekend in plaats van een groene. Wel

kan de compiler weten dat ‘groen’ als diameter van een cirkel nooit kan kloppen, omdat ‘groen’ een kleur is en de diameter een getal moet zijn.

De compiler controleert de typen van objecten die door een methode onder handen worden genomen, en ook van alle parameters van een methode. Ook bij het gebruik van rekenkundige operatoren worden de types van de twee argumenten gecontroleerd, zodat bijvoorbeeld niet twee kleuren opgeteld kunnen worden, maar alleen getallen of objecten waarvoor een ‘optelling’ zinvol kan zijn (zoals `string`, waarbij de plus-operator ‘teksten samenvoegen’ betekent).

Conversie van numerieke typen

Waarden van numerieke typen zijn in sommige situaties uitwisselbaar. Zo is de waarde 12 in principe van het type `int`, maar het is ook acceptabel als rechterkant van een toekenningsoopdracht aan een variabele van type `double`. Bijvoorbeeld, na de declaraties

```
int i;
double d;
```

Zijn de volgende toekenningen acceptabel:

```
i = 12;
d = 12;    // waarde wordt automatisch geconverteerd
d = i;     // waarde wordt automatisch geconverteerd
```

Bij de toekenningen van een `int`-waarde aan de `double` variabele, of dat nu een constante is of de waarde van een `int`-variabele, wordt de waarde automatisch geconverteerd.

Omgekeerd is toekenning van een `double`-waarde aan een `int`-variabele niet mogelijk, omdat er in een `int`-variabele geen ruimte is voor cijfers achter de decimale punt. De controle wordt uitgevoerd door de compiler op grond van de typen. Een `double`-expressie is nooit acceptabel als waarde voor een `int`-variabele, zelfs niet als de waarde toevallig een nul achter de decimale punt heeft. De compiler kan dat namelijk niet weten, omdat de uitkomst van berekeningen kan afhangen van de situatie tijdens het runnen. De controle gebeurt puur op grond van het type, en daarom zijn zelfs toekenningen van constanten met 0 achter de decimale punt aan een `int`-variabele verboden.

```
d = 2.5;    // dit is goed
i = 2.5;    // FOUT: double-waarde past niet in een int
i = d;      // FOUT: double-waarde past niet in een int
i = 2*d;    // FOUT: typecontrole doet geen berekeningen
i = 5.0;    // FOUT: 5.0 blijft een double
i = 5;      // dit mag natuurlijk wel
```

Het kan natuurlijk gebeuren dat je als programmeur zeker weet dat de conversie van `double` naar `int` in een bepaalde situatie wel verantwoord is. Je kunt dat aan de compiler meedelen door vóór de expressie tussen haakjes het gewenste type te zetten, dus bijvoorbeeld:

```
i = (int) d;        // cast converteert double naar int
i = (int) (2*d);    // cast van een double-expressie
```

De compiler accepteert de toekenningen, en converteert de `double`-waarden naar `int`-waarden door het gedeelte achter de decimale punt weg te gooien. Als er 0 achter de decimale punt staat is dat natuurlijk geen probleem; anders gaat er enige informatie verloren. Als programmeur geef je door het expliciet vermelden van `(int)` aan dat je dat geen probleem vindt. De conversie is een ruwe manier van afronden: 2.3 wordt geconverteerd naar 2, maar ook 2.9 wordt 2. De cijfers achter de decimale punt worden zonder meer weggegooid, er wordt niet afgerond naar de dichtstbijzijnde integer.

Deze notatie, waarmee expressies van een bepaald type kunnen worden geconverteerd naar een ander type, staat bekend als een *cast*. Letterlijk is de betekenis daarvan (althans een van de vele) een ‘gietvorm’; door middel van de cast wordt de double-expressie als het ware in de gietvorm van een `int` geperst.

Behalve voor conversie van `double` naar `int`, kan de cast-notatie ook worden gebruikt om conversies af te dwingen van `long` naar `int`, van `int` naar `short`, van `short` naar `ubyte`, en van `double` naar `float`; kortom in alle gevallen waarin de compiler het onverantwoord acht om een ‘grote’ waarde in een ‘kleine’ variabele te stoppen, maar waarin je als programmeur kan beslissen om de toekenning toch te laten plaatsvinden. Voor conversie van ‘klein’ naar ‘groot’ is een cast niet nodig, omdat daarbij nooit informatie verloren kan gaan.

Bij conversies van ‘signed’ typen naar hun ‘unsigned’ tegenhanger en andersom is de cast in beide

richtingen nodig. Bijvoorbeeld als je een `int` waarde wilt toekennen aan een `uint` variabele (de compiler is bang dat de waarde negatief zou kunnen zijn, wat in een `uint` niet opgeslagen kan worden), maar ook als je een `uint` waarde wilt toekennen aan een `int` variabele (de compiler is bang dat de waarde net te groot is om in een `int` te passen).

Operatoren en typering

Bij het gebruik van rekenkundige operatoren hangt het van de typen van de argumenten af, op welke manier de operatie wordt uitgevoerd:

- zijn beide argumenten een `int`, dan is het resultaat ook een `int`; bijvoorbeeld: het resultaat van `2*3` is 6, en het type daarvan is `int`.
- zijn beide argumenten een `double`, dan is het resultaat ook een `double`; bijvoorbeeld: het resultaat van `2.5*1.5` is 3.75, en het type daarvan is `double`.
- is één van de argumenten een `int` en de andere een `double`, dan wordt eerst de `int` geconverteerd naar `double`, waarna de berekening op `doubles` wordt uitgevoerd; het resultaat is dan ook een `double`. Bijvoorbeeld: het resultaat van `10.0/4` is 2.5, en het type daarvan is `double`.

Vooral bij een deling is dit van belang: bij een deling tussen twee integers wordt het resultaat naar beneden afgerond. Bijvoorbeeld: het resultaat van `10/4` is 2, met als type `int`. Als het resultaat daarna in een `double` variabele wordt opgeslagen, bijvoorbeeld met de toekenningsopdracht `d=10/4`; dan wordt de `int` 2 weer teruggeconverteerd naar de `double` 2.0, maar dan is het kwaad al geschied.

Een dergelijke regel geldt voor alle expressies waar een operator wordt toegepast op verschillende numerieke typen, bijvoorbeeld een `int` en een `long`: eerst wordt het ‘kleine’ type geconverteerd naar het ‘grote’ type, daarna wordt de operatie uitgevoerd, en het resultaat is het ‘grote’ type.

Een programmeur van een klasse kan er voor kiezen om ook operatoren een betekenis te geven voor objecten van die klasse. Eerder zagen we al dat de operator `+` gebruikt kan worden om strings samen te voegen. Als de linker operand van `+` een string is maar de rechter niet, wordt de rechter operand automatisch onderworpen aan de methode `ToString`.

Een andere type objecten waarop operator `+` kan werken is bijvoorbeeld `Size` (de breedte en de hoogte worden dan apart opgeteld). Ook kan een `Point` bij een `Size` worden opgeteld, en dan een nieuw `Point` oplevert. Twee `Point`-objecten bij elkaar optellen kan echter niet. In principe kunnen alle operatoren een betekenis krijgen voor nieuwe typen, maar dit wordt het vaakst gedaan voor de operator `+`.

4.5 Constanten

Numerieke constanten

Constanten van de numerieke typen kun je in het programma opschrijven als een rijtje cijfers, desgewenst met een minteken er voor. Dat ligt zo voor de hand dat we het al vele malen hebben gebruikt om `int`-constanten op te schrijven.

Hier zijn een paar voorbeelden:

```
0    3    17    1234567890    -5    -789
```

In feite zijn deze constanten niet allemaal van type `int`. Het type van een constante is namelijk het kleinste type waar het in past. Wanneer dat nodig is wordt dat type automatisch geconverteerd naar een ‘groter’ type, dus bijvoorbeeld van `byte` naar `int`, of van `int` naar `long`.

In bijzondere gevallen wil je een getal misschien in de 16-tallige (hexadecimale) notatie aangeven. Dat kan; je begint het getal dan met `0x`, waarachter behalve de cijfers 0 tot en met 9 ook de ‘cijfers’ `a` tot en met `f` mogen volgen. Voorbeelden:

```
0x10 (waarde 16)
0xa0 (waarde 160)
0xff (waarde 255)
0x100(waarde 256)
```

Constanten zijn van type `double` zodra er een decimale punt in voorkomt. Een nul voor de punt mag je weglaten (maar waarom zou je?). Voorbeelden van `double`-waarden zijn:

```
0.0    123.45    -7.0    .001
```

Voor hele grote, of hele kleine getallen kun je de ‘wetenschappelijke notatie’ gebruiken, bekend van de rekenmachine:

1.2345E3 betekent: 1.2345×10^3 , oftewel 1234.5
 6.0221418E23 het aantal atomen in een gram waterstof: 6.022×10^{23}
 3E-11 de straal van een waterstofatoom: 3×10^{-11} meter

Net als op een rekenmachine worden hele grote getallen niet meer exact opgeslagen. Er zijn circa 15 significante cijfers beschikbaar.

Om aan te geven dat een getal met punt en/of E-teken er in niet bedoeld is als **double** maar als **float**, kun je er de letter **F** achter zetten. Er is dan geen cast nodig in toekenningen als

```
float f = 1.0F;
```

Net zo moet er een **M** staan achter een **decimal** constante.

Behalve gewone getallen zijn er speciale waarden voor plus en min oneindig, en een waarde genaamd ‘NaN’ (voor ‘Not a Number’), die als resultaat gebruikt wordt bij onmogelijke berekeningen.

String constanten

Letterlijke teksten in een programma zijn constanten van het type **String**. Ook die hebben we al de nodige keren gebruikt. Je moet de tekst tussen dubbele aanhalingstekens zetten. Daartussen kun je vrijwel alle symbolen die op het toetsenbord zitten neerzetten. Voorbeelden:

"hallo"	een gewone tekst
"h o i !"	spaties tellen ook als symbool
"Grr#\$%]&*{"	in een tekst mag alles...
"1234"	dit is ook een string, geen int
""	een string met nul symbolen

Alleen een aanhalingsteken in een string zou problemen geven, omdat de compiler dat zou beschouwen als het einde van de string. Daarom moet je, als je toch een aanhalingsteken in een string wilt zetten, daar een backslash-symbool (omgekeerde schuine streep) vóór zetten. Dat roept een nieuw probleem op: hoe zet je het backslash-symbool zelf dan in een string? Antwoord: zet daar een extra backslash-symbool voor, dus verdubbel de backslash. Voorbeelden:

```
"Ze zei \"meneer\" tegen me!"
"gewone slash: / backslash: \\ hekje: # "
```

Met behulp van de backslash kunnen nog een aantal andere bijzondere tekens worden aangeduid: een regelovergang door `\n`, een tabulatie door `\t` en het symbool met Unicode-nummer (hexadecimaal) 12ab door `\u12ab`. Dat laatste is vooral handig om symbolen die niet op het toetsenbord voorkomen in een string te zetten.

Hoofdstuk 5

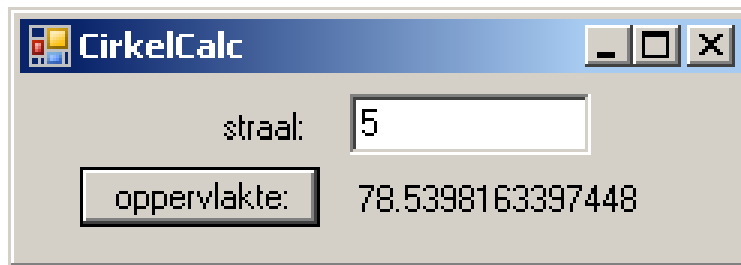
Interactie

5.1 Controls op een Form

In sectie 2.5 schreven we een programma dat een `Label` op een `Form` neerzette om daarin de tekst ‘Hallo’ te tonen. Een `Label` is het simpelste voorbeeld van een *control*: iets wat zichtbaar is voor de gebruiker in de grafische userinterface (GUI), en waar hij wat mee kan doen. In het geval van een label kan hij er alleen maar naar kijken.

We gaan nu een programma maken waarin nog twee andere soorten controls voorkomen: `TextBox` en `Button`. In een textbox kan de gebruiker een tekst intikken, een button kan hij indrukken (als het ware dan, met de muis op het scherm, maar de visuele ervaring van het verschuivende schaduwrandje is zodanig dat het net lijkt of de button echt wordt ingedrukt).

In figuur 13 is het programma in werking te zien. De gebruiker mag een getal intikken in de textbox, en het programma berekent dan de oppervlakte van een cirkel waarvan dat getal de straal is.



Figuur 13: Het programma CirkelCalc in werking

```
using System.Windows.Forms;

namespace CirkelCalc
{
    class Program
    {
        static void Main()
        {
            Application.Run(new Scherm());
        }
    }
}
```

Listing 9: CirkelCalc/Program.cs

```
using System.Windows.Forms;
using System.Drawing;
using System;

5 namespace CirkelCalc
{
    class Scherm : Form
    {
        private TextBox invoer;
        private Label uitkomst;

10        public Scherm()
        {
            Label tekst;
            Button knop;

15            tekst = new Label();
            knop = new Button();
            this.invoer = new TextBox();
            this.uitkomst = new Label();

            tekst.Location = new Point( 65, 10);
            knop.Location = new Point( 20, 30);
            this.invoer.Location = new Point(110, 6);
            this.uitkomst.Location = new Point(110, 34);
25            tekst.Size = new Size ( 35, 20);
            knop.Size = new Size ( 80, 20);
            this.invoer.Size = new Size(80, 20);
            this.uitkomst.Size = new Size(120, 20);
            tekst.Text = "straal:";
            knop.Text = "oppervlakte:";
            knop.Click += this.bereken;

            this.Text = "CirkelCalc";
35            this.ClientSize = new Size(240, 60);
            this.Controls.Add(tekst);
            this.Controls.Add(knop);
            this.Controls.Add(this.invoer);
            this.Controls.Add(this.uitkomst);
40        }

        private void bereken(object sender, System.EventArgs e)
        {
            double straal = double.Parse(invoer.Text);
            double opp = this.oppervlakte(straal);
45            uitkomst.Text = opp.ToString();
        }

        private double kwadraat(double x)
        {
50            return x * x;
        }

        private double oppervlakte(double r)
        {
            return Math.PI * this.kwadraat(r);
55        }
    }
}
```

Namespaces

blz. 60
blz. 61

De programmatekst is ditmaal gespreid over twee aparte files, die te zien zijn in listing 9 en listing 10. De reden hiervoor is dat we de klasse **Program** (waarin alleen de methode **Main** staat) in een aparte file hebben ondergebracht. Dat is niet per se noodzakelijk, want in één file mogen meerdere klassen gedefinieerd worden. Maar voor de overzichtelijkheid is het wel gebruikelijk om klassen, zeker als ze wat groter zijn, in aparte files te zetten.

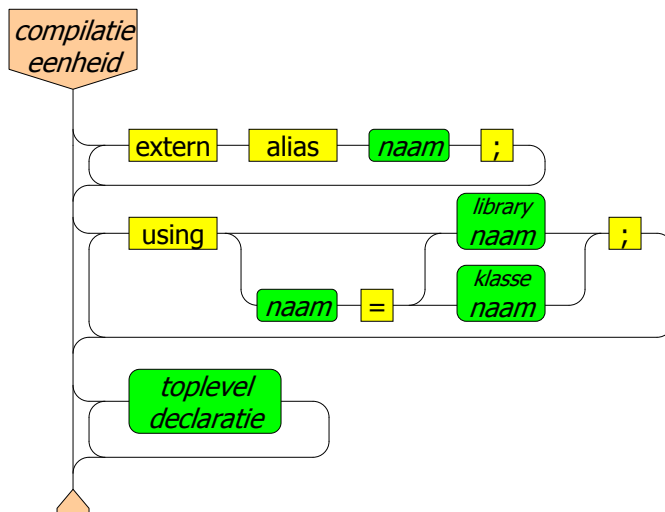
Het programma bestaat nu nog uit twee klassen. In grotere programma's kunnen dat er al gauw tientallen of honderden worden, en als je de libraries meetelt zelfs duizenden. Het zou onhandig worden als programmeurs bij het bedenken van de naam voor een klasse steeds met de collega-programmeurs zou moeten checken of die naam nog 'vrij' is, en daarnaast moet uitkijken dat de naam niet al in gebruik is voor een library-klasse. Daarom mogen klassen worden ondergracht in een zogeheten *namespace*. Binnen zo'n namespace moeten de namen van klassen uniek zijn, maar het is geen probleem als in andere namespaces klassen met dezelfde naam voorkomen.

In dit programma hebben we inderdaad een namespace geïntroduceerd: het is een extra schil rond de klasse-declaratie. We hebben nu dus één namespace, met daarin twee klassen, die ieder uit een of meer methoden bestaan, die ieder uit een of meer opdrachten bestaan.

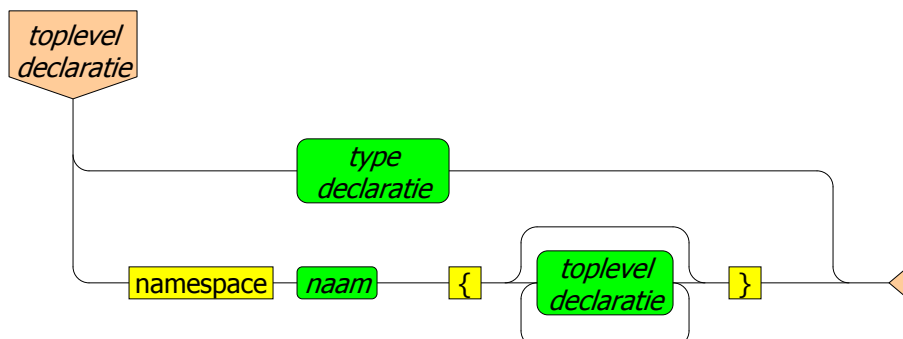
De twee klassen die samen in de namespace zijn ondergebracht staan wel in verschillende files. We kunnen dus niet de klassen samen tussen de accolades van de namespace zetten. Daarom moet de header van de namespace in elke file herhaald worden. Conceptueel horen de twee klassen echter wel thuis in dezelfde namespace.

Syntax van namespaces

In sectie 2.3 schreven we dat een compilatie-eenheid (een bestand met broncode) uit nul of meer klasse-declaraties bestaat. Dat was dus niet helemaal volledig, want nu blijkt dat een compilatie-eenheid ook een namespace mag zijn. Hier is een preciezere beschrijving van de syntax van een compilatie-eenheid. Die bestaat, na de *using*-directieven, uit één of meerdere *oplevel declaraties*.



Elke toplevel declaratie kan een *type-declaratie* zijn, en één van de mogelijkheden van een type-declaratie is een klasse-declaratie. Maar de tweede mogelijkheid voor toplevel declaratie is een namespace:



De body van een namespace bestaat weer uit toplevel declaraties, en dat kunnen dus onder andere klasse-declaraties zijn.

Uit het diagram blijkt dat het mogelijk is om binnen een namespace deel-namespaces aan te maken. We hebben dat in het voorbeeld-programma niet gebruikt, maar in de libraries komt dit wel voor. Zo heeft bijvoorbeeld de namespace `System` deel-namespaces `Drawing` en `Windows`. Die kunnen hiërarchisch worden aangeduid met een punt-notatie: `System.Drawing` en `System.Windows`. Namespaces mogen naar believen diep ‘genest’ worden. Zo bestaat de forms-library, waarin alle klassen die het type zijn van controls zijn ondergebracht, in feite uit de deel-deel-namespace `System.Windows.Forms`.

Controls voor interactie

In listing 10 geven we met een `using`-directief aan dat we de forms-library willen gebruiken. Ook hebben we de namespace `System.Drawing` nodig (vanwege de klassen `Point` en `Size`) en de namespace `System` zelf (vanwege de klasse `Math`). Met een `using`-directief krijg je alleen de typen (klassen, structs enz.) tot je beschikking die direct in de namespace staan, niet de typen uit de deel-namespaces. Daarom schrijven we aparte `using`-directieven voor het `System.Drawing` en `System.Windows.Forms`.

blz. 61

Uit de forms-library gebruiken we de klassen `TextBox` en `Button`. In de constructormethode van `Scherm` worden die op dezelfde manier gebruikt als `Label`: er wordt een `new` object aangemaakt, daarvan worden de nodige properties aangepast, en tenslotte worden ze door aanroep van methode `Add` toegevoegd aan de `Controls`-property. (Die property bestaat, omdat `Scherm` een subklasse is van `Form`, en elk `Form` beheert een collectie `Controls`).

Event-handler voor button-click

Een bijzondere property van `Button` is `Click`. Dit is een *event*, en we kunnen deze property dan ook niet aanpassen, maar wel ons erop ‘abonneren’. Dat wil zeggen: een methode naar keuze wordt aangeroepen op het moment dat het event optreedt. Het `Click`-event treedt natuurlijk op als de gebruiker de button aanklikt.

Het abonneren gebeurt met de opdracht

```
knop.Click += this.bereken;
```

die maakt dat onze zelfgeschreven methode `bereken` aangeroepen zal worden op het moment dat de button `knop` aangeklikt wordt.

Dit is hetzelfde mechanisme dat we hebben gebruikt in sectie 2.7 om een eigen methode te laten reageren op het `Paint`-event van een `Form`-object.

Variabele-declaraties als klasse-member

In de methode `bereken` schrijven we de opdrachten die uitgevoerd moeten worden op het moment dat de gebruiker op de knop klikt. Dit is het eigenlijke rekenwerk: we halen het getal op uit de tekstbox, berekenen de bijbehorende oppervlakte, en tonen die op het scherm in het voor dit doel klaargezette `Label`.

Om de tekstbox `tekst` en de label `uitvoer` te kunnen gebruiken (voor het inspecteren respectievelijk veranderen van de property `Text` ervan) moet de declaratie ervan geldig zijn binnen de methode `bereken`. Maar we kunnen deze variabelen niet lokaal in deze methode declareren, want de variabelen zijn ook nodig in de constructormethode. Als we ze daar zouden declareren, zijn ze weer niet te gebruiken in `bereken`. Tweemaal declareren is al helemaal geen optie, want dan zouden we twee verschillende variabelen krijgen die niets met elkaar te maken hebben.

De uitweg is om deze variabelen *buiten* de methode, maar nog wel binnen de klasse te declareren. De variabelen worden dan een zelfstandige *member* van de klasse, net zoals de methoden dat zijn. De semantiek van member-variabele-declaraties is als volgt. Elk object van de klasse krijgt de beschikking over een setje van de gedeclareerde variabelen. Een object *is* tenslotte een groepje variabelen, en de declaraties daarvan staan in de bijbehorende klasse. Alle methoden (behalve de statische) in de klasse hebben toegang tot deze variabelen: ze hebben immers zo’n object onder handen. Als we vanuit een methode een member-variabele willen gebruiken, bijvoorbeeld de variabele `invoer`, dan kunnen we die aanduiden als `this.invoer`. Dat is precies wat we doen, zowel in de constructormethode als in de methode `bereken`.

Public of private: zichtbaarheid van members

Bij alle members van een klasse, zowel de methoden als de variabelen, kun je bij de declaratie aangeven of deze **public** of **private** is. Dit heet de *zichtbaarheid* van de member. Members die **public** zijn, kunnen ook buiten de klasse gebruikt worden. In ons geval is dat alleen de constructormethode, want die is in de klasse **Program** nodig om een **new** object te maken. Alle andere methoden en member-variabelen zijn alleen binnen de eigen klasse nodig, en kunnen dus veiligheidshalve **private** gemaakt worden. Als je geen zichtbaarheid opgeeft, is deze standaard **private**.

Het is algemeen gebruik om **public** members met een hoofdletter te schrijven, en **private** members met een kleine letter. De library doet dat ook consequent, en wij in dit programma ook.

Converteren van en naar string

De variabele **invoer** is gedeclareerd als **TextBox** object(-verwijzing). Zo'n object kent de property **Text** die de tekst die de gebruiker in de tekstbox intikt bevat. Het type van die property is **string**, en daarmee kun je niet rekenen. We hebben dus een methode nodig die de string converteert naar een getal.

Zo'n methode bestaat: het is de statische methode **Parse** van het type **double**. Deze methode krijgt een string mee als parameter, en heeft als resultaat een **double**. In het programma kennen we die waarde eerst maar eens toe aan de variabele **straal**, in een gecombineerde declaratie-met-toekenning:

```
double straal = double.Parse(invoer.Text);
```

Bij deze straal willen we de oppervlakte van de bijbehorende cirkel berekenen. Een blik in het wiskundeboek leert dat de oppervlakte A berekend kan worden uit de straal r met de formule $A = \pi r^2$. Het zou leuk zijn als daarvoor ook een methode bestond, maar die is er helaas niet. Zo erg is dat nou ook weer niet, want we kunnen zo'n methode gemakkelijk zelf maken. Even aannemend dat dat is gebeurd, kunnen we de methode aanroepen met

```
double opp = this.oppervlakte(straal);
```

De berekende oppervlakte moet worden gerapporteerd aan de gebruiker. Daar hebben we een label **uitkomst** voor gedeclareerd, waarvan we de **Text**-property kunnen veranderen. Dat is echter een string, en daar kunnen we niet zomaar een **double** aan toekennen.

Voor het converteren naar een string kun je van zo'n beetje elke waarde de methode **ToString** aanroepen. Het is een beetje asymmetrisch ten opzichte van het omgekeerde parsen, want dit is een *niet*-statische methode, die de te converteren waarde *onder handen* neemt. Het converteren van de oppervlakte **opp** naar een string, die vervolgens meteen op het uitkomst-label op het scherm getoond wordt gebeurt met de opdracht

```
uitkomst.Text = opp.ToString();
```

5.2 Methoden met een resultaat

Berekenen van functies

Door middel van parameters kan de methode-aanroeper waarden (**int**-waarden, **double**-waarden, en zelfs waarden met een objectverwijzings-type zoals **Graphics**) doorspelen aan een methode, zodat die waarden in de body van de methode gebruikt kunnen worden. Dit is echter éénrichtingverkeer: de methode kan niets “terugzeggen” tegen de aanroeper.

Toch is dat vaak wel gewenst. Vergelijk het maar met het berekenen van functies in de wiskunde. Je kunt een functie, zoals de functie *kwadraat*, als het ware “aanroepen” door hem van een parameter te voorzien: *kwadraat*(5). Zo'n functie berekent een resultaat; in dit geval is dat 25. Dat antwoord is beschikbaar voor degene die de functie aanroept.

In C# kun je methoden, net als wiskundige functies, ook een resultaat laten berekenen, dat door de aanroeper kan worden gebruikt. En net als in de wiskundige functies kan een methode meerdere parameters krijgen, maar heeft hij maar één resultaat. Zo'n resultaatwaarde kan een bijvoorbeeld **int**- of een **double**-waarde zijn, maar je kunt methoden ook een object (-waarde of -verwijzing) laten teruggeven aan de aanroeper.

Het resultaattype van een methode

Net als de parameters, heeft de resultaat-waarde van een methode een *type*. Dat type kan **int** zijn als je een geheel getal wilt teruggeven, maar ook bijvoorbeeld het object-type **Color**, als je de

methode een kleur wilt laten teruggeven.

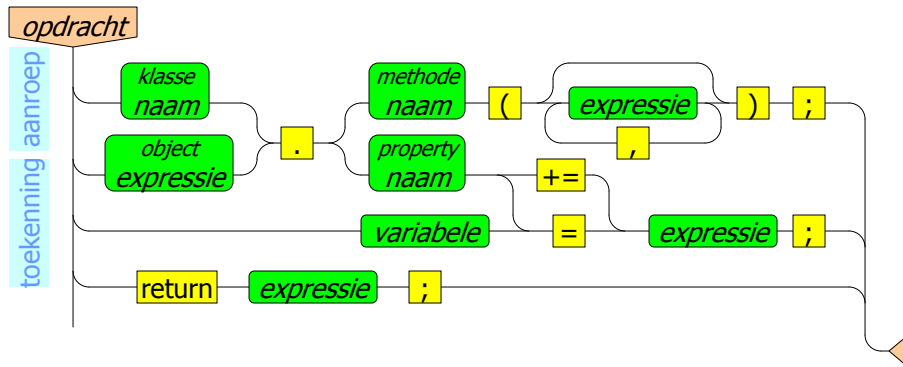
Het resultaattype van de methode moet in de header van de methode worden gespecificeerd, direct vóór de naam van de methode. De header van de kwadraat-methode kan er dus zo uitzien:

```
private double kwadraat(double x)
```

Als de methode geen resultaat heeft, staat op de plaats van het resultaattype het woord `void`.

De return-opdracht

In de body van een methode met een resultaat moet op de een of andere manier worden aangegeven wat het resultaat dan wel is. Voor dit doel is er in C# een speciale opdracht beschikbaar: de **return**-opdracht. Dit is de derde opdracht-vorm die we tegenkomen, naast de methode-aanroep en toekenningsoopdracht:



In de body van de kwadraat-methode staat als enige opdracht zo'n **return**-opdracht:

```
private double kwadraat(double x)
{
    return x*x;
}
```

Een **return**-opdracht bestaat uit het, speciaal voor dit doel gereserveerde, woord **return**, gevolgd door een expressie. Net als de toekennings-opdracht en de methode-aanroep wordt de return-opdracht afgesloten met een puntkomma.

Op het moment dat de return-opdracht wordt uitgevoerd, wordt de expressie uitgerekend, gebruik makend van de op dat moment geldende waarden van variabelen en parameters. Die waarde wordt dan als resultaatwaarde opgeleverd aan de aanroeper van de methode.

Aanroep van methoden

Methoden met een resultaatwaarde moeten op een andere manier worden aangeroepen dan methoden met `void` als resultaat-type; de resultaatwaarde die door de return-opdracht in de methode wordt teruggegeven moet immers op de een of andere manier worden geaccepteerd.

Aanroepen van void-methoden hebben de status van een zelfstandige opdracht. Bijvoorbeeld:

```
gr.DrawLine(Pens.Blue, 10,10, 20,20);
```

Aanroep van een niet-void-methode heeft echter de status van een *expressie*. Die expressie staat niet op zichzelf, maar kan gebruikt worden in een opdracht naar keuze, bijvoorbeeld als rechterhelft van een toekenningsoopdracht:

```
double k;
k = this.kwadraat(5);
```

Maar de aanroep van de methode kwadraat kan ook op andere plaatsen worden gebruikt waar double-waarden nodig zijn, bijvoorbeeld als onderdeel van een grotere expressie, of als parameter van een andere aanroep:

```
double oppervl, vierdemacht;
oppervl = Math.PI * this.kwadraat(5);
vierdemacht = this.kwadraat( this.kwadraat(5) );
```

Uiteindelijk zal je die grotere expressie dan toch weer moeten gebruiken in een opdracht (een toekenningsoopdracht, een aanroep van een void-methode, of een **return**-opdracht).

Methodes die elkaar aanroepen

Methodes kunnen elkaar aanroepen. De methode `kwadraat` bijvoorbeeld, komt goed van pas in de methode die de oppervlakte van een cirkel kan uitrekenen:

```
private double oppervlakte(double x)
{
    return Math.PI * this.kwadraat(x);
}
```

Omdat de methoden `oppervlakte` en `kwadraat` zich in dezelfde klasse bevinden, kunnen ze elkaar aanroepen met `this` als betrokken object.

Geen van beide methoden zet een resultaat op het scherm. Dat kunnen ze niet eens, want ze hebben niet de beschikking over een `Graphics`-object, `Label`-object of ander direct lijntje naar de gebruiker. Het opleveren van een resultaatwaarde betekent nog niet automatisch dat die waarde op het scherm verschijnt! Dat moet expliciet gebeuren, bijvoorbeeld door in de event-handler het resultaat van de methode (of eigenlijke de `ToString`-geconverteerde versie daarvan) toe te kennen aan de `Text`-property van een label:

```
uitkomst.Text = this.oppervlakte(straal).ToString();
```

De plaats van de return-opdracht

Voorafgaand aan de `return`-opdracht kunnen nog andere opdrachten staan, die wat voorbereidende berekeningen uitvoeren. Onderstaande methode berekent bij een parameter x de waarde van $(x + 1)^3$:

```
private int derdeMachtVanOpvolger(int x)
{
    int v;
    v = x+1;
    return v*v*v;
}
```

Een `return`-opdracht zou in theorie ook halverwege een methode kunnen staan, en je zou zelfs twee `return`-opdrachten in één methode kunnen zetten:

```
private int raar(int x)
{
    return x*x;
    return x*x*x;
}
```

Maar wat levert deze methode nou op: het kwadraat of de derdemacht van de parameter? Bij de eerste `return`-opdracht wordt de waarde uitgerekend en teruggegeven aan de aanroeper, en daarmee is de methode-aanroep ook meteen beëindigd. Aan het uitrekenen van de derdemacht komt deze methode `raar` helemaal nooit toe.

Hoewel het dus in principe is toegestaan om een `return`-opdracht op een andere plaats dan als laatste in een methode te zetten, is dat in de praktijk zinloos: opdrachten na de `return`-opdracht kunnen nooit worden uitgevoerd. De compiler zal in zo'n geval dan ook waarschuwen dat de methode “unreachable code” bevat.

Het woord “return” kun je op twee manieren vertalen:

- “teruggeven”: de waarde van de expressie in de `return`-opdracht wordt teruggegeven aan de aanroeper;
- “terugkeren”: na het uitvoeren van de `return`-opdracht keert de processor terug naar de aanroeper, ongeacht of er nog meer opdrachten staan na de `return`-opdracht.

Beide vertalingen zijn een adequate beschrijving van de `return`-opdracht.

5.3 Forms ontwerpen met Visual Studio

GUI opbouwen kost veel regels code

Het programma in listing 10 heeft 57 regels. Dat is best veel voor een programma waarvan het hele idee zich laat samenvatten in een wiskundige formule van vijf symbolen: $A = \pi r^2$. Het eigenlijke toepassen van deze formule had in één regel programmacode gekund. Doordat we er eens goed voor zijn gaan zitten om dit in twee aparte methoden-met-resultaat uit te werken zijn dat 8 regels geworden. Daarnaast is er de vaste 10 regels code voor `using`-directieven en klasseheaders. Maar

de resterende code (70% van het programma!) wordt helemaal besteed aan het opbouwen van de grafische userinterface (GUI).

Dat is vervelend veel, want eigenlijk is de opbouw van de GUI niet het interessantste gedeelte van een programma. Daar komt nog bij dat het een heel gepuzzel is om alle **Location**- en **Size**-properties zo in te vullen dat het een beetje mooie interface wordt, en als je het programma terugleest het nog niet eens makkelijk voorstelbaar is hoe die interface er uitziet. Met het plaatje in figuur 13 is dat wel meteen duidelijk.

Visueel vormgeven van de GUI

Om het ontwerpen van de GUI gemakkelijker te maken, is er in de ontwikkelomgeving Visual Studio een faciliteit ingebouwd om dat op een grafische manier te doen: met wat klikken met de muis kun je een GUI in elkaar zetten zonder dat je de saaie code in de constructor zelf hoeft te schrijven. Daarom heet het natuurlijk ook *Visual Studio*.

Deze ‘design mode’ van Visual Studio dringt zich zelfs actief aan je op: zodra je een bestand opent waarin een subklasse van **Form** wordt gedefinieerd krijg je in eerste instantie niet de programmacode van die klasse te zien, maar een grafische weergave van de GUI. Wil je toch de code zien, dan moet je op de F7-toets drukken. Met Shift-F7 ga je weer terug naar design mode.

Gebruik van de Visual Studio GUI-designer

We gaan in deze sectie het programma schrijven dat in figuur 14 in werking te zien is. Met drie schuifregelaars kan de gebruiker een rood-, groen- en blauw-waarde instellen, en op het vlak daarnaast wordt de bijbehorende mengkleur weergegeven. In een label daaronder worden ook de getalswaarden getoond. Verder zijn er drie buttons, waarmee de gebruiker de kleur op wit, grijs of zwart kan zetten. De schuifregelaars bewegen dan overeenkomstig mee.

Vanwege het grotere aantal controls (8) is het de moeite waard om dit programma met hulp van de GUI-designer te schrijven. Dat betekent dat een deel van het programma automatisch wordt gegenereerd. De gegenereerde code kun je desgewenst ook aanpassen. Zo kun je bijvoorbeeld de getallen in de **Position** en **Size** van een control nog een beetje aanpassen. Met de muis is het soms een beetje priegelwerk om dingen netjes recht boven elkaar te krijgen.

In figuur 15 is een snapshot te zien van het ontwerpen van het programma in Visual Studio. Je ziet in een deel-window de GUI van het mixer-programma. Dat is *niet* het programma in werking, maar een ontwerp van de GUI.

In het langwerpige deel-window in het midden staan alle mogelijke controls opgesomd. Daaruit kun je nieuwe controls naar het GUI-ontwerp toe slepen.

Met de muis kun je een control in de GUI aanwijzen en verplaatsen of vergroten/verkleinen. In de afbeelding is dat gebeurd met de onderste schuifregelaar, die zich verraadt door de ‘oortjes’ links en rechts. Daarmee kun je de control groter en kleiner maken.

Van de op een bepaald moment geselecteerde control kun je in een apart deel-window de properties bekijken en zo nodig aanpassen. Voor elke property met een niet-standaard waarde wordt uiteindelijk een regel C#-code gegenereerd met een toekenningsopdracht aan de betreffende property van de control. In het property-window heb je de mogelijkheden handig op een rijtje, en bovendien word je geholpen met de mogelijke waarden (bijvoorbeeld dat de **Orientation** van een schuifregelaar **Horizontal** of **Vertical** kan zijn).

Gegenereerde en zelfgeschreven controls

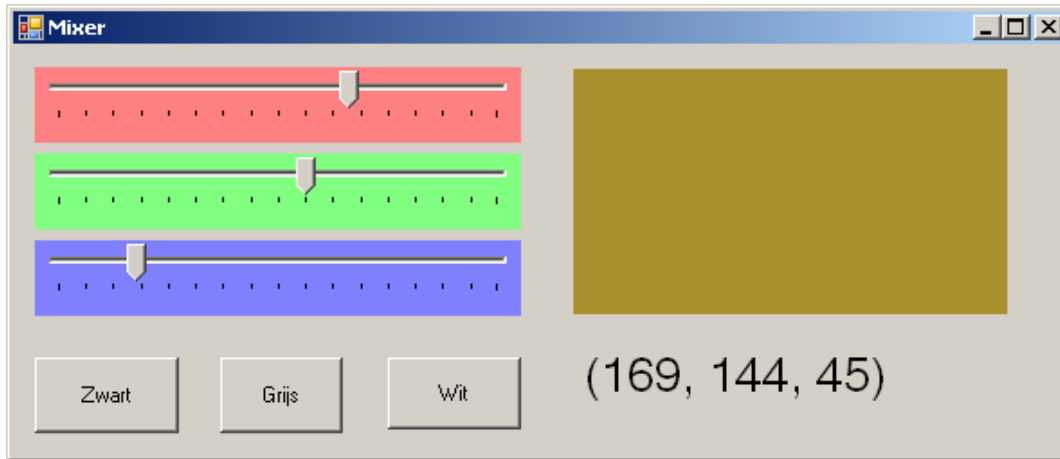
Je kunt in principe een al (gedeeltelijk) bestaand programma visueel ontwerpen. Omgekeerd kun je van de visueel ontworpen GUI de programmacode nog aanpassen, en eventueel daarna weer verder gaan met het visueel ontwerp.

Bij het in de programmatekst aanpassen van het ontwerp moet je je wel enigzins conformeren aan de structuur die de visuele designer eist. In het eerdere programma *CirkelCalc* is dat niet gebeurd. De visuele designer herkent in dat programma weliswaar dat we een subklasse van **Form** definiëren, en biedt aan om het werkvlak visueel met controls te vullen. Maar de vier al bestaande controls worden niet getoond, omdat die op een te eigenwijze manier zijn gemaakt.

De visuele designer wil zich niet met de inhoud van de constructormethode bemoeien. Dat is een goede zaak, want zo kunnen we er zeker van zijn dat de visuele designer niet aan de haal gaat met andere (niet met controls te maken hebbende) code die in de constructor staat. In plaats daarvan zet de visuele designer alle initialisaties van controls in een methode **InitializeComponent**, die dan vanuit de constructor moet worden aangeroepen. Deze methode heeft geen parameters en

void resultaat.

Zouden we dat in het CirkelCalc programma ook gedaan hebben, dan had de visuele designer de zelfgeschreven controls wel kunnen herkennen.



Figuur 14: Het programma Mixer in werking

Raamwerk voor een Forms-programma

Als je in Visual Studio aan een nieuw project begint kun er voor kiezen om het raamwerk van een programma alvast neer te zetten. Kies je hier voor een 'Windows Forms Application', dan krijg je het raamwerk voor een forms-applicatie kado.

Deze aanpak is gevolgd bij het maken van het programma Mixer. Met de visuele designer is binnen het raamwerk de GUI ontworpen, en daarna is de rest van het programma met de hand geschreven. De opbouw van het programma die in het raamwerk wordt gehanteerd is afgebeeld in figuur 16. Net als in programma CirkelCalc zijn er twee klassen. Beide klassen zitten ingebed in dezelfde namespace, die dezelfde naam heeft gekregen als het project: Mixer.

In de klasse `Program` zit alleen de verplichte methode `Main`. Die doet weinig meer dan het aanmaken van een object van de andere klasse (`Mixer`) om dat vervolgens mee te geven aan `Run`. Omdat de klasse uitsluitend statische methoden bevat, is de hele klasse ook van het bijscript `static` voorzien. Deze klasse staat in een eigen file `Program.cs`.

De andere klasse, die ook de naam `Mixer` krijgt, is interessanter. Dit is een subklasse van `Form` die in elk forms-programma een rol speelt. In de constructor wordt de methode `InitializeComponent` aangeroepen, om het programma geschikt te maken voor de visual designer. Bijzonder aan deze klasse is dat hij is gesplitst in twee aparte files: `Mixer.cs` en `Mixer.Designer.cs`.

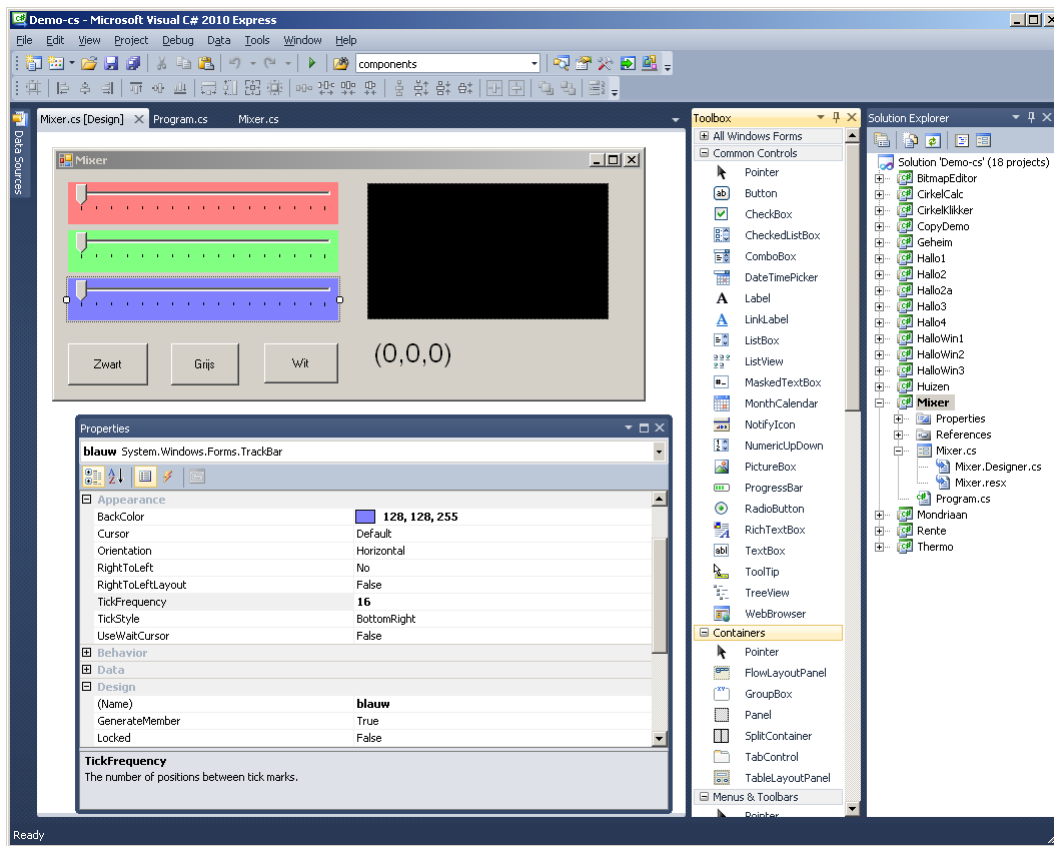
Partial klassen: één klasse verdeeld over meerdere files

Een klasse mag over meerdere files verdeeld worden, als je in de header aangeeft dat elk gedeelte slechts een *partial* klasse is. Dat is precies wat er in dit raamwerk is gedaan. De reden hiervoor is dat alle code die met de hand geschreven wordt in een aparte file terecht kan komen (`Mixer.cs`), en de code die door de visuele designer wordt gegenereerd in een andere file (`Mixer.Designer.cs`). In de gegenereerde file komt de methode `InitializeComponent` te staan.

Deze verdeling over twee files is handig, want nu word je als programmeur niet steeds met de gegenereerde code geconfronteerd, en zul je die niet snel per ongeluk beschadigen. Omgekeerd is het een prettig idee dat de visual designer met zijn vingers van de zelfgeschreven code afblijft. Conceptueel blijft dit echter wel één klasse. De members die in een van de helften staan, zijn dan ook zonder meer bruikbaar in de andere helft. Je ziet dat bijvoorbeeld aan het feit dat vanuit de constructor de methode `InitializeComponent` kan worden aangeroepen.

Events in de visuele designer

De visuele designer biedt ook faciliteiten voor het definiëren van event-handlers. Event-properties zijn natuurlijk minder visueel voorstelbaar dan de afmeting of de kleur van een control, maar toch is het nog wel handig om hiervoor de hulp van de designer in te roepen.



Figuur 15: Ontwerp van de Mixer in Visual Studio

Program.cs	Mixer.cs	Mixer.Designer.cs
<pre> namespace Mixer { static class Program { static void Main() { Application.Run (new Mixer()); } } } </pre>	<pre> namespace Mixer { partial class Mixer : Form { // constructor Mixer() { this.InitializeComponent(); } // event-handlers void klik(object o, EventArgs ea) { } } } </pre>	<pre> namespace Mixer { partial class Mixer { #region generated code void InitializeComponent() { this.b = new Button(); b.Text = "zwart"; b.Click += this.klik; this.Controls.Add(this.b); } #endregion private Button b; } } </pre>

Figuur 16: Het raamwerk van een forms-applicatie: drie files met daarin twee klassen

Kijk nog eens in figuur 15. In het deel-window ‘properties’ kun je de properties van de geselecteerde control wijzigen. De events staan daar in eerste instantie niet bij, maar dat verandert als je op het icoon met de bliksemflits, bovenin het properties-window, klikt. Een control heeft al snel een twintigtal verschillende events waarop je mag reageren. Dubbelklik op de naam van de event, en er worden automatisch twee stukjes code gegenereerd:

- een vooralsnog lege event-handler-methode, met een naam als `zwart_Click` als het om het `Click`-event van de button genaamd `zwart` gaat
- een regel code bij de initialisatie van de control, waarbij de methode abonnee gemaakt wordt van het event:

```
zwart.Click += zwart_Click;
```

De event-handler methode komt in de file met zelfgeschreven code terecht, want het is de bedoeling dat de programmeur de body van deze methode gaat invullen. De initialisatie komt in de methode `InitializeComponent` terecht in de partial class waar de programmeur normaalgesproken afblijft.

De event-handlers van het Mixer-programma

blz. 72

In listing 11 staat de listing van het zelfgeschreven deel van het programma. Het bestaat voornamelijk uit event-handlers, die zijn aangemaakt met bovenbeschreven procedure. Zo is de methode `rood_Scroll` de handler van het `Scroll`-event van de schuifregelaar met de naam `rood`. We reageren op dit event met het aanroepen van de methode `toonKleur`. Daarin krijgt de control die de mengkleur moet tonen de juiste `BackColor` toegekend, en worden de numerieke waarden van de schuifregelaars ook op een label getoond.

De andere twee schuifregelaars hebben zo op het oog geen eigen event-handler. Toch moet de kleur natuurlijk ook getoond worden als die schuifregelaars veranderen. Deze schuifregelaars maken echter mede gebruik van de event-handler `rood_Scroll`. In de visuele designer konden we in plaats van het nieuw aanmaken van een event-handler ook een van de bestaande event-handlers uit een lijstje kiezen.

Voor de drie buttons hebben we wel drie aparte event-handlers laten genereren: `zwart_Click`, `grijs_Click` en `wit_Click`. Als reactie moeten de drie schuifregelaars op een bepaalde waarde worden geforceerd (0 voor zwart, 128 voor grijs, en 255 voor wit), waarna bovendien de nieuwe kleur getoond moet worden. Omdat de benodigde code voor de drie knoppen bijna hetzelfde is (alleen de gewenste waarde verschilt), hebben we een methode `reset` gemaakt die dit werk kan opknappen voor een willekeurige, als parameter mee te geven waarde. Dit voorkomt dat de vier regels code in elk van de drie event-handlers helemaal uitgeschreven moeten worden. De aanwezigheid van de methode `toonKleur` komt nu ook weer goed van pas.

De initialisatie van het Mixer-programma

blz. 73

In listing 12 staat de listing van het gegenereerde deel van het programma. Het bestaat vooral uit opgeklopte lucht: het begint met de declaratie van een variabele `components` die niemand gebruikt, en een methode `Dispose` die niet veel zinvols doet. Maar daarna wordt het belangrijker: de definitie van methode `InitializeComponent`. Hier worden de acht controls aangemaakt en van properties voorzien. De methode is te lang voor 1 bladzijde in dit boek; in listing 14 staat een stukje, en de rest is meer van hetzelfde.

blz. 74

blz. 73

Helemaal aan het eind van de klasse, weergegeven in listing 13, wordt het nog weer even interessant: hier staan de member-variabelen gedefinieerd voor alle controls. In het `CirkelCalc` programma hadden we alleen de controls die ook echt ergens buiten de constructor werden gebruikt als zelfstandige member-variabele gedeclareerd, en de rest lokaal in de constructor gehouden. De designer heeft wat minder scrupules, en declareert ze allemaal als member-variabele.

Korte en lange klasse-namen

Opvallend in de gegenereerde code is dat er niet simpelweg `Button` wordt geschreven, maar dat elke keer de naam van de library er helemaal bij staat: `System.Windows.Forms.Button`. In dit geval betekent de punt dus: ‘pak een onderdeel van een namespace’, niet te verwarren met ‘pak een member van een object’ of ‘pak een statische member van een klasse’.

Dit heeft als voordeel dat de library dan niet bovenin het programma met een `using`-directief hoeft te worden aangekondigd. Het nadeel is natuurlijk dat het meer schrijfwerk is, en vooral: dat het programma saaier wordt om te lezen, en de echt belangrijke gedeelten minder goed opvallen. Voor

robots, zoals onze designer, spelen die nadelen niet zo sterk, maar dit is niet een programmeerstijl om te gaan imiteren.

Lees en huiver ook bij het aantal overbodige paren haakjes en overbodige extra casts ('maak er een byte van, o nee, toch maar een int') die de robot zich permitteert in de parameters bij de aanroep van `FromRGB`.

Pragmas

Er zijn nog twee niet eerder besproken notaties in de gegenereerde code. Ten eerste wordt de gegenereerde methode met initialisaties ingeklemd tussen de regels

```
#region generated code  
  
en  
  
#endregion
```

Regels die met een hekje beginnen worden, net als commentaar, door de compiler gegenereerd. Ze zijn bedoeld als pragmatische aanwijzingen (kortweg: *pragma's*) voor externe tools. In dit geval voor de in Visual Studio ingebouwde editor: de region kan in de editor in zijn geheel worden in- en uitgeklaapt, zodat je bij het lezen van de listing de oninteressante details gemakkelijk kunt wegklikken.

Verder valt het op dat commentaar-regels soms met *drie* schuine strepen beginnen, terwijl twee ook al genoeg zou zijn. Tja, al schrijf je er tien, het blijft commentaar en wordt door de compiler hoe dan ook genegeerd, dus als die robot het nou mooi vindt om er drie te schrijven... Maar drie commentaar-strepen zijn voor sommige externe tools een indicatie dat het commentaar voldoet aan bepaalde opmaak-conventies die het semi-automatisch genereren van documentatie faciliteren. Meer precies: de rol van het commentaar (algemene toelichting op de methode, rol van een specifieke parameter) wordt met XML-tags zoals `<summary>` en `param` expliciet gemaakt.

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace Mixer
{
    public partial class Mixer : Form
    {
        public Mixer()
10     {
            InitializeComponent();
        }

        private void rood_Scroll(object sender, EventArgs e)
15     {
            this.toonKleur();
        }

        private void zwart_Click(object sender, EventArgs e)
20     {
            this.reset(0);
        }
        private void grijs_Click(object sender, EventArgs e)
        {
25         this.reset(128);
        }
        private void wit_Click(object sender, EventArgs e)
        {
30         this.reset(255);
        }

        private void reset(int waarde)
        {
            rood.Value = waarde;
35         groen.Value = waarde;
            blauw.Value = waarde;
            this.toonKleur();
        }
        void toonKleur()
40     {
        mengkleur.BackColor = Color.FromArgb(rood.Value, groen.Value, blauw.Value);
        cijfers.Text = "(" + rood.Value + ", " + groen.Value + ", " + blauw.Value + ")";
        }
    }
45 }
```

Listing 11: Mixer/Mixer.cs

```

namespace Mixer
{
    partial class Mixer
    {
        5      /// <summary>
            /// Required designer variable.
            /// </summary>
            private System.ComponentModel.IContainer components = null;

        10     /// <summary>
            /// Clean up any resources being used.
            /// </summary>
            /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
            protected override void Dispose(bool disposing)
        15     {
                if (disposing && (components != null))
                {
                    components.Dispose();
                }
        20     base.Dispose(disposing);
    }

    #region Windows Form Designer generated code

        25     /// <summary>
            /// Required method for Designer support - do not modify
            /// the contents of this method with the code editor.
            /// </summary>
            private void InitializeComponent()
        30     {
                this.zwart = new System.Windows.Forms.Button();

```

Listing 12: Mixer/Mixer.Designer.cs, deel 1 van 4

```

    }

145     #endregion

    private System.Windows.Forms.Button zwart;
    private System.Windows.Forms.TrackBar rood;
    150    private System.Windows.Forms.TrackBar groen;
    private System.Windows.Forms.TrackBar blauw;
    private System.Windows.Forms.Panel mengkleur;
    private System.Windows.Forms.Label cijfers;
    private System.Windows.Forms.Button wit;
    155    private System.Windows.Forms.Button grijs;
}
}

```

Listing 13: Mixer/Mixer.Designer.cs, deel 4 van 4

```
private void InitializeComponent()
30 {
    this.zwart = new System.Windows.Forms.Button();
    this.rood = new System.Windows.Forms.TrackBar();
    this.groen = new System.Windows.Forms.TrackBar();
    this.blauw = new System.Windows.Forms.TrackBar();
35 this.mengkleur = new System.Windows.Forms.Panel();
    this.cijfers = new System.Windows.Forms.Label();
    this.wit = new System.Windows.Forms.Button();
    this.grijs = new System.Windows.Forms.Button();
    ((System.ComponentModel.ISupportInitialize)(this.rood)).BeginInit();
40 ((System.ComponentModel.ISupportInitialize)(this.groen)).BeginInit();
    ((System.ComponentModel.ISupportInitialize)(this.blauw)).BeginInit();
    this.SuspendLayout();
    //
    // zwart
45 //
    this.zwart.Location = new System.Drawing.Point(12, 173);
    this.zwart.Name = "zwart";
    this.zwart.Size = new System.Drawing.Size(80, 42);
    this.zwart.TabIndex = 0;
50 this.zwart.Text = "Zwart";
    this.zwart.UseVisualStyleBackColor = true;
    this.zwart.Click += new System.EventHandler(this.zwart_Click);
    //
    // rood
55 //
    this.rood.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)255)))), ((int)(((byte)0)));
    this.rood.Location = new System.Drawing.Point(12, 12);
    this.rood.Maximum = 255;
    this.rood.Name = "rood";
60 this.rood.Size = new System.Drawing.Size(270, 42);
    this.rood.TabIndex = 1;
    this.rood.TickFrequency = 16;
    this.rood.Scroll += new System.EventHandler(this.rood_Scroll);
    //
65 // groen
    //
    this.groen.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)128)))), ((int)(((byte)0)));
    this.groen.Location = new System.Drawing.Point(12, 60);
    this.groen.Maximum = 255;
70 this.groen.Name = "groen";
    this.groen.Size = new System.Drawing.Size(270, 42);
    this.groen.TabIndex = 2;
    this.groen.TickFrequency = 16;
    this.groen.Scroll += new System.EventHandler(this.rood_Scroll);
75 //
    // blauw
    //
    this.blauw.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)128)))), ((int)(((byte)0)));
    this.blauw.Location = new System.Drawing.Point(12, 108);
80 this.blauw.Maximum = 255;
    this.blauw.Name = "blauw";
```

Hoofdstuk 6

Herhaling

Dit hoofdstuk *is* niet een herhaling, maar gaat *over* herhaling in C#, en is dus nieuw!

6.1 De while-opdracht

Opdrachten herhalen

Met behulp van event-handlers kunnen we het programma nu weliswaar laten reageren op de gebruiker, maar nadat de afhandelings-methode is afgelopen, staat het programma toch weer stil (totdat de gebruiker een nieuw event genereert).

Om het programma gedurende langere tijd bezig te houden, zijn erg veel opdrachten nodig. òf: we moeten er voor zorgen dat één (of een paar) opdrachten steeds opnieuw worden uitgevoerd. Dit is mogelijk met een speciale opdracht: de *while-opdracht*.

Een voorbeeld van het gebruik van zo'n while-opdracht is het volgende programma-fragment:

```
public int test()
{
    int x;
    x = 1;
    while (x<1000)
        x = 2*x;
    return x;
}
```

In deze methode staan een declaratie, een toekenningsopdracht, dan zo'n while-opdracht, en tenslotte een return-opdracht. De programma-tekst

```
while (x<1000)
    x = 2*x;
```

is dus één opdracht, bestaande uit een soort header: **while** (x<1000) en een body: **x=2*x;** . De header bestaat uit het woord **while** gevolgd door een *voorwaarde* tussen haakjes; de body is zelf een opdracht: hier een toekenningsopdracht, maar dat had ook bijvoorbeeld een methode-aanroep kunnen zijn.

Bij het uitvoeren van zo'n while-opdracht wordt de body steeds opnieuw uitgevoerd. Dit blijft net zolang doorgaan als dat de voorwaarde in de header geldig is. Daarom heet het ook een while-opdracht: de body wordt steeds opnieuw uitgevoerd *while* de voorwaarde geldt.

In het voorbeeld krijgt de variabele x aan het begin de waarde 1. Dat is zeker kleiner dan 1000, dus wordt de body uitgevoerd. In die body wordt de waarde van x veranderd in zijn eigen dubbele; de waarde van x wordt daarmee gelijk aan 2. Dat is nog steeds kleiner dan 1000, en dus wordt de body nogmaals uitgevoerd, waardoor x de waarde 4 krijgt. Ook dat is kleiner dan 1000, dus nogmaals wordt de waarde van x verdubbeld tot 8. Dat is kleiner dan 1000, en zo doorgaand krijgt x daarna nog de waarden 16, 32, 64, 128, 256 en 512. Dat is kleiner dan 1000, en dus wordt de body weer opnieuw uitgevoerd, waarmee x de waarde 1024 krijgt. En dat is *niet* kleiner dan 1000, waarmee de herhaling eindelijk tot een eind komt.

Pas dan wordt de volgende opdracht uitgevoerd: de return-opdracht die de eindwaarde die x na al dat verdubbelen heeft gekregen (1024) aan de aanroeper van de methode **test** teruggeeft.

Groepjes opdrachten herhalen

Je kunt ook meerdere opdrachten herhalen met behulp van een while-opdracht. Je zet de opdrachten dan tussen accolades, en maakt het hele bundeltje tot body van de while-opdracht. De

methode in het volgende programmafragment bijvoorbeeld, bepaalt *hoe vaak* je het getal 1 kunt verdubbelen totdat het groter dan 1000 wordt:

```
int hoeVaak()
{
    int x, n;
    x = 1; n = 0;
    while (x<1000)
    {
        x = 2*x;
        n = n+1;
    }
    return n;
}
```

We gebruiken hier een variabele **n** om het aantal verdubbelingen te tellen. Voorafgaand aan de while-opdracht is er nog niets herhaald, en daarom maken we **n** gelijk aan 0. Elke keer dat de waarde van **x** in de body verdubbeld wordt, verhogen we ook de waarde van **n**, zodat die variabele inderdaad de tel bijhoudt. Na afloop van de while-opdracht bevat de variabele **n** dan het aantal uitgevoerde herhalingen.

De twee opdrachten die herhaald moeten worden, zijn met accolades samengesmeed tot een blok, die in z'n geheel als body van de while-opdracht geldt. Voorafgaand aan het uitvoeren van het blok wordt de voorwaarde **x<1000** steeds gecontroleerd. Als de voorwaarde geldt, dan wordt het blok in z'n geheel uitgevoerd. Dus ook de laatste keer, als door de toekenning **x** de waarde 1024 heeft gekregen, wordt toch ook nog **n** opgehoogd.

Twee dingen vallen op aan deze programmafragmenten:

- De variabelen die in de body gebruikt worden, moeten voorafgaand aan de herhaling een beginwaarde hebben gekregen
- De voorwaarde die de herhaling controleert moet een variabele gebruiken die in de body wordt veranderd (zo niet, dan is de herhaling òf direct, òf helemaal nooit afgelopen).

Herhalen met een teller

Variabelen die het aantal herhalingen tellen zijn ook heel geschikt om het verdergaan van de herhaling te controleren. Je kunt met zo'n teller een opdracht bijvoorbeeld precies tien keer laten uitvoeren. Deze methode (er van uitgaande dat dit de paint-event-handler van een form is) tekent 10 keer dezelfde bitmap (er van uitgaande dat die beschikbaar is in de variabele **bm**) onder elkaar op het scherm:

```
void tekenScherm(object o, PaintEventArgs pea)
{
    int n;
    n = 0;
    while (n<10)
    {
        pea.Graphics.DrawImage( bm, 0, 50*n );
        n = n+1;
    }
}
```

Behalve om het aantal herhalingen tellen, komt de teller **n** hier ook goed van pas om de positie te bepalen waar het **n**-de plaatje moet worden getekend: de y-coördinaten 0, 50, 100, 150 enzovoorts kunnen eenvoudig uit **n** worden berekend.

Opbouw van een resultaat

Bij een while-opdracht wordt er vaak gedurende de herhaling een resultaat opgebouwd. Een voorbeeld hiervan is de volgende methode, die de *faculteit* berekent van een getal, dat als parameter wordt meegegeven. (De faculteit van een getal is het product van alle getallen tussen 1 en dat getal; bijvoorbeeld: de faculteit van 4 is $1*2*3*4=24$.)

Behalve een teller gebruikt deze methode een variabele **result**, waarin het resultaat langzaam wordt opgebouwd:

```
private static int faculteit(int x)
{
    int n, result;
    n=1; result=1;
    while (n<=x)
```

```

    {   result = result*n;
        n = n+1;
    }
    return result;
}

```

Deze methode kan `static` gemaakt worden, omdat hij behalve de parameter `x` en zijn lokale variabelen geen (member-)variabelen gebruikt, en dus geen object onder handen hoeft te hebben.

6.2 bool waarden

Vergelijkings-operatoren

De voorwaarde in de header van de `while`-opdracht is een expressie, die na berekening een waarheidswaarde oplevert: “ja” of “nee”. De herhaling wordt voortgezet zolang de uitkomst van de berekening “ja” is.

In voorwaarden kun je vergelijkings-operatoren gebruiken. De volgende operatoren zijn beschikbaar:

- `<` kleiner dan
- `<=` kleiner dan of gelijk aan
- `>` groter dan
- `>=` groter dan of gelijk aan
- `==` gelijk aan
- `!=` ongelijk aan

Deze operatoren kunnen worden gebruikt tussen twee getallen, zowel `int`-waarden als `double`-waarden. Links en rechts van de operator mogen constante getallen staan, variabelen, of complete expressies met optellingen en vermenigvuldigingen en dergelijke.

Let er op dat de gelijkheids-test met een dubbel `is`-teken wordt geschreven. Dit moet, omdat het enkele `is`-teken al in gebruik is als toekenningsoopdracht. Het verschil tussen `=` en `==` is erg belangrijk:

```

x=5;   opdracht   maak x gelijk aan 5 !
x==5   expressie   is x op dit moment gelijk aan 5 ?

```

Logische operatoren

Een voorwaarde is wat in de logica een predicaat wordt genoemd. De operatoren die in de logica gebruikt worden om predicaten te verbinden (“en”, “of” en “niet”) kunnen ook in C# gebruikt worden. De mooie symbolen die de logica ervoor gebruikt zitten helaas niet op het toetsenbord, dus we zullen het moeten doen met een ander symbool:

- `&&` is de logische “en”
- `||` is de logische “of”
- `!` is de logische “niet”

Het type bool

Expressies waarin de vergelijkingsoperatoren gebruikt worden, of waarin vergelijkingen met logische operatoren worden gekoppeld, hebben evengoed een type als expressies waarin rekenkundige operatoren gebruikt worden.

De uitkomst van zo’n expressie is immers een waarde: één van de twee waarheidswaarden “ja” of “nee”. Logici noemen deze waarden “waar” en “onwaar”; de gangbare Engelse benamingen zijn “true” en “false”.

Behalve gebruiken als voorwaarde in een `while`-opdracht, kun je allerlei andere dingen doen met logische expressies. Een logische expressie is namelijk net zoiets als een rekenkundige expressie, alleen van een ander type. Je kunt de uitkomst van een logische expressie bijvoorbeeld opslaan in een variabele, of als resultaat laten opleveren door een methode.

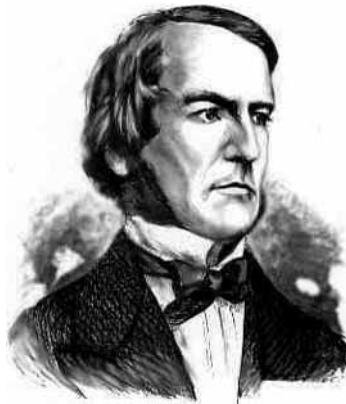
Het type van logische waarden heet `bool`. Dit is een van de primitieve typen van C#. Variabelen van dit type bevatten net als bij de numerieke typen een waarde, dus niet een referentie. Het type is genoemd naar de Engelse logicus George Boole (zie figuur 17).

Een voorbeeld van een declaratie van en toekenning aan een `bool` variabele:

```

bool test;
test = x>3 && y<5;

```



Figuur 17: George Boole (1815-1864)

Wat nuttiger lijkt een methode met een `bool` waarde als resultaat. Bijvoorbeeld een methode die het antwoord oplevert op de vraag of een getal een zevenvoud is:

```
private bool isZevenvoud(int x)
{
    return x%7==0;
}
```

Een getal is een zevenvoud als de rest bij deling door zeven nul is. De uitkomst van de `bool` expressie die deze test uitvoert is het resultaat van de methode. De methode kan vervolgens worden gebruikt als voorwaarde in een `while`-opdracht, om –noem ’ns wat– het eerste 7-voud groter dan 1000 te vinden:

```
n = 1000;
while ( ! this.isZevenvoud(n) )
    n = n+1;
```

Dit voorbeeld maakt duidelijk dat de voorwaarde in een `while`-opdracht niet altijd een vergelijking hoeft te zijn, maar ook een andere `bool` expressie mag zijn; omgekeerd zijn voorwaarden van `while`-opdrachten niet de enige plaats waar vergelijkingen een rol spelen: dit kan ook op andere plaatsen waar een `bool` expressie nodig is.

6.3 De `for`-opdracht

Verkorte notatie van teller-ophoging

In de bodies van veel `while`-opdrachten, vooral die waarin een teller wordt gebruikt, komen opdrachten voor waarin een variabele wordt opgehoogd. Dit kan door middel van de opdracht

```
n = n+1;
```

(Even terzijde: alleen al vanwege dit soort opdrachten is het onverstandig om de toekenning uit te spreken als “is”. De waarde van `n` kan natuurlijk nooit gelijk *zijn* aan `n+1`, maar de waarde van `n` *wordt* gelijk aan de (oude) waarde van `n+1`).

Opdrachten zoals deze komen zo vaak voor, dat er een speciale, verkorte notatie voor bestaat:

```
n++;
```

Een adequate uitspraak voor `++` is “wordt opgehoogd”.

Voor ophoging met meer dan 1 is er nog een andere notatie:

```
n += 2;
```

betekent hetzelfde als

```
n = n+2;
```

Automatisch tellen

Veel `while`-opdrachten gebruiken een tellende variabele, en hebben dus de volgende structuur:

```

int n;
n = beginwaarde ;
while (n < eindwaarde )
{
    doe iets nuttigs gebruikmakend van n
    n++;
}

```

Omdat zo'n “tellende herhaling” zo vaak voorkomt, is er een aparte notatie voor beschikbaar:

```

int n;
for (n=beginwaarde; n<eindwaarde; n++)
{
    doe iets nuttigs gebruikmakend van n
}

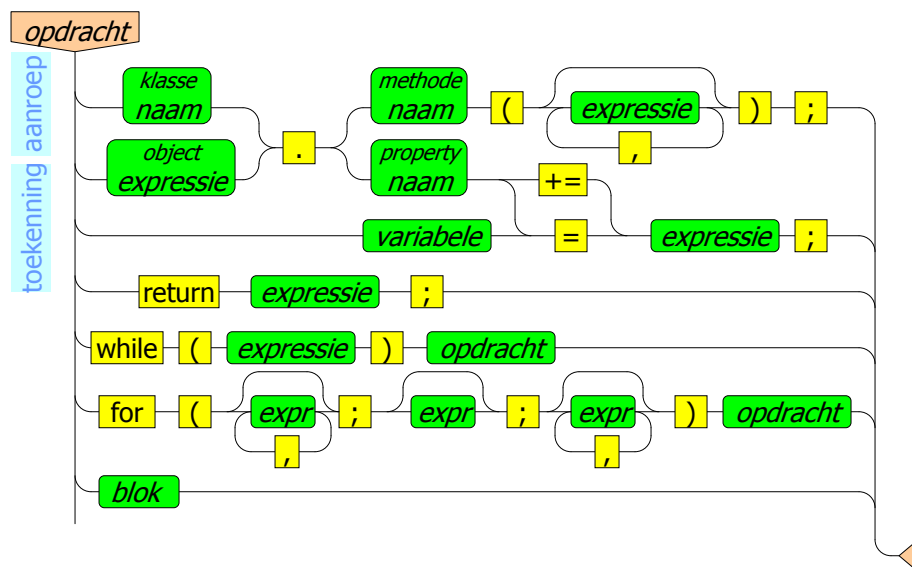
```

De betekenis van deze **for**-opdracht is precies dezelfde als die van de hierboven genoemde **while**-opdracht. Het voordeel is echter dat de drie dingen die met de teller te maken hebben (de beginwaarde, de eindwaarde en het ophogen) netjes bij elkaar staan in de header. Dat maakt de kans veel kleiner dat je het ophogen van de teller vergeet op te schrijven.

In die gevallen waar “doe iets nuttigs” uit maar één opdracht bestaat, kun je de accolades ook nog weglaten, wat de notatie nog iets compacter maakt.

Syntax van while- en for-opdrachten

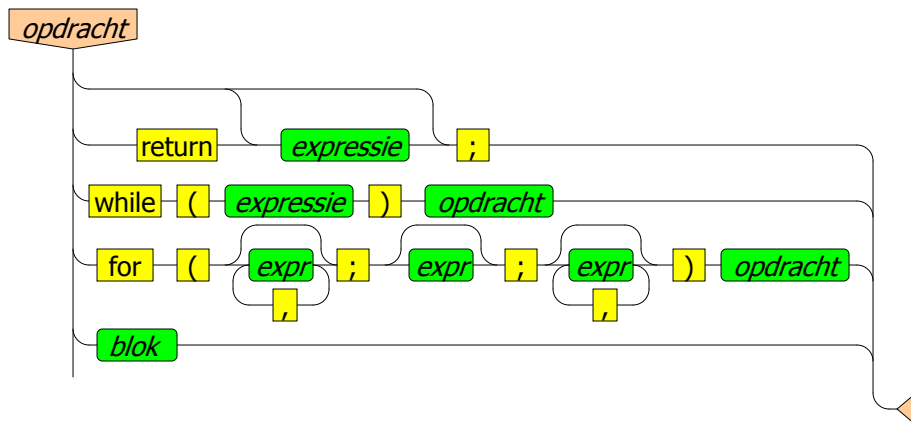
Al met al zijn er drie soorten opdrachten bijgekomen: naast de methode-aanroep, de toekenningsopdracht, en de return-opdracht zijn er nu ook: de while-opdracht, de for-opdracht, en een blok als opdrachtvorm. In dit syntax-diagram worden die samengevat:



Opmerkelijk in dit syntax-diagram is dat in de header van een **for**-opdracht driemaal een **expressie** staat, met twee puntkomma's ertussen. Hoe kan dat nou kloppen? De eerste van de drie is toch een toekenning, zoals **n=0**, en dat is een opdracht en niet een **expressie**! Ook de derde van de drie, iets als **n=n+1** lijkt niet echt een **expressie**.

Toch klopt het syntax-diagram. Technisch gesproken is het ‘wordt’-teken namelijk gewoon een operator, net zoals ‘plus’ en ‘maal’ dat zijn. Het fragment **n=0** (zonder de puntkomma!) is dus een heuse **expressie**. Met de puntkomma erbij wordt het een opdracht. De syntax van ‘opdracht’ is dus eigenlijk veel eenvoudiger: er is geen apart spoor nodig voor een toekenningsopdracht, en ook niet voor methode-aanroep-opdracht: dit zijn beide verschijningsvormen van de **expressie-met-een-puntkomma-erachter-opdracht**. Bij toekenningsopdracht wordt een operator-**expressie** met de bijzondere operator **=** gebruikt, en ook de methode-aanroep is een van de mogelijke **expressies**.

We kunnen dus het syntaxdiagram voor ‘opdracht’ vereenvoudigen: de methode-aanroep en de toekenningsopdracht vervallen, en in plaats daarvan komt de **expressie-met-puntkomma**.



In bijlage E hebben we de toekennings-opdracht en de (void-)methode-aanroep-opdracht er in het syntax-diagram van ‘opdracht’ er toch maar bij laten staan, voor de overzichtelijkheid, maar strikt genomen was dat niet nodig.

Je kunt je afvragen of een expressie zoals `3+4` met een puntkomma er achter dan ook als opdracht gebruikt mag worden. In talen als C en C++ is dat inderdaad het geval, maar in C# is, buiten de syntax om, nader bepaald dat dit niet mag: een expressie-met-puntkomma-opdracht, en de eerste en derde expressie in een for-header, mag alleen een operator-expressie met een toekennings-operator (=, of een variant als += of ++) zijn, of een methode-aanroep (inclusief constructor-methoden). Kortom: zo’n expressie moet een permanent effect (kunnen) hebben.

Uit dit laatste syntax-diagram blijkt ook dat zelfs een losse puntkomma als een opdracht beschouwd kan worden. Dat maakt het mogelijk om ongestraft nog wat extra puntkomma’s neer te zetten (bijvoorbeeld achter de sluit-accolade van een ‘blok’, waar dat eigenlijk helemaal niet vereist is). Maar pas op: achter de *header* van een while-, for- of if-opdracht kun je niet zomaar een puntkomma toevoegen zonder dat de semantiek, soms ingrijpend, verandert!

6.4 Bijzondere herhalingen

Niet-uitgevoerde herhaling

Het kan gebeuren dat de voorwaarde in de header van een while-opdracht meteen aan het begin al onwaar is. Dit is het geval in de volgende opdracht:

```
x=1; y=0;
while (x<y)
    x++;
```

In deze situatie wordt de body van de while-opdracht helemaal niet uitgevoerd, zelfs niet één keer. In het voorbeeld blijft `x` dus gewoon de waarde 1 houden.

Oneindige herhaling

Een gevaar van while-opdrachten is dat er soms nooit een einde aan komt (qua uitvoeren dan, niet qua programmatekst!).

Zo’n opdracht is gemakkelijk te schrijven. Met

```
while (1==1)
    x = x+1;
```

wordt de waarde van `x` steeds maar verhoogd. De voorwaarde `1==1` blijft namelijk altijd “waar”, zodat de opdracht steeds opnieuw uitgevoerd wordt.

In dit programma was die oneindige herhaling wellicht de bedoeling, maar vaak slaat een while-opdracht ook op hol als gevolg van een programmeerfout.

Bijvoorbeeld in:

```
x = 1;
aantal = 0;
while (aantal<10)
    x = x*2;
    aantal = aantal+1;    // fout!
```


Het is de bedoeling dat de waarde van `x` tienmaal wordt verdubbeld. Helaas heeft de programmeur vergeten om de twee opdrachten van de body tussen accolades te zetten. De bedoeling wordt wel gesuggereerd door de lay-out, maar daar heeft de compiler geen boodschap aan. Daardoor wordt alleen de opdracht `x=x*2;` herhaald, en op die manier wordt de waarde van `aantal` natuurlijk nooit groter of gelijk aan 10. Na afloop van de while-opdracht zou de opdracht `aantal=aantal+1;` éénmaal worden uitgevoerd, maar daaraan komt de computer niet eens meer toe.

De bedoeling van de programmeur was natuurlijk:

```
while (aantal<10)
{
    x = x*2;
    aantal = aantal+1;    // goed.
}
```

Het zou jammer zijn als je, na een computer met een vergeten accolade in coma gebracht te hebben, het dure apparaat weg zou moeten gooien omdat hij steeds maar bezig blijft met dat ene programma. Gelukkig kan het operating system met geweld de uitvoering van het programma beëindigen, ook al is het nog niet voltooid. Het programma wordt dan direct gestopt, en je kunt de oorzaak van het “hangen” van het programma gaan zoeken.

In het algemeen moet je, als het programma bij het uittesten niets lijkt te doen, de while-opdrachten in je programma nog eens kritisch bekijken. Een beruchte fout is het vergeten van het ophogen van de tellende variabele, waardoor de bovengrens van de telling nooit wordt bereikt, en de herhaling dus steeds maar doorgaat.

Herhaalde herhaling

De body van een while-opdracht en van een for-opdracht is zelf óók weer een opdracht. Dat kan een toekenningsopdracht zijn, of een methode-aanroep, of een met accolades gebouwde samengestelde opdracht. Maar de body kan ook zelf weer een while- of for-opdracht zijn. Bijvoorbeeld:

```
int x, y;
for (y=0; y<10; y++)
    for (x=0; x<y; x++)
        pea.Graphics.DrawString("+", font, brush, 20*x, 20*y);
```

In dit fragment telt de variabele `y` van 0 tot 10. Voor elk van die waarden van `y` wordt de body uitgevoerd, en die bestaat zelf uit een herhaling, gecontroleerd door de teller `x`. Deze teller heeft als bovengrens de waarde van `y`. Daardoor zal de “binnenste” herhaling, naarmate `y` groter wordt, steeds langer doorgaan. De opdracht die herhaald herhaald wordt, is het tekenen van een plus-symbool op posities evenredig met `x` en `y`. Het resultaat is een driehoek-vormig tableau van plus-tekens:

```
+
+ +
+ + +
+ + + +
+ + + + +
+ + + + + +
+ + + + + + +
+ + + + + + + +
+ + + + + + + + +
```

Op de bovenste rij in dit tableau staan nul plus-tekens. De waarde van `y` is op dat moment nog 0, en de eerste keer dat de for-`x`-opdracht wordt uitgevoerd, betreft het een herhaling die nul keer wordt uitgevoerd. Zo’n niet-uitgevoerde herhaling past hier prima in de regelmaat van het schema.

6.5 Toepassing: renteberekening

Rente op rente

Een aantal besproken ideeën komt samen in het programma die te zien is in listing 15 en figuur 18 (voor respectievelijk de programmatekst en het runnende programma). De GUI is gemaakt met de

```

using System;
using System.Windows.Forms;

namespace Rente
5 {
    public partial class Rente : Form
    {
        public Rente()
        {
10            InitializeComponent();
            bedragTekst.Text = "100";
            renteTekst.Text = "5";
        }

15    private void rekenKnop_Click(object sender, EventArgs e)
    {
        double bedrag = double.Parse(bedragTekst.Text);
        double rente = double.Parse(renteTekst.Text);
        resultaat.Text = "";
20        int jaar;
        for (jaar = 0; jaar <= 10; jaar++)
        {
            resultaat.Text += "Na " + jaar + " jaar: " + bedrag + "\n";
            bedrag *= (1 + 0.01 * rente);
25        }
    }
    // gegenereerde declaraties in de andere helft van de partial class:
    // TextBox bedragTekst;
    // TextBox renteTekst;
30    // Button rekenKnop; met rekenKnop_Click als Click-event-handler
    // Label resultaat;
}
}

```

Listing 15: Rente/Rente.cs

visuele designer van Visual Studio; de listing toont alleen de partial klasse met het handgeschreven deel.

Dit programma laat de gebruiker een bedrag en een rentepercentage invoeren, en toont dan de ontwikkeling van het kapitaal (of als je wilt de schuld...) in de komende tien jaren.

Door het effect van “rente op rente” komt er niet elk jaar een vast bedrag bij, maar stijgt het kapitaal/de schuld steeds sterker. De vermeerdering van het bedrag wordt beschreven door de opdracht

```
bedrag *= (1 + 0.01*rente);
```

De hierin gebruikte operator `*` heeft de betekenis “wordt vermenigvuldigd met”, net zoals `+=` de betekenis heeft “wordt vermeerderd met”. Deze opdracht is een verkorte schrijfwijze voor

```
bedrag = bedrag * (1 + 0.01*rente);
```

Bij een rentepercentage van 5 wordt het bedrag door middel van deze opdracht vermenigvuldigd met 1.05.

In een for-opdracht wordt de opdracht elfmaal uitgevoerd, en daaraan voorafgaand wordt steeds het tussenresultaat aan de `Text` van een label toegevoegd.



Figuur 18: De applet Rente in werking

Invoer via een TextBox

De gebruiker kan zelf het te gebruiken startbedrag en het rentepercentage invullen. Dat is mogelijk met twee controls van het type `TextBox`. De GUI is met de visual designer gemaakt; de methode `InitializeComponent` die verantwoordelijk is voor de aanmaak van de controls staat niet in de listing.

In de eventhandler van de button, `rekenknop_Click`, pakken we de `Text`-properties van de textboxes, en converteren de daarmee verkregen strings meteen (zonder ze eerst in variabelen op te slaan!) met de methode `double.Parse` tot getallen. Met die getallen gaat de for-opdracht vervolgens aan de slag. De resultaten worden met de woorden 'Na' en 'jaar' tot een leesbaar zinnetje gevormd, en al die zinnetjes worden steeds achter de tekst op de `resultaat`-label geplakt. Let op het gebruik van de operator `+=`, die steeds een string aan de eerdere string toevoegt.

Doordat elk zinnetje ook op het speciale newline-teken `"\n"` eindigt, komen de zinnetjes onder elkaar op de label te staan.

Hoofdstuk 7

Keuze

7.1 De if-opdracht

Opdrachten voorwaardelijk uitvoeren

Opdrachten in een programma worden normaal gesproken de één na de ander uitgevoerd. Met een *while*-opdracht kun je er eens een paar herhalen, maar daarna gaat het (als de herhaling tenminste niet oneindig lang doorgaat) onverbiddelijk verder. Niet altijd is dat gewenst: soms moeten opdrachten alleen maar onder bepaalde omstandigheden worden uitgevoerd. Die omstandigheden kunnen afhangen van wat er voorafgaand in het programma is gebeurd, en hangen uiteindelijk af van invoer die de gebruiker heeft verstrekt.

Stel bijvoorbeeld dat in het programma de variabele `temperatuur` een waarde heeft gekregen, bijvoorbeeld doordat de gebruiker dat via een tekstveld heeft ingevoerd. Met een speciale opdrachtvorm kunnen we het programma nu een opmerking over het weer laten maken, maar dan alleen als dat toepasselijk is:

```
if (temperatuur<0)
    uitvoer.Text = "Het vriest!";
```

De opbouw van deze *if-opdracht* lijkt op die van een *while*-opdracht: er is een header met een voorwaarde, en een opdracht die de body vormt. De opdracht in de body wordt alleen uitgevoerd als de voorwaarde waar is, anders wordt-ie overgeslagen.

Een tweede alternatief achter else

Mocht het nodig zijn om in het andere geval, dus als de voorwaarde in de header juist niet waar is, een andere opdracht uit te voeren, dan kun je de *if*-opdracht uitbreiden met een *else*-gedeelte. Bijvoorbeeld:

```
if (temperatuur<0)
    uitvoer.Text = "Het vriest!";
else
    uitvoer.Text = "Het dooit.";
```

Let wel, het geheel “*if*-voorwaarde-opdracht-*else*-opdracht” heeft zelf de status van één opdracht. Het geheel, met of zonder *else*-gedeelte, kan dus zelf optreden als bijvoorbeeld de body van een *for*-opdracht, zonder dat er accolades nodig zijn:

```
for (n=1; n<20; n++)
    if (n%3==0)
        uitvoer.Text += n + " is deelbaar door 3";
    else
        uitvoer.Text += n + " is niet deelbaar door 3";
```

Groepjes opdrachten voorwaardelijk uitvoeren

Als je meerdere opdrachten voorwaardelijk wilt uitvoeren, dan kun je die net als bij de *while*-opdracht groeperen met accolades. Bijvoorbeeld:

```
if (temperatuur<0)
{
    uitvoer1.Text = "Het vriest";
    uitvoer2.Text = "Koud he!";
}
```

Ook de opdracht achter *else* kan een met accolades samengebundelde groep opdrachten zijn.

Een reeks alternatieven

Als er meerdere categorieën van waarden zijn, dan kun je met if-opdrachten uittesten welk geval zich voordoet. De tweede test komt te staan achter de **else** van de eerste if-opdracht, zodat de tweede test alleen maar wordt uitgevoerd als de eerste test mislukt is. Een eventuele derde test komt te staan achter de **else** van de tweede if-opdracht.

Het volgende fragment bijvoorbeeld bepaalt tot welke tarief-categorie iemand met een bepaalde leeftijd behoort, en verandert de **Text**-property van een label **uitvoer** om dat aan de gebruiker duidelijk te maken:

```
if (leeftijd<4)
    uitvoer.Text = "Gratis";
else if (leeftijd<12)
    uitvoer.Text = "Railrunner";
else if (leeftijd<65)
    uitvoer.Text = "Vol tarief";
else uitvoer.Text = "Seniorenkaart";
```

Achter elke **else** (behalve de laatste) staat opnieuw een if-opdracht. Voor babies wordt de tekst “Gratis” getoond, en wordt de hele rest overgeslagen (die staat immers achter de **else**). Bejaarden daarentegen, doorlopen alle tests (kleiner dan 4? kleiner dan 12? kleiner dan 65?) voordat we tot een “Seniorenkaart” concluderen.

In het programma is met inspringen duidelijk aangegeven welke **else** bij welke **if** hoort. Bij lange reeksen tests gaat de tekst van het programma dan wel erg naar rechts hangen. Als uitzondering op onze gewoonte om deel-opdrachten achter **else** naar rechts in te springen, zullen we bij zo’n herhaalde if-opdracht de lay-out iets simpeler houden:

```
if (leeftijd<4)
    uitvoer.Text = "Gratis";
else if (leeftijd<12)
    uitvoer.Text = "Railrunner";
else if (leeftijd<65)
    uitvoer.Text = "Vol tarief";
else uitvoer.Text = "Seniorenkaart";
```

Dat is ook wel mooi, want dan zie je alle alternatieven netjes op een rijtje staan.

Stop als gevonden

De if-opdracht kan natuurlijk ook in een methode staan. Je kunt dan de return-opdracht, waarmee een methode zijn resultaat bekendmaakt, voorwaardelijk maken. Hier is het treintarief-voorbeeld nog eens, maar nu in methode-vorm:

```
private static string tarief(int leeftijd)
{
    if (leeftijd<4)
        return "Gratis";
    if (leeftijd<12)
        return "Railrunner";
    if (leeftijd<65)
        return "Vol tarief";
    return "Seniorenkaart";
}
```

Omdat de return-opdracht direct terugkeert naar de aanroeper van de methode, is het hier niet eens nodig om de tweede test achter “else” te schrijven. Als de eerste test waar is, dan komt de methode niet eens toe aan de tweede test.

In sectie 5.2 noemden we het nog zinloos om meer dan één return-opdracht in een methode te schrijven, omdat de tekst achter de eerste return-opdracht “unreachable code” zou zijn. In dit geval is het echter wel zinvol, omdat de eerste return-opdracht niet *altijd* wordt uitgevoerd, en de rest van de code dus onder sommige omstandigheden wel reachable is.

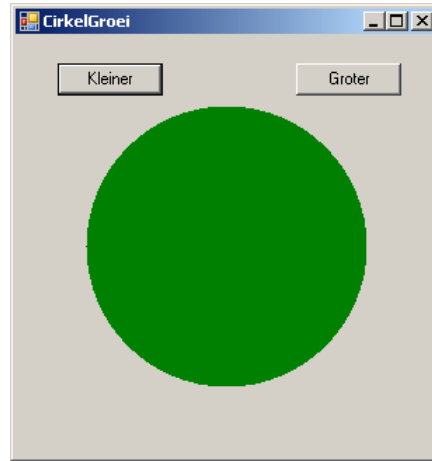
blz. 64

7.2 Voorbeelden van if-opdrachten

Onderscheiden van Buttons

In listing 16 en figuur 19 staan (tekst en snapshot) van een programma dat een groene cirkel op het scherm tekent. Er zijn twee buttons met opschrift “kleiner” en “groter”. Met deze knoppen

blz. 87



Figuur 19: Het programma CirkelGroeï in werking

kan de gebruiker de cirkel kleiner en groter maken, mits hij niet onzichtbaar wordt of buiten het window gaat vallen.

Omdat de GUI erg eenvoudig is, is het programma helemaal met de hand geschreven, dus zonder hulp van de visuele designer. Zelfs de methode `Main` staat in dezelfde klasse als de rest, en daarom is er ook geen `namespace` nodig. De opzet van het programma is zoals ieder interactief programma: de constructor creëert de nodige controls, er is een event-handler `klik` die gekoppeld is aan het `Click`-event van de buttons, en een methode `tekenScherm` die gekoppeld is aan het `Paint`-event van de hele form.

Variabelen voor de twee buttons zijn als membervariabelen gedeclareerd, omdat deze variabelen zowel in de constructor als in `klik` nodig zijn. Ditmaal is er bovendien een `int`-variabele (`straal`) als membervariabele gedeclareerd. Deze variabele is in alle drie de methoden nodig:

- in de constructor krijgt `straal` zijn beginwaarde
- in `teken` wordt `straal` gebruikt om de grootte van de cirkel te bepalen
- in `klik` wordt `straal` groter of kleiner gemaakt, afhankelijk van welke button is ingedrukt.

Voor dat laatste komt een `if`-opdracht goed van pas. We gebruiken bovendien iets nieuws: de eerste parameter van de event-handler. Dat is een object van het type `object`, die het object bevat dat het event heeft veroorzaakt.

Met behulp van twee `if`-opdrachten wordt getest of het bewuste object de “groter”- of de “kleiner”-button is. Door middel van een tweede voorwaarde, achter de logische “and”-operator, wordt bovendien gecontroleerd dat de straal niet té groot, respectievelijk klein wordt.

Je ziet dat je met de `==` operator behalve getallen ook objecten, of liever gezegd objectverwijzingen, kunt testen op gelijkheid. Ook de ongelijkheids-operator `!=` werkt op objecten. Je kunt objecten echter niet ordenen met operatoren zoals `<` en `>=`.

Nadat de straal is veranderd, roepen we de methode `Invalidate` aan. Dit heeft tot gevolg dat `this`, dat is dus het hele form, opnieuw getekend zal worden, en omdat we op dat event zijn geabonneerd zal uiteindelijk de methode `teken` worden aangeroepen.

Controle van passwords

In listing 17 staat de tekst van een programma dat een mooie tekening maakt, maar alleen als de gebruiker eerst het juiste password intikt in een daarvoor bestemde tekstbox.

Als membervariabelen hebben we behalve een variabele voor de tekstbox nog twee variabelen. Er is een string waarin de sleutel tot de toegang wordt bewaard, en er is een bool-variabele `open`, die aangeeft of het “slot” al is opengemaakt. Meteen bij de declaratie krijgt de sleutel-string zijn waarde (maar die is geheim, dus die verklappen we hier niet), en krijgt de variabele `open` de waarde `false`, omdat het slot bij de start van het programma nog niet is opengemaakt.

De eventhandler `veranderd` wordt aangeroepen als de gebruiker iets in de tekstbox heeft veranderd. In die methode gaan we controleren of de gebruiker het password goed heeft geraden. Als dat zo is,

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 public class CirkelGroei : Form
{
    private Button kleiner, groter;
    private int straal;

10 public CirkelGroei()
{
    this.straal = 100;
    this.ClientSize = new Size(300, 300);
    this.Text = "CirkelGroei";
15 this.Paint += this.teken;
    this.kleiner = new Button();
    this.groter = new Button();
    this.kleiner.Text = "Kleiner";
    this.groter.Text = "Groter";
20 this.kleiner.Location = new Point( 30, 20);
    this.groter.Location = new Point(200, 20);
    this.kleiner.Click += this.klik;
    this.groter.Click += this.klik;
    this.Controls.Add(this.kleiner);
25 this.Controls.Add(this.groter);
}

void klik(object sender, EventArgs e)
{
30 if (sender == this.kleiner && this.straal > 10)
    this.straal -= 10;
    if (sender == this.groter && this.straal < 150)
        this.straal += 10;
    this.Invalidate();
35 }

void teken(object obj, PaintEventArgs pea)
{
    pea.Graphics.FillEllipse( Brushes.Green
40                             , 150-this.straal, 150-this.straal
                             , 2 * this.straal, 2 * this.straal);
}

static void Main()
45 {
    CirkelGroei cg = new CirkelGroei();
    Application.Run(cg);
}
}
```

Listing 16: CirkelGroei/CirkelGroei.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace Geheim
{
    public class Geheim : Form
    {
        private TextBox password;
        10 private bool open = false;
        private const string sleutel = "geheim";

        public Geheim()
        {
            15 this.Text = "Geheim";
            this.Paint += tekenScherm;
            this.password = new TextBox();
            this.password.Location = new Point(65, 15);
            this.password.Size = new Size(136, 20);
            20 this.password.PasswordChar = '*';
            this.password.TextChanged += veranderd;
            this.Controls.Add(this.password);
        }

        25 void veranderd(object sender, EventArgs e)
        {
            if (this.password.Text == sleutel)
            {
                open = true;
                30 password.Visible = false;
                this.Invalidate();
            }
        }

        35 void tekenScherm(object obj, PaintEventArgs pea)
        {
            if (open)
            {
                pea.Graphics.FillEllipse(Brushes.Yellow, 100, 100, 100, 100);
                40 pea.Graphics.FillEllipse(Brushes.Blue, 131, 135, 8, 8);
                pea.Graphics.FillEllipse(Brushes.Blue, 161, 135, 8, 8);
                pea.Graphics.DrawArc(new Pen(Color.Blue,3), 125, 125, 50, 50, 45, 90);
            }
            else
            45 pea.Graphics.DrawString( "please enter password", new Font("Arial", 14)
                                , Brushes.Black, 50, 50);
        }
    }
}
```

Listing 17: Geheim/Geheim.cs

dan geven we de variabele `open` de waarde `true`, om aan te geven dat het slot nu is opengemaakt. In de eventhandler `tekenScherm`, die door een aanroep van `Invalidate` aan het werk wordt gezet, controleren we de waarde van de variabele `open`. Alleen als die `true` is, wordt de tekening gemaakt. Omdat `open` een bool-variabele is, kan die direct dienen als voorwaarde in een if-opdracht: die voorwaarde moet immers een bool-expressie zijn. Als de bool-variabele de waarde `true` heeft, wordt de opdracht in de body uitgevoerd, als hij de waarde `false` heeft, wordt de opdracht achter `else` uitgevoerd.

Als finishing touch maken we van het tekstveld een echt password-veld. Door het veranderen van de property `PasswordChar` kunnen we er voor zorgen dat er tijdens het intikken alleen sterretjes in beeld verschijnen, zoals gebruikelijk bij password-velden. Is de sleutel eenmaal geraden, dan halen we het tekstveld helemaal weg door de property `Visible` de waarde `false` te geven. Dit is een property met het type `bool`.

Minimum/maximum thermometer

In listing 18 staat de tekst van een programma dat het gedrag vertoont van een maximum/minimum-thermometer; figuur 20 toont het programma in werking. Met de (witte) schuifregelaar in het midden (een `TrackBar`-control) kan de gebruiker de temperatuur instellen. De minimale en maximale waarde ooit behaald worden getoond in de blauwe en rode trackbar links en rechts van de ingestelde temperatuur. Als de gebruiker op de “reset”-toets drukt, worden maximum en minimum weer gelijk aan de huidige temperatuur, en kunnen we op nieuwe records gaan jagen. De userinterface is gemaakt met hulp van de visuele designer.

blz. 90

Kern van dit programma is de if-opdracht in de event-handler die gekoppeld is aan de temperatuur-trackbar: `temperatuur.Scroll`. Is de huidige waarde groter dan het maximum-tot-nu-toe? Dan moet het maximum worden aangepast!

7.3 Exceptions

Het afhandelen van fouten

Bij het uitvoeren van methodes kunnen er uitzonderlijke omstandigheden zijn die het onmogelijk maken dat de methode compleet wordt uitgevoerd. In dat geval is er sprake van een *exception*. Zo'n exception wordt door de methode opgeworpen, en het is dan aan de aanroeper om daar een oplossing voor te vinden.

Een voorbeeld van een methode die een exception opwerpt, is de statische methode `Parse` die beschikbaar is in types als `int` en `double`. Deze werpt een exception op als de aangeboden string iets anders dan cijfers (en eventueel een minteken, of in het geval van een `double` een decimale punt of letter E) bevat. In dat geval kan het programma geen normale verdere doorgang vinden.

De try-catch opdracht

Je zou kunnen vermijden dat er exceptions ontstaan, door vooraf te controleren of aan alle voorwaarden is voldaan (in het geval van `int.Parse`: of de aangeboden string uitsluitend cijfer-tekens bevat). Maar dan doe je dubbel werk, want `int.Parse` doet die controle nogmaals. Beter is het om te reageren op het optreden van de exception. Dat gebeurt met de speciale `try-catch`-opdracht. Je kunt de aanroep die mogelijkwerps een exception zal opwerpen in de body van een `try`-opdracht zetten. In het geval dat er inderdaad een exception optreedt, gaat het dan verder in de body van het `catch`-gedeelte. Gaat echter alles goed, dan wordt het `catch`-gedeelte overgeslagen. Bijvoorbeeld:

```
try
{
    n = int.Parse(s);
    uitvoer.Text = "kwadraat van " + n + " is " + n*n;
}
catch (Exception e)
{
    uitvoer.Text = s + " is geen getal";
}
```

In het `catch`-gedeelte wordt de opgeworpen exception als het ware “opgevangen”. Achter het woord `catch` moet een soort parameter worden gedeclareerd. Via deze parameter is te achterhalen wat er precies fout is gegaan. In het geval van `Parse` is dat zo ook wel duidelijk, maar de parameter moet toch gedeclareerd worden, ook als we hem niet gebruiken.

In de body van `try` kunnen meerdere opdrachten staan. Bij de eerste exception gaat het echter verder bij `catch`. De rest van de opdrachten achter `try` mag er dus van uitgaan dat er geen

```

using System;
using System.Windows.Forms;

namespace Thermo
5 {
    public partial class Thermo : Form
    {
        private TrackBar minimum;
        private TrackBar temperatuur;
10     private TrackBar maximum;
        private Button reset;

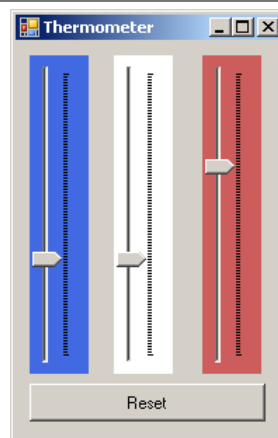
        public Thermo()
        {
15             InitializeComponent();
            this.temperatuur.Scroll += temperatuur_Scroll;
            this.reset.Click += reset_Click;
        }

20     private void temperatuur_Scroll(object sender, EventArgs e)
        {
            int x = this.temperatuur.Value;
            if (x < this.minimum.Value)
                this.minimum.Value = x;
25             if (x > this.maximum.Value)
                this.maximum.Value = x;
        }

        private void reset_Click(object sender, EventArgs e)
30         {
            this.minimum.Value = this.temperatuur.Value;
            this.maximum.Value = this.temperatuur.Value;
        }
    }
35 }

```

Listing 18: Thermo/Thermo.cs



Figuur 20: Het programma Thermo in werking

exception is opgetreden.

Let op dat de bodies van het `try`- en het `catch`-gedeelte tussen accolades moeten staan, zelfs als er maar één opdracht in staat. (Dat is wel onlogisch, want bij opdrachten zoals `if` en `while` mogen in die situatie de accolades worden weggelaten.)

Het type `Exception`, waarvan achter `catch` een variabele wordt gedeclareerd, is een klasse met allerlei subklassen: `FormatException`, `OverflowException`, `DivideByZeroException` enzovoorts. Deze verschillen in het soort details dat je over de exception kunt opvragen (door het opvragen van properties van het exception-object). Ben je niet geïnteresseerd in de details, dan kun je ruwweg een `Exception`-object declareren, maar anders kun je het object van het juiste type declareren.

Het is toegestaan om bij één `try`-opdracht meerdere `catch`-gedeeltes te plaatsen. Die kun je dan parameters van verschillend (sub-)type geven. Bij het optreden van een exception wordt de eerste afhandeling gekozen met een passend type. Bijvoorbeeld:

```
try
{
    n = int.Parse(s);
    uitvoer.Text = "kwadraat van" + n + " is" + n*n;
}
catch (FormatException e)
{
    uitvoer.Text = s + " is geen getal";
}
catch (OverflowException e)
{
    uitvoer.Text = s + " is te groot";
}
```

7.4 Toepassing: grafiek en nulpunten van een parabool

Beschrijving van de casus

We gaan een wat groter programma maken, waarin zowel keuze als herhaling als exceptions een belangrijke rol spelen. Maar bovendien controls met event-handlers, methoden met parameters en resultaat, locale variabelen en membervariabelen. Ook komen enkele typische aspecten van het werken met double-waarden aan de orde.

Het programma tekent een *parabool*. Een parabool is de grafiek van een wiskundige functie met het voorschrift $y = a \cdot x^2 + b \cdot x + c$, waarbij a , b en c nog nader te bepalen constanten zijn. In de GUI kan de gebruiker de waarden van a , b en c invoeren in tekstboxes, om dan meteen de grafiek te zien veranderen.

Een klassiek onderwerp in de middelbare-schoolwiskunde is het bepalen van de nulpunten van een parabool. Die kunnen gemakkelijk worden bepaald met de zogenaamde *abc*-formule (genoemd naar de constanten a , b en c die er in gebruikt worden). Het programma gaat deze formule gebruiken om naast de grafiek ook de waarden van de nulpunten aan de gebruiker te tonen.

Structuur van het programma

Het programma is gemaakt met hulp van de visuele designer. Er zijn drie tekstboxes `aBox`, `bBox` en `cBox`, waarin de gebruiker de waarden van a , b en c in kan vullen. Ze zijn alledrie gekoppeld aan dezelfde event-handler `box.TextChanged`, die zodra er iets wijzigt in een van de boxes de variabelen `a`, `b` en `c` een waarde geeft.

In de visuele designer hebben we de drie tekstboxes alvast een `Text` gegeven, zodat de gebruiker al een parabool kan aanschouwen zonder eerst waarden te hoeven invullen. Vanuit de constructor roepen we daartoe zelf ook de event-handler aan.

In de event-handler halen we de teksten uit de tekstboxes op, en gebruiken `Parse` om ze te converteren naar `double`-waardes. Dit gebeurt in een `try-catch`-opdracht. Mocht er een exception optreden, dan reageren we daarop door de box waarin als laatste iets is veranderd een rode achtergrondkleur te geven.

Gaat alles goed met het parsen, dan kunnen we het tekenen van de grafiek uitlokken door van het betreffende `Panel` de methode `Invalidate` aan te roepen. Op een daarvoor bestemd `Label` wordt een tekst met de door de *abc*-formule berekende nulpunten neergezet.

De nulpunten

In de methode `nulpunten` kunnen we ons nu helemaal op het uitrekenen van de nulpunten concentreren, zonder gedoe met tekstvelden. We kunnen er op vertrouwen dat de variabelen `a`, `b` en `c` de door de gebruiker ingevoerde waarden bevatten.

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
5 using System.Globalization;

namespace Parabool
{
    public partial class Parabool : Form
10 {
        public Parabool()
        {
            InitializeComponent();
            box_TextChanged(aBox, null);
15 }

        private double a, b, c;

        private double functie(double x)
20 {
            return a * x * x + b * x + c;
        }

        private void grafiek_Paint(object sender, PaintEventArgs pea)
25 {
            int xMid = grafiek.Width / 2;
            int yMid = grafiek.Height / 2;
            double schaal = 0.03;
            Graphics gr = pea.Graphics;
30 gr.SmoothingMode = SmoothingMode.AntiAlias;

            // assen
            gr.DrawLine(Pens.Red, 0, yMid, grafiek.Width, yMid);
            gr.DrawLine(Pens.Red, xMid, 0, xMid, grafiek.Height);
35

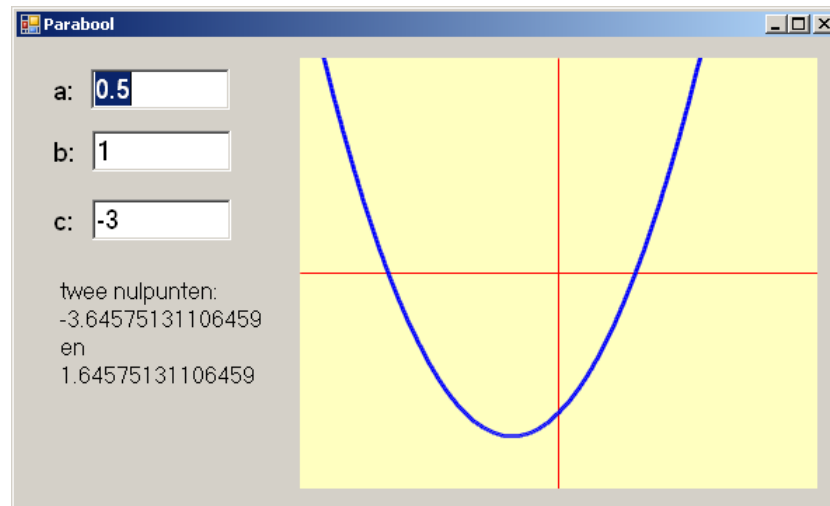
            // grafiek
            int xPixel, yPixel, yVorigePixel;
            double xWaarde, yWaarde;
            Pen pen = new Pen(Color.Blue, 3);
            yVorigePixel = 0;
40 for (xPixel = -1; xPixel < grafiek.Width; xPixel++)
            {
                xWaarde = (xPixel - xMid) * schaal;
                yWaarde = this.functie(xWaarde);
45 yPixel = (int)(yMid - (yWaarde / schaal));
                if (xPixel > 0)
                    gr.DrawLine(pen, xPixel-1, yVorigePixel, xPixel, yPixel);
                yVorigePixel = yPixel;
            }
        }
    }
}
```

Listing 19: Parabool/Parabool.cs, deel 1 van 2

```
private string oplossingen()
{
    double discr, noemer, wortel;
55     discr = b * b - 4 * a * c;
    noemer = 2 * a;
    if (noemer == 0)
        return "rechte lijn!";
    if (discr < 0)
60     return "geen nulpunten";
    if (discr == 0)
        return "een nulpunt: " + -b / noemer;
    wortel = Math.Sqrt(discr);
    return "twee nulpunten: "
65     + ((-b - wortel) / noemer).ToString(CultureInfo.InvariantCulture)
    + " en "
    + (-b + wortel) / noemer;
}

70 private void box_TextChanged(object box, EventArgs ea)
{
    try
    {
        a = double.Parse(aBox.Text, CultureInfo.InvariantCulture);
75     b = double.Parse(bBox.Text);
        c = double.Parse(cBox.Text);
        ((TextBox)box).BackColor = Color.White;
        grafiek.Invalidate();
        nulpunten.Text = this.oplossingen();
80    }
    catch (Exception e)
    {
        ((TextBox)box).BackColor = Color.Red;
        nulpunten.Text = e.Message;
85    }
}
}
```

Listing 20: Parabool/Parabool.cs, deel 2 van 2



Figuur 21: Het programma Parabool in werking

De nulpunten kunnen we uitrekenen met behulp van de *abc*-formule, maar dan moeten er natuurlijk wel nulpunten zijn. In de *abc*-formule wordt de wortel getrokken uit $b^2 - 4ac$, dus als die waarde negatief is, zijn er geen oplossingen. Bovendien wordt er gedeeld door $2a$, dus als die waarde 0 is moet er ook speciale actie ondernomen worden. Met if-opdrachten worden al deze gevallen uitgesplitst, waarna in elk van de gevallen een toepasselijke melding op het scherm kan worden gezet.

De grafiek

Basisidee van het tekenen van de grafiek is dat we voor alle mogelijke x -waarden de bijbehorende y -waarde berekenen. Op dat punt zouden we een stipje kunnen zetten:

```
for (xPixel=0; xPixel<500; xPixel++)
{
    yPixel = this.functie(xPixel);
    gr.FillRectangle(Brushes.Blue, xPixel, yPixel, 1, 1);
}
```

Maar als de grafiek erg steil loopt, en y -waarden voor aangrenzende x -waarden meer verschillen dan de dikte van de stippen, dan komen de stippen los te staan. Beter is het daarom, om in plaats van een stip een lijn te trekken vanaf het vorige berekende punt. De y -waarde van het punt wordt daarom na gebruik bewaard in de variabele `yVorigePixel` voor de volgende ronde. Met een if-opdracht zorgen we ervoor om de eerste keer geen lijn te trekken (want dan was er geen vorige ronde, en heeft `yVorigePixel` dus nog geen zinvolle waarde). De oude x -waarde hoeven we niet apart te bewaren, want die is natuurlijk $x - 1$.

```
for (xPixel=-1; xPixel<500; xPixel++)
{
    yPixel = this.functie(xPixel);
    if (xPixel>=0)
        gr.DrawLine(Pens.Blue, xPixel-1, yVorigePixel, xPixel, yPixel);
    yVorigePixel = yPixel;
}
```

De compiler is nogal streng, en blijft zeuren dat we de variabele `yVorigePixel` gebruiken zonder hem vantevoren een waarde te geven. Dat is omdat de compiler wel ziet dat in de while-body `yVorigePixel` gebruikt lijkt te worden voordat hij een waarde krijgt, maar niet slim genoeg is om te voorzien dat die if-opdracht dat nou juist voorkomt. Om de compiler tevreden te stellen geven we `yVorigePixel` voorafgaand aan de while-opdracht zomaar een waarde.

Schaling

Het interval $[0, 500]$ is niet het interessantste gebied om een parabool te bekijken, althans niet voor de waarden van a , b en c die je als eerste te binnen schieten. Interessanter is bijvoorbeeld het

interval $[-7, 7]$. Daarom wordt de berekening van de functiewaarde niet gedaan met de waarde van `xPixel`, maar met de verschoven en opgeschaalde versie `xWaarde` daarvan. De verkregen `yWaarde` wordt teruggeschaald, teruggeschoven, en er wordt bovendien gecorrigeerd voor het C#-coördinatensysteem dat de verkeerde kant oploopt. 't Is een beetje lastig onder woorden te brengen; bekijk de formules in listing 19 en reken maar na.

blz. 92

In de listing is bovendien te zien dat we niet tot pixel 500 doorlopen, maar dit afhankelijk maken van de breedte van het panel. Zo hoeven we niet dat getal te veranderen als we het panel met de visuele designer een andere breedte geven.

Hoofdstuk 8

Objecten en klassen

8.1 Klasse: beschrijving van een object

Object: groepje variabelen met methoden

Een object is een groepje variabelen dat bij elkaar hoort. Je realiseert je dat niet voortdurend, maar iedere keer als je een object creëert met **new**, maak je een nieuw groepje variabelen. Bijvoorbeeld, als je een nieuwe button maakt met de expressie **new Button()** dan ontstaat er een groepje variabelen waarin alle noodzakelijke administratie voor een button wordt bijgehouden: de afmetingen, de positie op het scherm, de kleur, de status (ingedrukt of niet), de bijbehorende event-handler, het opschrift, enzovoorts.

Met die variabelen heb je echter niet direct te maken: om een button-object te kunnen gebruiken in je programma hoef je niet eens te weten welke variabelen er precies in een button-object zitten, en hoe die heten. Wel is het van belang om te weten welke properties het object heeft, en met welke methoden je het object kunt manipuleren. Voor een button-object zijn er bijvoorbeeld de properties **Text** (om het opschrift te veranderen) en de event-property **Click**. Als je deze properties aanpast, verander je indirect de variabelen die deel uitmaken van het object.

Klasse: declaratie van variabelen plus definitie van methoden

De klasse-definitie is een beschrijving van de objecten van die klasse. In de klasse-definitie staan daarom:

- declaraties van de member-variabelen waaruit het object bestaat
- een definitie van de methoden waarmee het object gemanipuleerd kan worden
- een definitie van de properties van het object die opgevraagd en/of gewijzigd kunnen worden

Behalve voor de klassen in de libraries geldt dit ook voor de klassen die je zelf schrijft. Als voorbeeld kijken we wat er in het geheugen gebeurt als het programma **CirkelGroe**i uit sectie 7.2 wordt uitgevoerd.

De definitie van klasse **CirkelGroe**i begint met:

```
class cirkelGroe : Form
{   private Button kleiner, groter;
    private int straal;
```

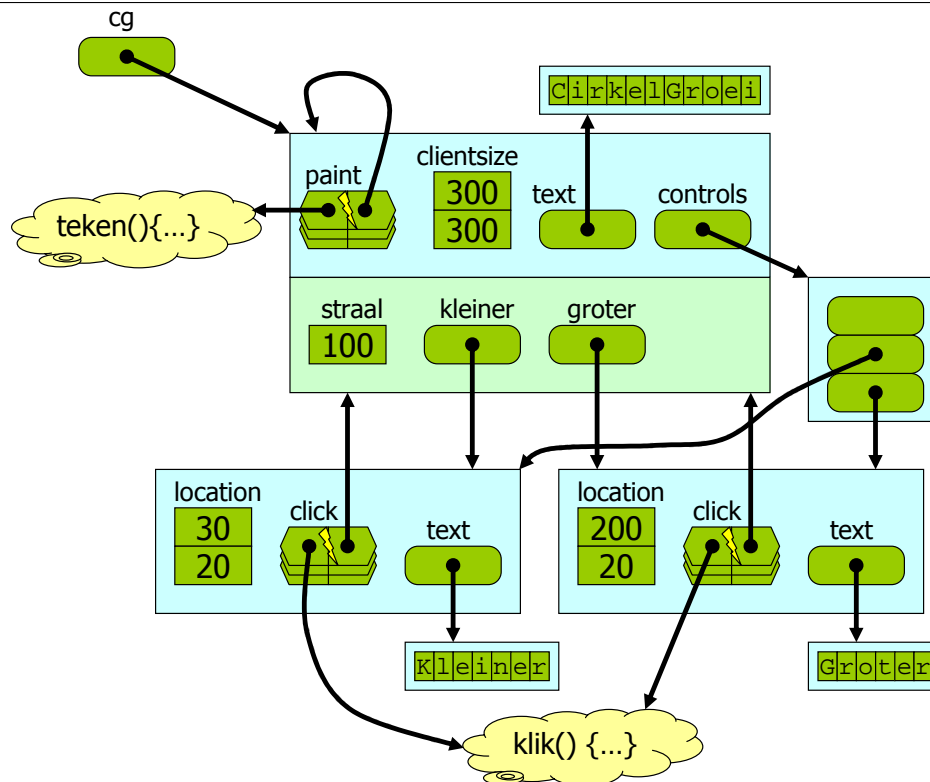
Het **CirkelGroe**i-object dat in de methode **Main** wordt gemaakt bestaat dus uit drie variabelen: twee verwijzingen naar button-objecten en een geheel getal. Bovendien bevat een **CirkelGroe**i-object, omdat de klasse een subklasse is van **Form**, alle variabelen die in de klasse **Form** al waren gedeclareerd. Welke variabelen dat precies zijn kunnen we niet weten, want dat zijn tenslotte **private** variabelen van de klasse **Form**. Maar om een idee te krijgen van wat er gebeurt kun je je voorstellen dat er voor elke property een overeenkomstige private variabele bestaat. Laten we zeggen een variabele **text** voor de property **Text**, een variabele **controls** voor de property **Controls**, en een variabele **paint** voor de event-property **Paint**.

Terwijl je het programma schrijft heb je misschien het gevoel dat van elk van de variabelen die bovenin de klasse zijn gedeclareerd, er maar één aanwezig is. Dat is ook wel zo binnen één object, maar je kunt meerdere exemplaren van zo'n object aanmaken. Elk van de aangemaakte objecten heeft zijn eigen setje variabelen. Iedere button heeft bijvoorbeeld een eigen **Location**.

Objecten en objectverwijzingen

Om een idee te krijgen van hoe objecten in het geheugen worden opgebouwd, bekijken we in detail wat er gebeurt bij het runnen van het programma **CirkelGroe**i. Bekijk figuur 22 voor een impressie

van de samenhang tussen de relevante objecten.



Figuur 22: Impressie van het geheugen bij uitvoer van het programma CirkelGroei

In de methode `Main` wordt een object-verwijzing `cg` van het type `CirkelGroei` gedeclareerd (deze variabele is linkboven in de figuur getekend). Door de aanroep van de constructormethode `new CirkelGroei()` wordt het object ook echt aangemaakt.

Het `CirkelGroei`-object bestaat uit de variabelen die in de klasse zijn gedeclareerd: een int-variabele `straal` en twee button-object-verwijzingen `kleiner` en `groter`. Maar omdat de klasse `CirkelGroei` een subklasse is van de klasse `Form`, bestaat het `CirkelGroei`-object voor een deel ook uit de variabelen die al in de klasse `Form` waren gedeclareerd. Dat zijn er erg veel; in de figuur tonen we er een aantal: `clientsize`, `text`, `controls` en `paint`.

Bij het aanmaken van het object wordt ook de bijbehorende constructormethode aangeroepen, die het gloednieuwe object met de naam `this` kan aanspreken. Met toekenningsoopdrachten krijgen de diverse variabelen een waarde:

```
this.straal = 100;
this.ClientSize = new Size(300, 300);
this.Text      = "CirkelGroei";
this.Paint    += this.tekenScherm;
this.kleiner  = new Button();
this.groter   = new Button();
```

Het zijn zowel zelf-gedeclareerde member-variabelen (zoals `straal`, `kleiner` en `groter`) als variabelen die zijn geërfd van de klasse `Form`: `ClientSize`, `Text` en `Paint`. Of eigenlijk zijn het de properties die we veranderen, maar voor deze bespreking nemen we maar even aan dat die één op één corresponderen met private membervariabelen.

Bekijk in figuur 22 hoe de verschillende variabelen worden opgeslagen: `straal` is een `int` en staat dus direct in het object, `clientsize` is een `Size`. Dat is een als `struct` gedefinieerd object, en de waarde staat dus direct in het object. Dit in tegenstelling tot `text`, `kleiner` en `controls`: dat zijn als `class` gedefinieerde objecten, en hiervoor wordt dus een verwijzing opgeslagen naar het eigenlijke object.

De variabele `controls` stond niet in bovenstaand rijtje toekenningsoopdrachten. Toch heeft die variabele een waarde gekregen. Dat is gebeurd in de constructormethode van de klasse `Form`: bij het uitvoeren van een constructormethode wordt eerst de constructor van de superklasse automatisch uitgevoerd. Dit geeft de programmeur van de superklasse de gelegenheid om alvast de nodige members een waarde te geven. Dat kan natuurlijk alleen de members van die superklasse betreffen: de programmeur van `Form` wist nog niet dat er in de subklasse een variabele `straal` zal bestaan. Twee andere opdrachten in de constructormethode zijn:

```
this.Controls.Add(this.kleiner);
this.Controls.Add(this.groter);
```

Hier wordt (via de corresponderende property) de private member-variabele `controls` onder handen genomen door de methode `Add`. Het type van deze variabele is `ControlCollection`; je kunt je dat voorstellen als een rijtje object-verwijzingen naar controls. Met de methode `Add` worden daar extra verwijzingen in neergezet, en op deze manier geven we de twee buttons als het ware bij de `Form` in beheer.

En dan zijn er nog de toekenningsoopdrachten aan event-properties:

```
this.Paint += this.teken;
this.kleiner.Click += this.klik;
this.groter.Click += this.klik;
```

Een event-property is ingewikkelder van opbouw dan je misschien zou verwachten. De figuur geeft een vrij nauwkeurige indruk hiervan. Ten eerste bevat de variabele een verwijzing naar een methode (of eigenlijk: de gecompileerde code ervan). Dat is op zich al bijzonder, want een methode is geen object. Maar daarnaast houdt de event-variabele ook bij welk object door die methode onder handen genomen zal gaan worden als het event inderdaad optreedt. En wat het nog ingewikkelder maakt: een event kan meerdere ‘abonnees’ hebben, dus de event-variabele bevat niet één code-verwijzing met bijbehorende object, maar een heel stapeltje daarvan.

Het object wat in de ‘rechterhelften’ van de event-variabelen wordt opgeslagen, is in dit geval steeds het `this` object op het moment dat de toekenning plaatsvindt. Omdat de toekenningen in de constructormethode van het `CirkelGroe`i object staan, is dat hier dus steeds het `CirkelGroe`i object. Dit maakt dat de event-variabele van `Paint` naar zijn eigen object wijst, maar die van de twee buttons naar een ander object.

Pas als de hele constructormethode is voltooid, wordt uiteindelijk de toekenningsoopdracht van

```
cg = new CirkelGroe();
```

uitgevoerd, die maakt dat de objectverwijzing `cg` naar het nieuwe object gaat wijzen. Daarmee is de situatie in figuur 22 helemaal compleet.

8.2 Toepassing: Bewegende deeltjes

Programma’s met meerdere klassen

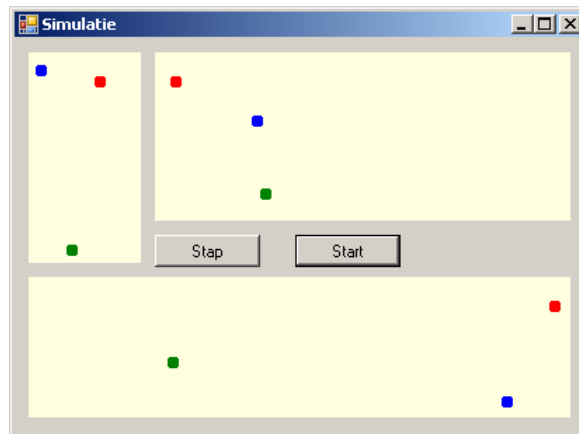
Tot nu toe hebben we steeds programma’s geschreven die grotendeels uit één klasse bestonden. In sommige programma’s hebben we een tweede klasse gedeclareerd, maar die diende dan alleen om de methode `Main` een plaatsje in een eigen klasse te geven, op de manier zoals dat ook in het standaardraamwerk gebeurt dat Visual Studio aanmaakt.

Steeds was de eerste klasse een subklasse van `Form`, en maakten we in methode `Main` een nieuw object van die klasse, om daarmee vervolgens `Run` aan te roepen. In de programma’s speelden weliswaar naast dat object ook nog andere objecten een rol, maar het type van die objecten was steeds een bibliotheek-klasse (`Button`, `TrackBar`, `Graphics`, enzovoorts).

In de nu volgende toepassing verandert dat. We gaan behalve de welbekende subklasse van `Form` nog meer klassen maken, en in het programma gaan we objecten gebruiken die die klassen als type hebben. Op deze manier kunnen we “zelfgemaakte” objecten naar eigen ontwerp in het programma gebruiken.

Beschrijving van de casus

Het programma dat we in deze sectie zullen ontwikkelen is een simulatie van bewegende deeltjes in een begrensde ruimte. Je kunt denken aan moleculen in een afgesloten vat, of aan biljartballen op een (wrijvingsloze) tafel, of aan ratten in een val. Voor het gemak zullen we de deeltjes afbeelden als kleine gekleurde cirkels, en de ruimte als grote rechthoek.



Figuur 23: Het programma Simulatie in werking

In het runnende programma zullen drie ruimtes zichtbaar zijn, ieder met verschillende afmetingen. In iedere ruimte bevinden zich drie deeltjes, ieder met een verschillende kleur. De deeltjes kunnen bewegen. Ze doen één “stap” als de gebruiker op de button met het opschrift “stap” drukt. Er is bovendien een button met het opschrift “start”. Als de gebruiker daarop drukt, blijven de deeltjes bewegen, en ziet de gebruiker dus een animatie. Het opschrift van deze button verandert op dat moment ook in “stop”, en met nog een druk op deze knop kan de gebruiker de animatie weer stopzetten. In figuur 23 is een snapshot van het programma te zien.

De klasse Ruimte

In het window van de simulatie (zie figuur 23) zijn vijf dingen te zien: drie ruimtes met deeltjes, en twee buttons. Het zou het gemakkelijkste zijn als we deze vijf dingen op dezelfde manier konden maken: creëren van een nieuw object met `new`, en deze aan de control-collection toevoegen met `this.Controls.Add`.

Voor de buttons is dat geen probleem, maar er bestaat natuurlijk geen bibliotheek-klasse `Ruimte`. Toch is dat geen beletsel, want we kunnen zo’n klasse zelf maken. We hoeven die klasse gelukkig niet van de grond af aan op te bouwen. We kunnen namelijk voortborduren op een al wel bestaande klasse: `UserControl`, die speciaal bedoeld is om als superklasse te dienen voor zelfbedachte controls.

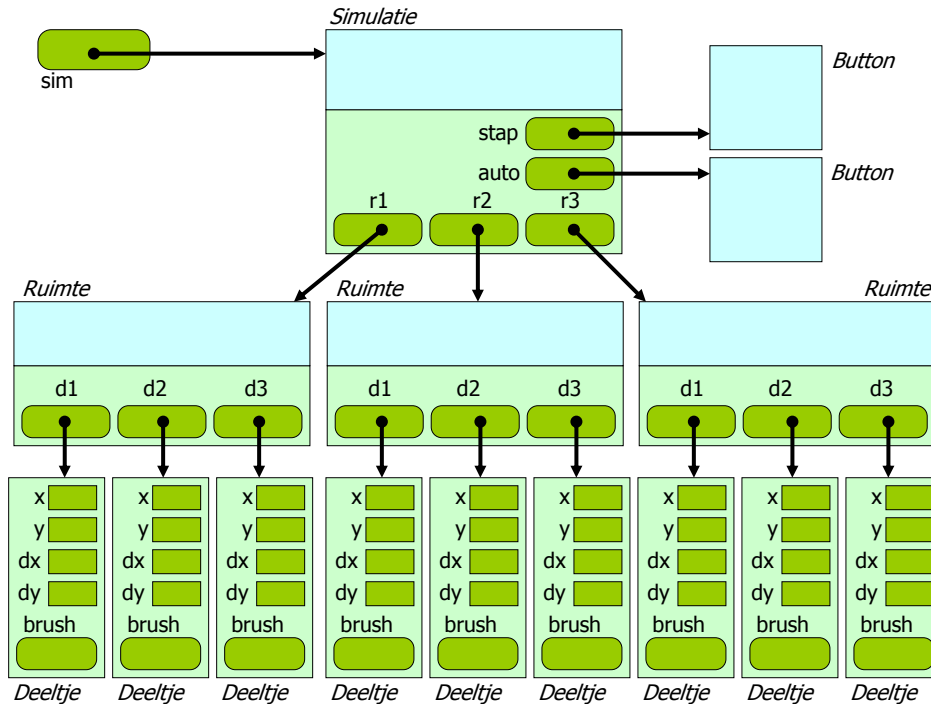
Een `UserControl` is een control, net zoals `Button` en `TextBox`. Het kan daarom met `Controls.Add` aan de door een `Form` beheerde controls worden toegevoegd. Een `UserControl` heeft afmetingen, een achtergrondkleur, dus daar hoeven we ons niet meer druk om te maken. In de klasse `Ruimte`, die een subklasse wordt van `UserControl`, hoeven we alleen maar extra member-variabelen toe te voegen die de gekleurde deeltjes beschrijven, en methoden die die deeltjes tekenen en laten bewegen.

De klasse Deeltje

Elk bewegend deeltje dat in dit programma een rol speelt, heeft een aantal eigenschappen: een brush die zijn kleur bepaalt, een positie binnen de ruimte, en een bewegingsrichting. Voor elk van deze eigenschappen kunnen we variabelen declareren. De brush is een `Brush`-verwijzing, de positie bestaat uit twee int-waarden `x` en `y`, en de bewegingsrichting kunnen we beschrijven met de afstand `dx` en `dy` die het deeltje bij elke stap moet bewegen.

Elk deeltje moet zijn eigen brush, positie en richting krijgen. We maken daarom een aparte klasse `Deeltje`, waarin deze variabelen zijn gedeclareerd. In het programma kunnen we dan voor elk deeltje een `Deeltje`-object creëren, waarin de benodigde variabelen gebundeld zijn.

De klasse `Deeltje` is geen subklasse van een al bestaande klasse; we maken hem puur vanuit het niets, en een `Deeltje`-object bestaat dan ook alleen uit de variabelen die we zelf in de klasse declareren.



Figuur 24: Impressie van het geheugen bij uitvoer van het programma Simulatie

Opzet van de klassen

Het programma zal dus gaan bestaan uit drie klassen: de subklasse van `Form` die we ditmaal `Simulatie` zullen noemen, en de extra klassen `Ruimte` en `Deeltje`. We bekijken nu eerst de opzet van de klassen voor wat betreft de declaraties van membervariabelen; de methoden volgen later.

```
class Simulatie : Form
{
    Button stap, auto;
    Ruimte r1, r2, r3;
    // methoden nog toe te voegen
}
class Ruimte : UserControl
{
    Deeltje d1, d2, d3;
    // methoden nog toe te voegen
}
class Deeltje
{
    Brush brush;
    int x, y, dx, dy;
    // methoden nog toe te voegen
}
```

Er moeten nu objecten gemaakt worden met deze klassen als type, zodat er een heel netwerk van verwijzingen ontstaat. De gewenste situatie is aangegeven in figuur 24.

In methode `Main` maken we een `Simulatie`-object. Dit bestaat voor een deel uit de geërfde variabelen van `Form` (in de figuur niet verder uitgewerkt), en voor een deel uit de vijf in de klasse `Simulatie` extra gedeclareerde variabelen: `stap`, `auto`, `r1`, `r2` en `r3`. Het is de taak van de constructormethode om deze variabelen naar nieuw te maken objecten te laten wijzen, en daarvan de nodige attributen aan te passen. Dat gebeurt zoals gewoonlijk door middel van toekenningsoverdrachten:

```
r1 = new Ruimte(); r1.Location = new Point( 10, 10); r1.Size = new Size( 80, 150);
r2 = new Ruimte(); r2.Location = new Point(100, 10); r2.Size = new Size(296, 120);
r3 = new Ruimte(); r3.Location = new Point( 10, 170); r3.Size = new Size(386, 100);
stap = new Button(); stap.Location = new Point(100, 140); stap.Text = "Stap";
```

```
auto = new Button(); auto.Location = new Point(200, 140); auto.Text = "Start";
```

Hierdoor gaan `stap` en `auto` naar nieuw gecreëerde `Button`-objecten wijzen, en `r1`, `r2` en `r3` naar nieuw gecreëerde `Ruimte`-objecten. De `Ruimte`-objecten bestaan voor een deel uit de geërfde variabelen van `UserControl`, en voor een deel uit de drie in de klasse `Ruimte` extra gedeclareerde variabelen: `d1`, `d2` en `d3`. Deze wijzen echter nog niet naar objecten.

De constructormethode van `Ruimte`

Bij het evalueren van een `new`-expressie gebeuren er twee dingen:

- er wordt ruimte aangemaakt voor het nieuwe object;
- de constructormethode, zoals gedefinieerd in de bijbehorende klasse, wordt aangeroepen.

Alle opdrachten die in de constructormethode staan, worden dus automatisch uitgevoerd op het moment dat het object gecreëerd wordt. Dat maakt de constructormethode de ideale plaats om de membervariabelen een beginwaarde te geven, en eventuele andere voorbereidingen te treffen.

De constructormethode onderscheidt zich op twee manieren van de andere methoden die in een klasse worden gedefinieerd:

- de naam van de constructormethode is hetzelfde als de naam van de klasse;
- de constructormethode heeft geen resultaattype, *zelfs niet void*; de constructormethode kan immers alleen maar via de speciale `new`-expressie worden aangeroepen, en die heeft automatisch het nieuw gecreëerde object als resultaatwaarde.

In onze nieuwe klasse `Ruimte` gaan we ook een constructormethode schrijven, waarin het `Ruimte`-object, direct nadat het is gecreëerd, wordt klaargezet voor gebruik.

Eerste taak is het instellen van de achtergrondkleur. Dit kan via een van `UserControl` geërfde property. We maken de achtergrond van elk `Ruimte`-object hetzelfde:

```
this.BackColor = Color.LightYellow;
```

Nu wordt het tijd om de membervariabelen van het `Ruimte`-object een waarde te geven: de verwijzingen naar `Deeltje`-objecten `d1`, `d2` en `d3`. Deze verwijzingen wijzen uit zichzelf nog nergens naar, dus de `Deeltje`-objecten moeten nog worden gecreëerd:

```
d1 = new Deeltje(iets );
d2 = new Deeltje(iets );
d3 = new Deeltje(iets );
```

Direct bij de creatie van een `Ruimte`-object, worden dus ook de drie bijbehorende `Deeltje`-objecten gemaakt. De klasse `Deeltje` heeft ook een constructormethode, waarin de variabelen van het nieuwe `Deeltje`-object een waarde krijgen. We verplaatsen onze aandacht daarom nu eerst even naar de methoden in de klasse `Deeltje`.

De methoden van `Deeltje`

De klasse `Deeltje` is nergens een subklasse van. Een `Deeltje`-object bestaat dus alleen maar uit de variabelen die in de klasse `Deeltje` zijn gedeclareerd: `x`, `y`, `dx`, `dy` en `brush`. De vraag die zich nu opdringt is wat we met zo'n `Deeltje`-object eigenlijk willen doen; met andere woorden: welke methoden zijn er nodig in de klasse `Deeltje`? Het is niet zo moeilijk om een paar nuttige handelingen met `Deeltje`-objecten te bedenken:

- een constructormethode, die de member-variabelen een beginwaarde geeft
- een methode `DoeStap`, die de plaats van het deeltje zodanig verandert, dat het deeltje een stap doet in de door de richting-variabelen aangegeven richting
- een methode `TekenDeeltje`, die het deeltje intekent op een als parameter te specificeren `Graphics`-object.

De methode `TekenDeeltje` krijgt een `Graphics`-object als parameter, zodat hij grafische methoden kan aanroepen. We gaan deze methode later aanroepen op het moment dat een deeltje getekend moet worden, maar dat is van later zorg. Voor het eigenlijke tekenen van het deeltje zijn alleen zijn eigen `brush`-kleur en positie van belang; die worden in de body van `TekenDeeltje` dan ook gebruikt.

```
public void TekenDeeltje(Graphics gr)
{
    gr.FillEllipse(this.brush, this.x-4, this.y-4, 9, 9);
}
```

Die variabelen moeten dan natuurlijk wel een waarde hebben. De ideale plaats om dat te doen is de constructormethode. Door de gewenste brush en startpositie als parameter van de constructor-methode mee te geven, kan elk deeltje zijn eigen kleur en positie krijgen. Ook de snelheid van het deeltje wordt bij de constructie vastgelegd:

```
public Deeltje(Brush brush, int x, int y, int dx, int dy)
{
    this.brush = brush;
    this.x     = x;
    this.y     = y;
    this.dx    = dx;
    this.dy    = dy;
}
```

Let op dat er in deze situatie zowel een member-variabele als een parameter is die **brush** heet. Dat is geen probleem: de member-variabele kan worden aangesproken met **this.brush**, de parameter met **brush** zonder meer.

De methode **DoeStap** is het interessantste. Door aanroep van deze methode gaat het deeltje bewegen. Omdat zowel de huidige plaats als de bewegingsrichting in het Deeltje-object zijn opgeslagen, hoeft deze methode geen parameters te krijgen. De beweging vindt in principe plaats door de variabelen **x** en **y** te vermeerderen met de “bewegings-waarden” **dx** en **dy**:

```
public void DoeStap()
{
    this.x += this.dx;
    this.y += this.dy;
}
```

Het kan echter gebeuren (bij negatieve bewegingsrichting) dat de coördinaten negatief worden. In dat geval is het gewenst dat het deeltje “terugkaats” tegen de muur. De coördinaat wordt dan juist zo positief als hij negatief geworden was. Maar door het kaatsen verandert ook de bewegingsrichting: bij het kaatsen tegen de zij-muren klappt het teken van de bewegingsrichting om: beweging naar links wordt beweging naar rechts, en omgekeerd.

Het kaatsen tegen de linker- en bovenmuur wordt afgehandeld door de volgende opdrachten:

```
if (this.x < 0)
{
    this.x = -this.x;
    this.dx = -this.dx;
}
if (this.y < 0)
{
    this.y = -this.y;
    this.dy = -this.dy;
}
```

Het deeltje kan echter ook bij de rechter- en ondermuur dreigen uit beeld te raken, en ook hier zouden we het deeltje willen laten terugkaatsen. Om dat te kunnen testen, moet het deeltje echter “weten” hoe groot de ruimte is waarin hij beweegt, maar aan de membervariabelen **x**, **y**, **dx**, **dy** en **brush** is dat niet te zien!

Voor dat doel gaan we aan **DoeStap** toch een parameter meegeven die de afmetingen van de ruimte waarin het deeltje beweegt aangeeft. Een **Size**-object is precies wat we nodig hebben. De **Width**-property van dat object kunnen we nu gebruiken in een derde test in de methode **DoeStap**:

```
public void DoeStap(Size hok)
{
    if (this.x >= hok.Width)
```

Het is wel leuk om zelf even na te denken hoe deze situatie moet worden opgevangen (de oplossing staat in listing 23).

blz. 109

De methoden van Simulatie

De klasse **Simulatie** wordt alleen maar gebruikt voor de creatie van één object, maar is evengoed belangrijk. Het object modelleert het totale form, en heeft dezelfde opbouw als alle eerdere programma's: een constructormethode om de controls te creëren, en event-handlers om de click-events van de buttons af te handelen. We gaan *niet* het paint-event gebruiken, omdat er niets op de achtergrond van de form wordt getekend. De twee buttons en de drie UserControl-objecten tekenen

zichzelf namelijk automatisch.

Een deel van de constructormethode hebben we hierboven al besproken: creatie van de vijf objecten waar de membervariabelen naar toe moeten wijzen.

```
public Simulatie()
{
    this.ClientSize = new Size(406, 280); this.Text = "Simulatie";
    r1 = new Ruimte(); r1.Location = new Point( 10, 10); r1.Size = new Size( 80, 150);
    r2 = new Ruimte(); r2.Location = new Point(100, 10); r2.Size = new Size(296, 120);
    r3 = new Ruimte(); r3.Location = new Point( 10, 170); r3.Size = new Size(386, 100);
    stap = new Button(); stap.Location = new Point(100, 140); stap.Text = "Stap";
    auto = new Button(); auto.Location = new Point(200, 140); auto.Text = "Start";
}
```

Zoals gewoonlijk moeten de controls ook aan de verzameling `Controls` worden toegevoegd

```
this.Controls.Add(r1); this.Controls.Add(r2); this.Controls.Add(r3);
this.Controls.Add(stap); this.Controls.Add(auto);
```

De twee buttons krijgen ieder een eigen event-handler

```
this.stap.Click += stap_Click;
this.auto.Click += auto_Click;
}
```

De event-handler `stap_Click` zou de methode `DoeStap` van alle `Deeltje`-objecten kunnen worden aangeroepen. We gaan dat echter niet voor alle negen `Deeltje`-objecten apart doen. In plaats daarvan laten we het werk doen door de drie `Ruimte`-objecten, door daarvan een methode aan te roepen, die we (ook!) `DoeStap` noemen. Niet vergeten dat we die straks nog moeten schrijven...

```
private void stap_Click(object o, EventArgs ea)
{
    r1.DoeStap();
    r2.DoeStap();
    r3.DoeStap();
}
```

De methoden van Ruimte

Het enige wat ons nu nog te doen staat is het schrijven van de methoden van de klasse `Ruimte`. Met de constructormethode van `Ruimte` hadden we al een begin gemaakt. Hier worden ook de drie `Deeltje`-objecten van het `Ruimte`-object gecreëerd. Inmiddels weten we nu ook wat daarbij als parameter meegegeven moet worden: een `Brush`-object, en vier `int`'s voor de startpositie en -snelheid.

```
public Ruimte()
{
    this.BackColor = Color.LightYellow;
    d1 = new Deeltje(Brushes.Red, 30, 40, 10, 10);
    d2 = new Deeltje(Brushes.Green, 100, 80, 5, -10);
    d3 = new Deeltje(Brushes.Blue, 200, 60, 8, 2);
    this.Paint += this.tekenRuimte;
}
```

We moesten niet vergeten om in de klasse `Ruimte` ook een methode `DoeStap` klaar te zetten. die hadden we immers aangeroepen vanuit de methode `DoeStap` van de klasse `Simulatie`. Het schrijven van deze methode is heel simpel: we zetten gewoon “onze” drie deeltjes aan het werk om een stap te doen. De deeltjes wilden de afmeting van de ruimte waarin ze zich bevinden als parameter meekrijgen. Die kunnen ze krijgen: de afmetingen van een `UserControl` (en dus ook van een `Ruimte`) zijn eenvoudig op te vragen als de property `Size`. Nadat de deeltjes zijn aangepast, forceren we het opnieuw tekenen van de `Ruimte`-control door een aanroep van `Invalidate`.

```
public void DoeStap()
{
    d1.DoeStap(this.Size);
    d2.DoeStap(this.Size);
    d3.DoeStap(this.Size);
}
```

```

        this.Invalidate();
    }

```

De laatste methode van de klasse `Ruimte` is de paint-event-handler `tekenRuimte`. Voor het tekenen van het `Ruimte`-object laten we “onze” drie deeltjes zichzelf tekenen, door aanroep van de methode `TekenDeeltje` die we daarvoor hadden klaargezet in de klasse `Deeltje`.

```

private void tekenRuimte(object o, PaintEventArgs pea)
{
    Graphics gr = pea.Graphics;
    d1.TekenDeeltje(gr);
    d2.TekenDeeltje(gr);
    d3.TekenDeeltje(gr);
}

```

Hiermee is het programma bijna voltooid. Wat rest is de afhandeling van het indrukken van de tweede button. Maar dat is een onderwerp apart.

8.3 Animatie

Automatische actie

Door steeds maar op de “Stap”-button te blijven drukken, kunnen we de deeltjes laten blijven bewegen. Maar dat wordt op den duur vervelend; leuker zou het zijn als we lui achterover kunnen leunen, en de deeltjes automatisch blijven bewegen. Met andere woorden: als het programma zich als een tekenfilm, oftewel een *animatie* zou kunnen gedragen.

De klasse `Thread`

Kern van het animatie-mechanisme is de klasse `Thread` in library `System.Threading`. Als je een animatie wilt maken, moet je een object van deze klasse creëren; een `Thread`-object dus. Als parameter moet je daarbij een zelfgeschreven *methode* meegeven. Dus niet het resultaat van de aanroep van een methode, maar de methode zelf! Laten we zeggen dat we een methode `run` hebben, dan maak je een `Thread`-object als volgt:

```

Thread animatie;
animatie = new Thread(this.run);

```

Nadat het `Thread`-object gecreëerd is, kun je de animatie starten door de methode `Start` aan te roepen:

```

animatie.Start();

```

Het `Thread`-object reageert daarop door op zijn beurt de methode `run` aan te roepen die bij creatie als parameter was meegegeven. Dit lijkt allemaal maar omslachtig. Waarom al die moeilijkdoenerij om de methode `run` aan te roepen? Dat kan toch ook direct door

```

this.run();

```

te schrijven? Dat kan inderdaad, maar er is één verschil: het `Thread`-object roept de methode `run` aan, maar *wacht niet totdat dat afgelopen is*. De methode `start` zet de methode `run` aan het werk, en keert direct daarna terug naar de aanroeper. Vanaf dat moment gebeuren er dus *twee dingen tegelijk*: de methode `run` begint, maar de rest van het programma draait ook weer verder!

Daar gaan we gebruik van maken, door in de methode `run` een opdracht steeds opnieuw te laten uitvoeren. De opdracht staat daarom in de body van een `while`-opdracht, met als voorwaarde iets wat altijd waar blijft:

```

private void run()
{
    while (1==1)
        this.stap_Click(this, null);
}

```

(In plaats van de altijd-ware voorwaarde `1==1` hadden we ook de bool constante `true` kunnen gebruiken). Hiermee wordt de event-handler van de “Stap”-button automatisch steeds opnieuw aangeroepen.

De methode `sleep`

Maar wacht even, dat gaat wel erg snel: nu is het stuiteren van de deeltjes bijna niet meer te volgen. Na elke stap moet eigenlijk een korte pauze worden ingelast. Dat kan, en wel door aanroep

van de statische methode `Sleep` uit de klasse `Thread`. Als parameter krijgt deze methode het aantal milliseconden dat de pauze moet duren. We herzien dus de methode `run`:

```
private void run()
{
    while (true)
    {
        this.stap_Click(this, null);
        Thread.Sleep(50);
    }
}
```

De pauze van 50 milliseconden zorgt ervoor dat er 20 keer per seconde een stap wordt uitgevoerd, wat een mooie vloeiende animatie oplevert.

Controleren van de animatie

Het Thread-mechanisme kunnen we gebruiken om het bewegen van de deeltjes automatisch te laten verlopen na het indrukken van de start-button. Het Thread-object wordt gecreëerd en gestart als reactie op het indrukken van de button:

```
private void auto_Click(object o, EventArgs ea)
{
    animatie = new Thread(this.run);
    animatie.Start();
}
```

Om de animatie ook weer te kunnen stoppen, veranderen we op dat moment het opschrift van de button:

```
auto.Text = "Stop";
```

De afhandeling van de auto-button moet, nu hij het opschrift “Stop” heeft, natuurlijk anders verlopen dan voorheen. Dat betekent dat we de body van `auto_Click` weer even moeten herzien. We gebruiken een bool variabele `beweging`, en we zorgen ervoor dat die de waarde `true` heeft zolang de animatie loopt. Deze variabele wordt als membervariabele gedeclareerd, en krijgt in bij zijn declaratie de waarde `false`.

De afhandeling van het indrukken van de auto-button verloopt nu zo:

```
private void auto_Click(object o, EventArgs ea)
{
    if (beweging)
    {
        // stop de animatie
        beweging = false;
        auto.Text = "Start";
    }
    else
    {
        // start de animatie
        beweging = true;
        animatie = new Thread(this.run);
        animatie.Start();
        auto.Text = "Stop";
    }
}
```

De animatie stopt natuurlijk niet door alleen maar een variabele de waarde `false` te geven. Maar we kunnen wel de methode `run` aanpassen. In plaats van de while-opdracht eeuwig te laten duren, kunnen we in de voorwaarde van de while-opdracht de bool variabele `beweging` nauwlettend in de gaten houden:

```
private void run()
{
    while (beweging)
    {
        this.stap_Click(this, null);
        Thread.Sleep(50);
    }
}
```

Normaal gesproken zal de voorwaarde van een while-opdracht niet veranderen als er in de body geen toekenningen aan worden gedaan. Maar we zitten nu een situatie met twee parallel verlopende processen, die elkaar beïnvloeden: terwijl de animatie in `run` bezig is, kan de gebruiker

op een knop drukken, en bij het afhandelen daarvan wordt de waarde van de variabele `beweging` gelijk aan `false` gemaakt.

De waarde null

Met een toekenningsoopdracht kun je een objectverwijzing naar een object laten wijzen. Het komt soms voor dat je de verwijzing ongedaan wilt maken, dus dat de verwijzing *juist niet* naar een object moet wijzen. Voor dit doel is er een speciale constante: `null`. Dit is als het ware de nulwaarde voor verwijzingen; dit is trouwens ook de waarde die verwijzingen hebben als er nog nooit een toekenning aan is gedaan.

Het aardige is dat je met een if-opdracht kunt testen of variabelen de waarde `null` hebben. Gebruikmakend daarvan, kunnen we het programma in de vorige paragraaf herschrijven, zo dat de bool variabele `beweging` niet meer nodig is. In plaats daarvan gebruiken we de variabele `animatie`. Dat is een verwijzing naar het Thread-object. We gaan nu deze verwijzing naar `null` laten wijzen op het moment dat de gebruiker de Stop-knop indrukt. Om te weten of de animatie nog beweegt, kunnen we testen of de variabele `animatie` een andere waarde dan `null` heeft (d.w.z. daadwerkelijk naar een Thread-object wijst).

Daarom nog één keer de button-afhandelingsmethode:

```
private void auto_Click(object o, EventArgs ea)
{
    if (animatie == null)
    {
        animatie = new Thread(this.run);
        animatie.Start();
        auto.Text = "Stop";
    }
    else
    {
        animatie = null;
        auto.Text = "Start";
    }
}
```

De methode `run` gebruikt nu als voorwaarde om door te gaan dat de waarde van deze verwijzing nog niet `null` gemaakt is; zie listing 21.

blz. 107

8.4 Klasse-ontwerp en -gebruik

Ontwerp: wat is en wat kan het object, en hoe?

Als je een nieuwe klasse gaat schrijven, moet je je drie dingen afvragen:

- Wat *is* het object dat door de klasse wordt beschreven? (Het antwoord op deze vraag schrijf je in de member-variabele-declaraties bovenin de klasse.)
- Wat kun je *doen* met het object dat door de klasse wordt beschreven, en/of wat wil je ervan *weten*? (Het antwoord op deze vraag zijn de headers van de methoden en/of properties in de klasse.)
- *Hoe* kun je dingen met het object doen? (Het antwoord op deze vraag zijn de bodies van de methoden en/of properties in de klasse.)

In dit hoofdstuk hebben we voor het eerst meerdere klassen geschreven. Het is nu dus voor het eerst dat al deze vragen aan de orde zijn gekomen.

Gebruik: wat kan het object?

Als je een klasse wilt gebruiken die iemand anders heeft geschreven, bijvoorbeeld een bibliotheek-klasse, dan hoef je eigenlijk maar één ding te weten:

- Wat kun je *doen* met het object dat door de klasse wordt beschreven?

Voor het gebruik van de klasse is het niet nodig om precies te weten uit welke variabelen het object zoal bestaat.

Ook de bodies van de methoden hoef je niet te kennen om de methoden te kunnen aanroepen. Het is natuurlijk wel handig als je een informeel begrip hebt van wat de methoden doen, maar daarvoor is het niet nodig om te weten *hoe* dat precies gebeurt.

Top-down versus bottom-up ontwerp

Bij het schrijven van klassen heb je een grote vrijheid. Je kunt zelf bedenken welke methoden je gaat schrijven, en welke parameters deze zullen krijgen. Dit natuurlijk wel op voorwaarde dat je

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;

5 namespace Simulatie
{
    public class Simulatie : Form
    {
10         private Ruimte r1, r2, r3;
        private Button stap, auto;
        private Thread animatie;

        public Simulatie()
15        {
            this.ClientSize = new Size(406, 280); this.Text = "Simulatie";
            r1 = new Ruimte(); r1.Location = new Point( 10, 10); r1.Size = new Size( 80, 150);
            r2 = new Ruimte(); r2.Location = new Point(100, 10); r2.Size = new Size(296, 120);
            r3 = new Ruimte(); r3.Location = new Point( 10, 170); r3.Size = new Size(386, 100);
20            stap = new Button(); stap.Location = new Point(100, 140); stap.Text = "Stap";
            auto = new Button(); auto.Location = new Point(200, 140); auto.Text = "Start";
            this.Controls.Add(r1); this.Controls.Add(r2); this.Controls.Add(r3);
            this.Controls.Add(stap); this.Controls.Add(auto);
            this.stap.Click += stap_Click;
25            this.auto.Click += auto_Click;
        }
        private void stap_Click(object o, EventArgs ea)
        {
            r1.DoeStap();
30            r2.DoeStap();
            r3.DoeStap();
        }
        private void auto_Click(object o, EventArgs ea)
        {
35            if (animatie == null)
            {
                animatie = new Thread(this.run);
                animatie.Start();
                auto.Text = "Stop";
            }
            else
40            {
                animatie = null;
                auto.Text = "Start";
            }
        }
        private void run()
45        {
            while (animatie!=null)
            {
                this.stap_Click(this, null);
                Thread.Sleep(50);
50            }
        }
    }
}
```

Listing 21: Simulatie/Simulatie.cs

```
using System.Drawing;
using System.Windows.Forms;

namespace Simulatie
5 {
    class Ruimte : UserControl
    {
        Deeltje d1, d2, d3;

10        public Ruimte()
        {
            this.BackColor = Color.LightYellow;
            d1 = new Deeltje(Brushes.Red, 30, 40, 10, 10);
            d2 = new Deeltje(Brushes.Green, 100, 80, 5, -10);
15            d3 = new Deeltje(Brushes.Blue, 200, 60, 8, 2);
            this.Paint += this.tekenRuimte;
        }

        public void DoeStap()
20        {
            d1.DoeStap(this.Size);
            d2.DoeStap(this.Size);
            d3.DoeStap(this.Size);
            this.Invalidate();
25        }

        private void tekenRuimte(object o, PaintEventArgs pea)
        {
            Graphics gr = pea.Graphics;
30            d1.TekenDeeltje(gr);
            d2.TekenDeeltje(gr);
            d3.TekenDeeltje(gr);
        }
    }
35 }
```

Listing 22: Simulatie/Ruimte.cs

```
using System.Drawing;

namespace Simulatie
{
5   class Deeltje
    {
        private int x, y, dx, dy;
        private Brush brush;

10   public Deeltje(Brush brush, int x, int y, int dx, int dy)
        {
            this.brush = brush;
            this.x      = x;
            this.y      = y;
15   this.dx          = dx;
            this.dy     = dy;
        }
        public void DoeStap(Size hok)
        {
20   this.x += this.dx;
            this.y += this.dy;

            if (this.x < 0)
            {
25   this.x = -this.x;
                this.dx = -this.dx;
            }
            else if (this.x >= hok.Width)
            {
30   this.x = 2 * hok.Width - this.x;
                this.dx = -this.dx;
            }
            if (this.y < 0)
            {
35   this.y = -this.y;
                this.dy = -this.dy;
            }
            else if (this.y >= hok.Height)
            {
40   this.y = 2 * hok.Height - this.y;
                this.dy = -this.dy;
            }
        }

45   public void TekenDeeltje(Graphics gr)
        {
            gr.FillEllipse(this.brush, this.x-4, this.y-4, 9, 9);
        }
50 }
}
```

Listing 23: Simulatie/Deeltje.cs

ze ook op een overeenkomstige manier aanroept; als je in de header van een methode schrijft dat deze een int als parameter krijgt, dan moet je bij aanroep ook een getal als parameter meegeven. Bij het bedenken van al die methoden in al die klassen, kun je ruwweg op twee manieren te werk gaan:

- Top-down ontwerp. Je begint met de meest veelomvattende klasse, d.w.z. de klasse die een subklasse is van `Form`. Bij het schrijven van de constructor en de event-handlers merk je vanzelf dat je allerlei objecten nodig hebt. Voor alles wat maar een beetje ingewikkeld is, bedenk je een naam voor een methode die je graag zou willen hebben. Daarna ga je die methoden één voor één schrijven, waarbij je alvast weer methoden aanroept, ook al heb je die nog niet geschreven. Daar ga je mee door tot de methoden die je nog moet schrijven zo simpel zijn dat je ze direct kunt schrijven. Sleutelzinnetje: “niet vergeten om deze methode straks nog te schrijven”.
- Bottom-up ontwerp. Je begint met de simpelste klasse (in de casus in dit hoofdstuk is dat `Deeltje`), en schrijft zo veel mogelijk methoden waarvan je denkt dat ze nog wel eens nuttig zullen zijn (zoals `DoeStap`). Daarna ga je verder met de wat complexere klassen (in het voorbeeld: `Ruimte`), waarbij je de zojuist geschreven klasse goed kunt gebruiken. Zo ga je verder, totdat je de allesomvattende subklasse van `Form` kunt schrijven. Sleutelzinnetje: “wat zou je met dit soort objecten willen kunnen doen?”

Bij het behandelen van de casus hebben we beide strategieën door elkaar heen gebruikt (en zo gaat het in de praktijk ook vaak). We zijn top-down begonnen met de klasse `Simulatie`. Maar halverwege zijn we overgeschakeld op bottom-up ontwerp: de methoden van `Deeltje` hebben we geschreven voordat we ze nodig hadden in de methoden van `Ruimte`.

8.5 Subklassen

Subklasse: toevoegen van nieuwe variabelen/methoden

Door gebruik te maken van klasse-bibliotheken kun je je veel werk besparen. Des te jammerder is het dan ook, als er in de bibliotheek een klasse zit die *bijna* doet wat je wilt, maar net niet helemaal. Of je hebt een vergelijkbare klasse al eens eerder geschreven, maar ditmaal wil je een kleine uitbreiding aan de klasse maken. Wat te doen?

Vroeger, dat wil zeggen in de tijd van voor de object-georiënteerde talen (Pascal, C) maakte men in zo'n geval een kopie van het eerdere programma, om daarin vervolgens aan te passen wat er aangepast moest worden. Dat scheelde in ieder geval een boel tikwerk. Toch had deze “knip-en-plak”-strategie een belangrijk nadeel: als het oorspronkelijke programma later verbeterd werd (fouten hersteld, of de snelheid verhoogd), dan moest die verbetering in de gewijzigde kopie apart doorgevoerd worden. En als die kopie ook alweer gebruikt was als uitgangspunt voor weer een andere versie, dan moest ook die versie weer aangepast worden.

Er ontstaat, kortom, een versie-probleem: verschillende versies groeien uit elkaar en kunnen niet gemakkelijk met elkaar in overeenstemming worden gebracht. Een kleine verbetering in het oorspronkelijke programma (bijvoorbeeld: een jaartal opslaan met vier cijfers in plaats van twee) kan met zich mee brengen dat vele duizenden zoveelste-generatie gewijzigde kopieën apart ook gewijzigd moeten worden. Het jaartal-voorbeeld (de ‘millennium-bug’) heeft rond de afgelopen eeuwwisseling voor veel opschudding en miljarden aan kosten gezorgd.

Hoe moet het dan wel? Als je een uitbreiding wilt maken van een eerder geschreven klasse, dan moet je er niet een kopie van maken en die veranderen, maar kun je beter de verandering beschrijven in een subklasse van de klasse. Dat gebeurt door in de klasse-header een dubbele punt te schrijven, met daarachter de naam van de klasse waarvan je een subklasse wilt maken. We hebben daar al veel voorbeelden van gezien:

```
class Hallo      : Form {... }
class Simulatie : Form {... }
class Ruimte     : UserControl {... }
```

Maken van subklassen

De uitgebreide klasse wordt ook wel een *subklasse* van de oorspronkelijke klasse genoemd. De oorspronkelijke klasse heet, omgekeerd, de *superklasse* of *base class* van de subklasse.

Een klasse kan maar één (directe) superklasse hebben, maar een klasse kan meerdere subklassen hebben. Bijvoorbeeld, de klasse `Form` heeft zowel de klasse `Hallo` als de klasse `Simulatie` als

subklassen, maar `Hallo` heeft maar één superklasse (namelijk `Form`).

Het is wel zo, dat je van de subklasse opnieuw weer een uitbreiding kunt maken: een sub-subklasse dus van de oorspronkelijke klasse. Die sub-subklasse heeft één directe superklasse, maar is indirect ook een subklasse van de super-superklasse. Er kan dus een hele “stamboom” ontstaan van klasse-afstammelingen.

De objecten van een subklasse kunnen tevens worden opgevat als object van de superklasse. Een `Ruimte`-object bijvoorbeeld (zoals gedefinieerd in het vorige hoofdstuk) is ook een `UserControl`-object. “Maar het is niet zomaar een `UserControl`-object, nee, het is een heel bijzonder `UserControl`-object; eentje namelijk met gekleurde bolletjes erop.” Het wiskundige jargon voor dit wat overdreven geformuleerde zinnetje is: “een `Ruimte`-object is een *bijzonder geval* van een `UserControl`-object”.

Overerving van methoden en variabelen

Objecten van de subklasse mogen de methoden en properties gebruiken van de superklasse, en van de eventuele superklasse daar weer van, enzovoorts. Die methoden en properties worden, zoals dat heet, geërfd van de superklasse. Hetzelfde geldt voor de member-variabelen die in de superklasse zijn gedeclareerd: die zijn ook in elk object van de subklasse aanwezig.

Zo kan het gebeuren dat je in subklassen van `Form` gewoon de properties `Size`, `Text` en `Controls` kunt opvragen, ook al is die helemaal niet apart gedefinieerd in de subklasse. De properties zijn namelijk wel bekend in de klasse `Form`, en worden door de subklassen daar van geërfd.

Naast de zichtbaarheden `private` en `public` is er nog een derde mogelijkheid: `protected`. Members die `protected` zijn, mogen wel in subklassen, maar niet daarbuiten worden aangesproken.

Een ander voorbeeld waar overerving een rol kan spelen, nu eens niet in verband met `Form` en dergelijke, maar gewoon in je eigen klassen. Stel dat je een klasse `Bolletje` hebt gemaakt. Zo’n `Bolletje` heeft een positie en een doorsnede. Er is een methode `zetPlaats`, een methode `groe`i en een methode `teken` gedefinieerd in de klasse:

```
class Bolletje
{
    int x, y, diam;
    void zetPlaats(int x0, int y0)
    {
        x = x0; y = y0;
    }
    void groei()
    {
        diam++;
    }
    void teken(Graphics gr)
    {
        gr.DrawEllipse(Pens.Black, x, y, diam, diam);
    }
}
```

Later kun je een subklasse `KleurBol` maken, waarin de klasse `Bolletje` wordt uitgebreid met een `kleur`, en een methode `zetKleur` om de kleur vast te leggen:

```
class KleurBol : Bolletje
{
    Color kleur;
    void zetKleur(Color k)
    {
        kleur = k;
    }
}
```

Heb je nu een object van type `KleurBol`, dan kun je daar de nieuwe methode `zetKleur` van aanroepen, maar ook nog steeds de methode `zetPlaats` en `groe`i. Een `KleurBol`-object is tenslotte nog steeds een `Bolletje`, en voor het groeien en plaatsen van een bolletje maakt de eventuele kleur niet uit. Omgekeerd kan het natuurlijk niet: je kunt niet met zomaar een bolletje de methode `zetKleur` aanroepen, want zo’n bolletje heeft geen kleur. (Tenzij het toevallig een heel bijzonder bolletje is, namelijk een gekleurd bolletje, maar daar kun je niet zomaar op rekenen).

Herdefinitie van methoden

Bij het erven van de methode `teken` door de klasse `KleurBol` ontstaat er toch een probleem. Een gekleurd bolletje moet namelijk op een andere manier getekend worden dan een gewoon bolletje: bij het tekenen speelt de kleur immers een rol. Voor dit soort situaties is het toegestaan om in subklassen een methode een andere definitie te geven dan de oorspronkelijke, dus om de methode te *herdefiniëren*. In de klasse `KleurBol` kunnen we de volgende herdefinitie van `teken` zetten:

```

void teken(Graphics gr)
{
    gr.FillEllipse( new SolidBrush(kleur), x, y, diam, diam);
    gr.DrawEllipse( Pens.Black, x, y, diam, diam);
}
}

```

Het nog-niet-uitgebreide object base

Het is wel jammer dat je, zodra je besluit om een methode te herdefiniëren, je de hele body opnieuw moet schrijven. In de hergedefinieerde versie van `teken` moest, om het zwarte randje om de gekleurde bol te tekenen, de aanroep van `DrawEllipse` bijvoorbeeld opnieuw worden opgeschreven. Nou valt dat hier nog wel mee, maar als het tekenen ingewikkelder was geweest is dat toch zonde van het werk, en wat erger is: gevoelig voor versie-problematiek.

Je zou eigenlijk bij de herdefinitie van de methode `teken` de oorspronkelijke methode willen kunnen aanroepen. Dat kan echter niet met

```
this.teken(gr);
```

want dan roep je de eigen methode aan, die dan weer zichzelf aanroept, die dan weer zichzelf aanroept, die dan weer zichzelf aanroept... Nee, dat willen we niet.

Als uitweg in deze situatie is er in C# een speciale constante beschikbaar, genaamd `base`. Dit is net zoiets als `this`: het is een verwijzing naar het huidige object. Maar met dit verschil: `base` wijst naar het huidige object, *alsof hij het type van de superklasse heeft*. De herdefinitie van `teken` kan dus als volgt verlopen:

```

class KleurBol : Bolletje
{
    public override void teken(Graphics gr)
    {
        gr.FillEllipse( new SolidBrush(kleur), x, y, diam, diam);
        base.teken(gr);
    }
}

```

Methoden die in een subklasse opnieuw gedefinieerd worden, moeten worden gemarkeerd met het woord `override`. In de oorspronkelijke klasse moet de methode daarop worden voorbereid door in de header het woord `virtual` te schrijven. In dit geval dus:

```

class Bolletje
{
    public virtual void teken(Graphics gr)
    {
        gr.DrawEllipse( Pens.Black, x, y, diam, diam);
    }
}

```

Polymorfie

Objecten van een subklasse zijn acceptabel op plaatsen waar een object van die superklasse wordt verwacht. Dat geldt onder andere voor parameters. Stel dat er een methode bestaat die een `Bolletje` als parameter wil hebben – misschien omdat hij hem wil laten groeien:

```

void laatGroeien(Bolletje bol)
{
    bol.groei();
}

```

In een ander deel van het programma hebben we zowel een `Bolletje` als een `KleurBol` beschikbaar:

```

Bolletje saai;
KleurBol mooi;
saai = new Bolletje();
mooi = new KleurBol();

```

We kunnen nu `laatGroeien` aanroepen met elk van beide objecten:

```

laatGroeien(saai);
laatGroeien(mooi);

```

Omgekeerd is het niet toegestaan. Een methode als

```
void maakRood(KleurBol kb)
```



```
{    kb.zetKleur(Color.Red);
}
```

kan niet worden aangeroepen met zomaar een bolletje (die immers niet gekleurd kan worden), maar natuurlijk wel met een `KleurBol` (die dat wel kan):

```
maakRood(saai); // fout
maakRood(mooi); // goed
```

De leukste situatie is, als er een hergedefinieerde virtuele methode wordt aangeroepen. Bekijk de volgende variant op `laatGroeien`:

```
void tekenEenBolletje(Bolletje bol)
{    bol.teken(ergens);
}
```

die we aanroepen met zowel het gewone bolletje als met de `kleurbol`:

```
tekenEenBolletje(saai);
tekenEenBolletje(mooi);
```

Het gewone bolletje wordt gewoon getekend, maar de `KleurBol` wordt netjes gekleurd getekend! Daar is die methode `tekenEenbolletje` mooi ingetrapt. Hij denkt van zomaar een `Bolletje` de `teken`-methode aan te roepen, maar als die `Bolletje`-parameter een `KleurBol` blijkt te zijn (wat mag, want dat is een subklasse van `Bolletje`), dan wordt daar wel de hergedefinieerde versie van `teken` voor aangeroepen. En dat komt in de meeste gevallen goed uit.

Dit verschijnsel heet *polymorfie*. De parameter kan letterlijk “vele vormen” aannemen (*poly*=veel, *morf*=vorm), en in elk van de gevallen wordt precies de juiste versie van de methode `teken` aangeroepen.

Sealed methoden

Als je niet wilt dat een programmeur van een subklasse een virtuele functie weer opnieuw kan herdefiniëren, kun je een methode `sealed` definiëren. Sealed methodes kunnen niet in een subklasse een nieuwe invulling krijgen: verzegeld, afgelopen met de pret.

8.6 Klasse-hiërarchieën

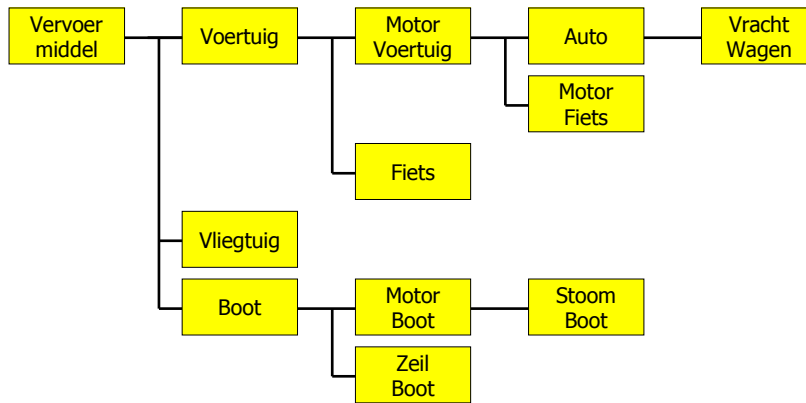
Subklasse: “is een”

Met subklassen kun je de objecten in de wereld proberen te ordenen. Als je de dubbelepunt daarbij leest als “is een”, kun je controleren of je geen denkfouten hebt gemaakt. Hier volgt een mogelijkheid om vervoermiddel-objecten te modelleren met behulp van allerlei klassen, die elkaars subklassen zijn.

```
class Vervoermiddel
class Voertuig      : Vervoermiddel
class Vliegtuig     : Vervoermiddel
class Boot          : Vervoermiddel
class MotorVoertuig : Voertuig
class Fiets         : Voertuig
class MotorBoot     : Boot
class ZeilBoot      : Boot
class StoomBoot     : MotorBoot
class Auto          : MotorVoertuig
class Vrachtwagen   : Auto
```

Met dit soort klasse-definities ontstaat een hele hiërarchie van klassen. Bij elk van de subklassen kunnen members (variabelen, properties en/of methoden worden toegevoegd, die alleen maar relevant zijn voor de betreffende klasse, en de subklassen daarvan. Bijvoorbeeld, de variabele `aantalWielen` hoort typisch thuis in `Voertuig` en niet in `Vervoermiddel`, want boten hebben geen wielen. De variabele `vliegHoogte` hoort thuis in `Vliegtuig`, en de bool variabele `belWerkt` in de klasse `Fiets`.

Zo’n hiërarchie van klassen kun je overzichtelijk in een schema weergeven:



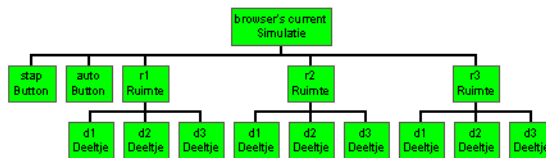
Bij het maken van een indeling in subklassen moet je als ontwerper allerlei beslissingen nemen. Er is niet één “goede” onderverdeling, maar afhankelijk van de toepassing kan een bepaalde onderverdeling handiger uitpakken dan een andere. In het voorbeeld hebben we bijvoorbeeld gekozen om de klasse **Vervoermiddel** eerst onder te verdelen naar het medium waarover het zich voortbeweegt: land, lucht of water. Pas daarna worden de klassen in verdere subklassen onderverdeeld: gemotoriseerd of niet. Dat had ook andersom gekund.

Voor sommige klassen is het ook niet altijd even duidelijk op welke plaats ze in de hiërarchie thuishoren: is een **MotorFiets** een bijzonder soort **Fiets** (namelijk met een motor), of een bijzonder soort **MotorVoertuig** (namelijk met weinig wielen)?

De onderlinge ligging van de klassen is echter wel altijd duidelijk. Een **VrachtWagen** is een **Auto**, maar een **Auto** is (lang niet altijd) een **VrachtWagen**. Een **Fiets** is een **Voertuig**, maar niet elk **Voertuig** is een **Fiets**.

Membervariabelen: “heeft een”

Werkend met diagrammen moet je wel uitkijken waar het eigenlijk over gaat. Er is nog een ander soort diagram dat wel eens getekend wordt in verband met objecten. Het gaat dan om welke objecten *onderdeel uitmaken* van andere objecten. Bijvoorbeeld in het Simulatie-programma in het vorige hoofdstuk:



Dit diagram moet je heel anders interpreteren. De lijnen moet je ditmaal van boven naar beneden lezen, met de uitspraak “heeft een”. Het diagram is heel bruikbaar om een overzicht te krijgen van de samenhang tussen de membervariabelen in een bepaald programma, maar het is geen klasse-hiërarchie. Een **Deeltje** is namelijk helemaal geen **Ruimte** (maar het maakt er onderdeel van uit). In dit diagram stellen de vakjes dan ook objecten voor, en geen klassen. Dat maakt ook dat (objecten met dezelfde) klassen meermalen kunnen voorkomen in het diagram, iets wat in een klasse-hiërarchie ook al onmogelijk is.

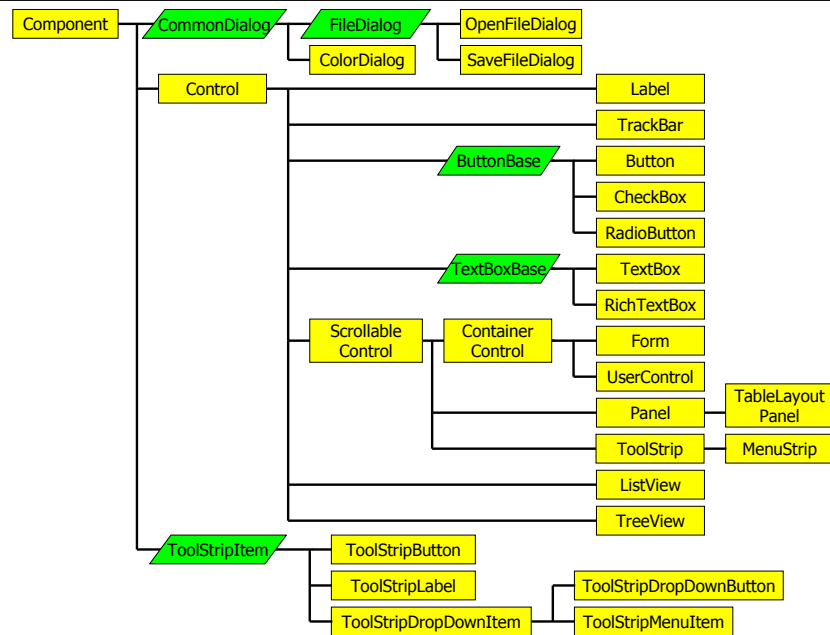
Klasse-hiërarchie in Forms

Veel klassen uit de systeem-libraries zijn geordend in hiërarchieën, vergelijkbaar met de hiërarchie van vervoermiddelen uit de vorige sectie.

Alle interactie-componenten, waaruit een grafische userinterface is opgebouwd, zijn bijvoorbeeld direct of indirect subklasse van de klasse **Component**.

Sommige componenten, zoals losse menu-items, kunnen niet zomaar overal in een GUI worden neergezet. Maar met de meeste componenten kan dat wel. We noemen ze dan een *control*, en er is dan ook een subklasse **Control** waarin dit soort vrij plaatsbare componenten worden gemodelleerd. In het schema in figuur 25 kun je zien wat voor soort controls er zoal zijn. Hierin zitten oude bekenden, zoals **Button** en **TextBox**. Sommige controls zijn subklasse van een gemeenschappelijke superklasse. Zo is **Button**, samen met twee andere button-achtige controls (**RadioButton** en **CheckBox**) een subklasse van **ButtonBase**. En **TextBoxBase** is de gemeenschappelijke superklasse

van alle textbox-achtige controls: behalve `TextBox` zelf bijvoorbeeld ook `RichTextBox`, waarin een tekst met verschillende fonts en kleuren kan worden getoond.



Figuur 25: De klassihiërarchie in `System.Windows.Forms`

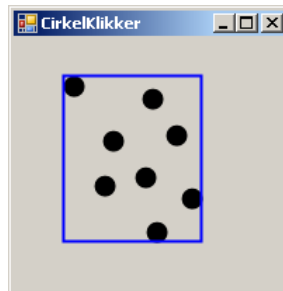
Hoofdstuk 9

Arrays en strings

9.1 Array: rij variabelen van hetzelfde type

Programma ‘CirkelKlikker’

In deze sectie bespreken we het programma CirkelKlikker. Zie figuur 26 voor een snapshot van het programma in werking. De gebruiker kan punten op het scherm aanklikken, en op die plaatsen verschijnt dan een cirkel. Bovendien wordt er een blauwe rechthoek getekend die alle aangeklikte punten omvat, maar dat laten we voorlopig even buiten beschouwing.



Figuur 26: Het programma CirkelKlikker in werking

Hoe maak je dit programma? Er zijn geen controls, dus we hoeven geen GUI op te bouwen. In plaats daarvan reageren we direct op de events van het form; in de constructor abonneren we daarom zelfgemaakte methodes op de relevante events:

```
this.Paint      += teken;
this.MouseClick += klik;
```

Die twee methodes moeten we verderop in de klasse definiëren:

```
private void klik(object o, MouseEventArgs mea)
{ ...
}
private void teken(object o, PaintEventArgs pea)
{ ...
}
```

Eerste poging: aangeklikte punt in membervariabelen

Deze twee methoden moeten met elkaar communiceren: de `klik`-methode weet (door middel van zijn `MouseEventArgs`-parameter) waar er geklikt is, en de `teken`-methode kan (door middel van zijn `PaintEventArgs`-parameter) tekenen. In methode `klik` kunnen we de aanroep van `teken` forceren door `Invalidate` aan te roepen. We kunnen voor die tijd het aangeklikte punt opslaan in membervariabelen, waar `teken` het kan terugvinden.

We krijgen dan:

```
private int x, y;
private void klik(object o, MouseEventArgs mea)
{
    this.x = mea.X;
    this.y = mea.Y;
    this.Invalidate();
}
private void teken(object o, PaintEventArgs pea)
{
    pea.Graphics.FillEllipse(Brushes.Black, this.x, this.y, 15, 15);
}
```

Maar als je het programma runt, zul je zien dat dit niet het bedoelde effect geeft: je krijgt elke keer alleen het nieuwste punt te zien, en de rest verdwijnt weer. Dit komt doordat **teken** elke keer weer met een blanco scherm begint te tekenen.

Tweede poging: meteen tekenen in klik

Je kunt je afvragen of het nou zo indirect moet, met twee methodes waarvan de ene de aanroep van de andere forceert via een derde methode. Waarom kan **klik** niet simpelweg het nieuwe punt *erbij* tekenen? Op het eerste gezicht is dat onmogelijk, omdat je in **klik** niet beschikt over het **Graphics**-object.

Wie wat verder rondkijkt in de library ontdekt echter in de klasse **Form** een methode **CreateGraphics**, waarmee je op elk gewenst moment een **Graphics**-object kunt maken dat gekoppeld is aan het window van de form. Dat geeft de mogelijkheid voor een hele nieuwe aanpak: we hebben dan de **teken**-methode niet eens meer nodig, en **klik** kan direct het nieuwe punt tekenen:

```
private void klik(object o, MouseEventArgs mea)
{
    Graphics gr = this.CreateGraphics();
    gr.FillEllipse(Brushes.Black, mea.X, mea.Y, 15, 15);
}
```

Als je dit programma runt, dan lijkt de aanpak te werken: de nieuwe cirkels verschijnen in beeld, en de oude blijven ook staan. Totdat... de gebruiker tussendoor even z'n mail gaat checken. Wanneer hij daarna weer verder wil met de cirkelklikkerij en dat window weer bovenop brengt, wacht een onaangename verrassing: alle cirkels zijn weg! Dat kan kloppen, want het bovenop leggen van het window forceert een aanroep van **teken**, die begint met een schone lei, en doet daar verder niets mee.

De aanpak faalt dus jammerlijk, en om deze reden is het geen goed idee om in event-handlers zelf met **CreateGraphics** aan de gang te gaan. Tekenen gaat alleen goed als je het doet in de methode die daarvoor bedoeld is: de event-handler van **Paint**. De methode **CreateGraphics** is uitsluitend bedoeld om dingen te tekenen die *zeer tijdelijk* op het scherm zichtbaar blijven. Denk bijvoorbeeld aan een stippelijntje terwijl de gebruiker bezig is om met de muis een blok te trekken op het scherm.

De goede aanpak: gebruik een array

We keren daarom weer terug naar de eerste poging, waarbij **klik** en **teken** ieder hun taak hebben, en communiceren via member-variabelen. Maar we hebben niet genoeg aan twee variabelen, want elk aangeklikt punt moet worden opgeslagen zodat **teken** *alle* punten kan tekenen, elke keer opnieuw.

Het is geen doen om voor elk punt aparte variabelen te declareren. We zouden dan zoiets krijgen als:

```
int x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6, x7, y7, x8, y8, x9, y9, x10, y10, x11, y11, x12, y12,
```

en in methode **teken** zouden een heleboel aanroepen van **FillEllipse** komen te staan. Gelukkig kan dat handiger, met gebruik van een *array*. Een array is een object waarmee in één klap een hele rij variabelen kan worden aangemaakt, mits die hetzelfde type hebben.

Creatie van arrays

Een array heeft veel gemeenschappelijk met een object. Als je een array wilt gebruiken moet je, net als bij een object, een variabele declareren die een verwijzing naar de array kan bevatten. Voor een array met int-waarden kan de declaratie er zo uitzien:

```
int [] tabel;
```

De vierkante haken geven aan dat we niet een losse int-variabele declareren, maar een array van int-variabelen. De variabele `tabel` zelf echter is niet de array: het is een verwijzing die ooit nog eens naar de array zal gaan wijzen, maar dat op dit moment nog niet doet:

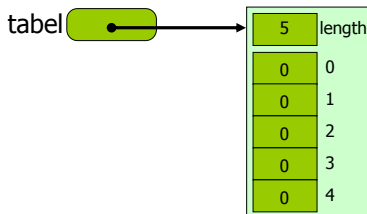
`tabel` 

Om de verwijzing daadwerkelijk naar een array te laten wijzen, hebben we een toekenningsopdracht nodig. Net als bij een object gebruiken we een `new`-expressie, maar die ziet er ditmaal iets anders uit: achter `new` staat het type van de elementen van de array, en tussen vierkante haken het gewenste aantal:

```
tabel = new int[5];
```

Het array-object dat is gecreëerd bestaat uit een int-variabele waarin de lengte van de array is opgeslagen, en uit een aantal genummerde variabelen van het gevraagde type (in dit voorbeeld ook int). De nummering begint bij 0, en daarom is het laatste nummer één minder dan de lengte. De variabelen in de array krijgen (zoals altijd bij membervariabelen) automatisch een neutrale waarde: 0 voor getallen, `false` voor bool-waarden, en `null` voor objectverwijzingen.

De situatie die hiermee in het geheugen ontstaat is:



Gebruik van array-waarden

Je kunt de genummerde variabelen aanspreken door de naam van de verwijzing te noemen, met tussen vierkante haken het nummer van de gewenste variabele. Je kunt de genummerde variabelen op deze manier een waarde geven:

```
tabel[2] = 37;
```

en je kunt de variabelen gebruiken in een expressie:

```
x = tabel[2] + 5;
```

Het zijn, kortom, echte variabelen.

Verder is er een property `Length` waarmee je de lengte kunt opvragen en gebruiken in een expressie, bijvoorbeeld

```
if (tabel.Length < 10) iets
```

Je kunt deze property niet veranderen: het is een *read-only* property. De lengte van een eenmaal gecreëerde array ligt dus vast; je kunt de array niet meer langer of korter maken.

De echte kracht van arrays ligt in het feit dat je het nummer van de gewenste variabele met een expressie kunt aanduiden. Neem bijvoorbeeld het geval waarin alle array-elementen dezelfde waarde moeten krijgen. Dat kan met een hele serie toekenningsopdrachten:

```
tabel[0] = 42;
tabel[1] = 42;
tabel[2] = 42;
tabel[3] = 42;
tabel[4] = 42;
```

maar dat is, zeker bij lange arrays, natuurlijk erg omslachtig. Je kunt de regelmaat in deze serie opdrachten echter uitbuiten, door op de plaats waar het volgnummer (0, 1, 2, 3 en 4) staat, een variabele te schrijven. In een `for`-opdracht kun je deze variabele dan alle gewenste volgnummers laten langslopen. De `Length`-property kan mooi dienen als bovengrens in deze `for`-opdracht:

```
int nummer;
for (nummer=0; nummer<tabel.Length; nummer++)
    tabel[nummer] = 42;
```

Cirkelklikker met arrays

In plaats van de twee losse variabelen declareren we in `CirkelKlikker` twee arrays:

```
int[] xs, ys
```

In de constructor worden de array-objecten gecreëerd. We moeten daarbij de lengte van de array vastleggen. Dat zet dus een bovengrens op het maximaal aanklikbare punten. Wat zullen we daarvoor nemen: 100? of 1000? We moeten een keus maken, maar om deze keus in een latere versie van het programma ('CirkelKlikker Pro') gemakkelijk te kunnen herzien, maken we met `const` een constante die de grens vastlegt:

```
const int maxAantal = 100;
```

Met gebruikmaking van deze constante worden de arrays gemaakt in de constructormethode:

```
public CirkelKlikker()
{
    this.xs = new int[maxAantal];
    this.ys = new int[maxAantal];
}
```

Nu hebben we nog een variabele nodig die aangeeft tot hoe ver de arrays al in gebruik zijn. Aan het begin is er nog geen enkele waarde in gebruik, dus declareren en initialiseren we:

```
int aantal = 0;
```

Maar daar komt verandering in als er een punt wordt aangeklikt: op de eerste vrije plaats worden de coördinaten van het aangeklikte punt bewaard, en daarna wordt het aantal opgehoogd.

```
public void klik(object sender, MouseEventArgs mea)
{
    this.xs[aantal] = mea.X;
    this.ys[aantal] = mea.Y;
    this.aantal++;
    this.Invalidate();
}
```

De methode `teken` kan nu alle waarden van de array langslopen, althans zo ver als nodig is volgens de variabele `aantal`. Daartoe gebruikt `teken` een eigen tellertje `t`, die loopt van 0 tot `aantal`. In `teken` mag het `aantal` natuurlijk niet worden veranderd, want dan zou de administratie van aangeklikte punten niet meer kloppen.

```
private void teken(object sender, PaintEventArgs pea)
{
    Graphics gr = pea.Graphics;
    for (int t = 0; t < aantal; t++)
    {
        gr.FillEllipse(Brushes.Black, this.xs[t], this.ys[t], 15, 15);
    }
}
```

Arrays raken vol

In listing 24 is het programma verder uitgewerkt, waarbij de cirkels netjes gecentreerd worden door bij het tekenen rekening te houden met hun diameter en straal. De waarde van `diameter` wordt als constante gedefinieerd, om hem gemakkelijk te kunnen aanpassen. De waarde van `straal` is ook een constante, die wordt uitrekend aan de hand van de diameter, zodat hij vanzelf meeveranderd mochten we ooit de diameter aanpassen.

blz. 120

Er is één belangrijk nadeel aan arrays: je moet bij het `new` maken opgeven hoe groot hij is. De bijbehorende geheugenruimte wordt meteen in beslag genomen, dus je moet het ook niet nodeloos op honderdmiljoen begroten. In het programma speculeren we er op dat de gebruiker nooit meer dan 100 cirkels aan zal klikken. Maar wat gebeurt er als hij het toch doet? Dan zal methode `klik` de array aanspreken op positie 100, en die bestaat niet. Het gevolg is een runtime exception, en als we die niet opvangen wordt het programma afgebroken.

Dat willen we niet hebben, en daarom test de methode `klik` voor de zekerheid of er nog ruimte is om het nieuwe punt op te slaan:

```
if (this.aantal < maxAantal)
```

Zo niet, dan wordt het punt niet opgeslagen. Dat is ook niet leuk voor de gebruiker, maar in ieder geval crasht het programma dan niet.

```

using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

5 namespace CirkelKlikker
{
    class CirkelKlikker : Form
    {
        const int maxAantal = 100;
10      const int diameter = 15;
        const int straal = diameter / 2;

        private int[] xs, ys;
        private int aantal;

15      public CirkelKlikker()
        {
            this.xs = new int[maxAantal];
            this.ys = new int[maxAantal];
            this.aantal = 0;
20      this.Text = "CirkelKlikker";
            this.Paint += teken;
            this.MouseClick += klik;
        }
        private void klik(object sender, MouseEventArgs mea)
25      {
            if (this.aantal < maxAantal)
            {
                this.xs[aantal] = mea.X;
                this.ys[aantal] = mea.Y;
30      this.aantal++;
                this.Invalidate();
            }
        }
        private void teken(object sender, PaintEventArgs pea)
35      {
            Graphics gr = pea.Graphics;
            gr.SmoothingMode = SmoothingMode.AntiAlias;
            for (int t = 0; t < aantal; t++)
            {
40      gr.FillEllipse(Brushes.Black
                        , this.xs[t]-straal, this.ys[t]-straal
                        , diameter      , diameter
                        );
            }
45      if (aantal > 1)
            {
                int minX = ArrayBieb.Kleinste(this.xs, this.aantal) - straal;
                int maxX = ArrayBieb.Grootste(this.xs, this.aantal) + straal;
                int minY = ArrayBieb.Kleinste(this.ys, this.aantal) - straal;
50      int maxY = ArrayBieb.Grootste(this.ys, this.aantal) + straal;
                gr.DrawRectangle( new Pen(Color.Blue,2), minX, minY, maxX-minX, maxY-minY);
            }
        }
55 }

```

```
namespace CirkelKlikker
{
    public class ArrayBieb
    {
5        public static int Kleinste(int[] a, int n)
        {
            int res = a[0];
            for (int t = 1; t < n; t++)
                if (a[t] < res)
10                res = a[t];
            return res;
        }
        public static int Grootste(int[] a, int n)
        {
15            int res = a[0];
            for (int t = 1; t < n; t++)
                if (a[t] > res)
                    res = a[t];
            return res;
20        }
        public static int Kleinste(int[] a)
        {
            return ArrayBieb.Kleinste(a, a.Length);
        }
25        public static int Grootste(int[] a)
        {
            return ArrayBieb.Grootste(a, a.Length);
        }
    }
30 }
```

Listing 25: CirkelKlikker/ArrayBieb.cs

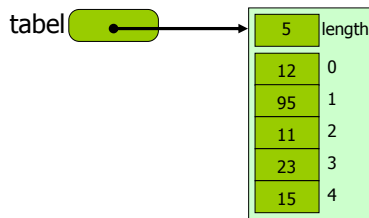
Zoeken van de grootste en kleinste waarde

Nu moeten we de *bounding box* rondom de cirkels nog tekenen. Daarvoor hebben we de kleinste en grootste waarde nodig uit de twee arrays. Om het werk in **teken** niet te ingewikkeld te maken gaan we een methode schrijven die de kleinste waarde van een array zoekt. Die kunnen we dan twee keer aanroepen: een keer voor de array **xs**, en een keer voor **ys**. Misschien komen deze methodes ook nog wel eens in een ander programma van pas.

Met een vooruitziende blik brengen we deze methodes daarom onder in een aparte klasse **ArrayBieb**, waarin we nuttige hulp-functies gaan kunnen verzamelen. De methodes zijn **static**, en er zijn in deze klasse dus geen membervariabelen gedeclareerd.

De methode **kleinste** krijgt een array als parameter. Dat gebeurt gewoon zoals je zou verwachten: een declaratie van een array in de header van de methode. Het is eigenlijk niet zozeer de array als geheel die wordt meegegeven, als wel de verwijzing ernaartoe. De array zelf blijft in het geheugen staan waar hij stond toen hij werd gecreëerd.

De parameter krijgt bij aanroep van de methode een waarde; in de body methode wordt *niet* een **new** array gemaakt, want we willen juist werken met de array die de aanroeper heeft meegegeven. We mogen hopen dat de persoon die de methode aanroept inderdaad een verwijzing naar een bestaande array meegeeft (en niet de waarde null), en dat bovendien de elementen van de array al een waarde hebben. De situatie zou bijvoorbeeld als volgt kunnen zijn:



De tweede parameter van **kleinste** geeft aan tot hoe ver we de array willen doorzoeken.

```
public static int Kleinste(int[] tabel, int n)
```

Zolang we de waarde nog niet geïnspecteerd hebben, gaan we er voorlopig van uit dat de waarde met volgnummer 0 de kleinste is.

```
{    int resultaat;
    resultaat = tabel[0];
```

Met een for-opdracht lopen we nu alle elementen langs. De nulde hebben we al gebruikt, dus we kunnen beginnen bij nummer 1. Bij elk element controleren we of die misschien kleiner is dan wat we tot nu toe dachten dat de kleinste was. Als dat zo is, passen we de waarde van de resultaatwaarde aan.

```
    int t;
    for (t=1; t<n; t++)
        if (tabel[t] < resultaat)
            resultaat = tabel[t];
```

Na afloop van de for-opdracht kunnen we de resultaatwaarde veilig opleveren, want die is getoetst aan alle waarden in de array.

```
    return resultaat;
}
```

Deze methode kunnen we nu twee keer aanroepen om de kleinste waarde van de twee arrays te bepalen. Omdat het een statische methode is, roepen we hem aan met de naam van de klasse voor de punt:

```
int minX = ArrayBieb.Kleinste(xs, aantal);
int minY = ArrayBieb.Kleinste(ys, aantal);
```

Op soortgelijke manier kunnen we een methode **Grootste** schrijven, die de maximale arraywaarde bepaalt. Met behulp van de gevonden minima en maxima kunnen we de bounding box tenslotte gemakkelijk tekenen:

```
gr.DrawRectangle(pen, minX, minY, maxX-minX, maxY-minY );
```

Overloaded methoden

De klasse met de hulp-methoden voor arrays, `ArrayBieb`, staat in listing 25. In de klasse staan twee versies van de methoden `Kleinste` en `Grootste`. Naast de methode die een array en een getal dat aangeeft tot hoe ver de array is gevuld meekrijgen, zijn er ook twee methoden die alleen de array meekrijgen. Die worden in dit programma weliswaar nog niet gebruikt, maar `ArrayBieb` heeft de pretentie om een algemeen bruikbare bibliotheek te worden, dus wie weet hebben we later toch nog eens plezier van deze methoden.

blz. 121

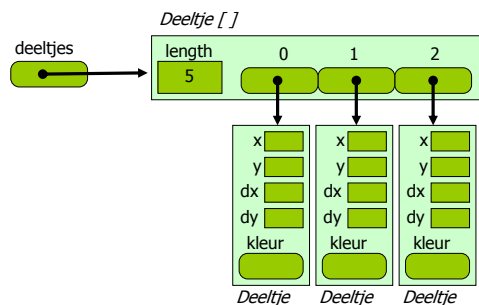
Het is in C# mogelijk om in een klasse meerdere methoden met dezelfde naam te definiëren, mits (het aantal en/of de types van) de parameters verschillen. Door te kijken naar (het aantal en/of de types van) de argumenten die bij de aanroep worden meegegeven kan de compiler een keuze maken uit de methode die wordt aangeroepen.

Het verschijnsel dat er verschillende methoden met dezelfde naam zijn heet *overloading*. Dat had je natuurlijk al veel eerder gemerkt, want de standaardlibraries zitten vol met overloaded methoden. Bijvoorbeeld bij `DrawRectangle` kun je de coördinaten als losse argumenten meegeven, maar er is ook een versie van `DrawRectangle` die ze verpakt in een `Rectangle`-object meekrijgt.

9.2 Syntax van arrays

Arrays van objecten

De elementen van de array kunnen van elk gewenst type zijn. Dat kan een primitief type zijn zoals `int`, `double`, `bool` of `char`, maar ook een object-type. Zo kun je bijvoorbeeld een array maken van `Strings`, van `Buttons`, van `TextFields`, of van objecten van een zelfgemaakte klasse, zoals `Deeltje`. Hier is een array met `Deeltje`-objecten, die hoofdstuk 9 gebruikt had kunnen worden in plaats van losse `Deeltje`-variabelen `d1`, `d2` en `d3`:



De verwijzings-variabele kan worden gedeclareerd met:

```
Deeltje[] deeltjes;
```

De eigenlijke array kan worden gecreëerd met

```
deeltjes = new Deeltje[3];
```

Maar pas op: hiermee is weliswaar het array-object gecreëerd, maar nog niet de individuele `deeltjes`! Dat moet apart gebeuren in een `for`-opdracht, die elk `deeltje` apart creëert:

```
int t;
for (t=0; t<deeltjes.Length; t++)
    deeltjes[t] = new Deeltje();
```

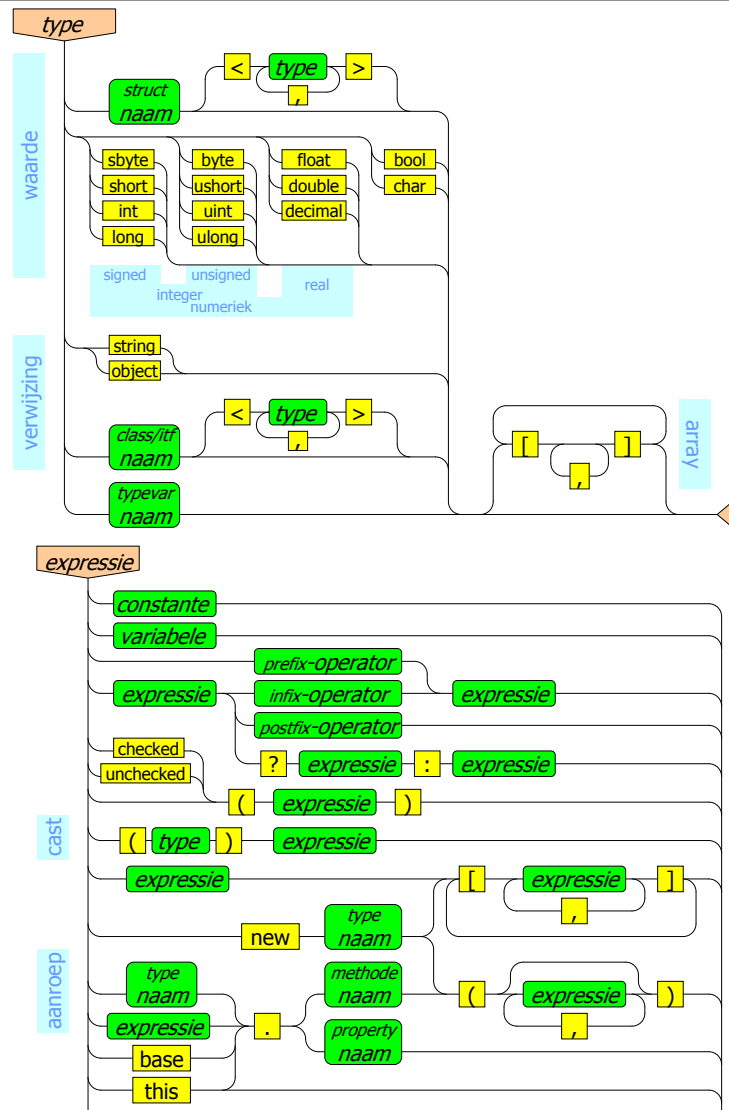
Syntax van arrays

De diverse notaties die arrays mogelijk maken zijn specifieke bijzonderheden in de C#-syntax. De aanwezigheid van arrays is dan ook veel fundamenteeler dan zomaar een extra klasse in de library. De mogelijkheid van het lege paar haken in een declaratie zoals

```
int [] tabel
```

wordt geschapen in het syntax-diagram van 'type' (zie figuur 27). Na het eigenlijke type (een van de vijftien standaardtypen of een struct- of klasse-naam) kan er nog een paar vierkante haken volgen. Tussen de vierkante haken staat niets, of een of meer losse komma's.

Het aanmaken van een array met `new`, en het opzoeken van een bepaalde waarde in de array wordt mogelijk gemaakt door de syntax van 'expressie' (zie weer figuur 27).



Figuur 27: Syntax van 'type' en 'expressie' maakt arrays mogelijk

Let op het verschil tussen `new-type` gevolgd door iets tussen ronde haken (de aanroep van de constructormethode van een klasse) en `new-type` gevolgd door iets tussen vierkante haken (de aanmaak van een array, maar –ingeval het een array van objecten is– nog niet van de objecten in die array).

Initialisatie van arrays

We wijzen er nog eens op dat, als je een array declareert, je een *verwijzing* naar een array-object hebt, en nog niet de array zelf. Bijvoorbeeld bij de volgende array van strings:

```
string[] dagnamen;
```

ontstaat de eigenlijke array pas door de toekenning:

```
dagnamen = new string[7];
```

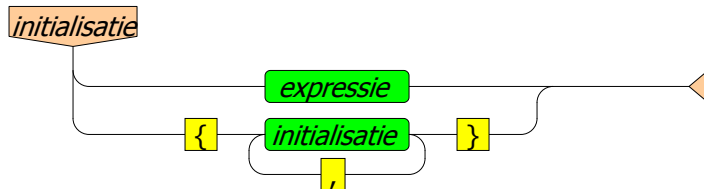
Nu bestaat de array wel, maar die zeven strings hebben ieder nog geen waarde gekregen. Dat kan nog een heel gedoe worden:

```
dagnamen[0]="maandag"; dagnamen[1]="dinsdag"; dagnamen[2]="woensdag"; dagnamen[3]="donderdag";
dagnamen[4]="vrijdag"; dagnamen[5]="zaterdag"; dagnamen[6]="zondag";
```

Gelukkig is er speciale notatie om arrays met constanten te initialiseren. Zelfs de `new` is dan niet meer nodig. De initialisatie moet meteen bij de declaratie gebeuren, door de elementen op te sommen tussen accolades:

```
string[] dagnamen = { "maandag", "dinsdag", "woensdag", "donderdag"
    , "vrijdag", "zaterdag", "zondag" };
```

Deze notatie hebben we te danken aan de syntax van initialisaties:



De opsomming tussen accolades is dus *niet* een vorm van expressies, en kan dus ook niet gebruikt worden in een toekennings-opdracht. De opsomming is een vorm van initialisatie, en kan dus alleen maar gebruikt worden direct bij de declaratie.

Handig is het wel. De notatie kan ook gebruikt worden voor de initialisatie van arrays met getallen:

```
int[] maandlengtes = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Twee- en meerdimensionale arrays

Bij de declaratie van een array mag er tussen de vierkante haken een *komma* staan. Je maakt dan een *tweedimensionale* array: in plaats van een rijtje variabelen heb je dan een blok variabelen. Vergelijk:

```
int[] tabel = new int[10];
int[,] blok = new int[6,8];
```

Je maakt hiermee een blok van $6 \times 8 = 48$ variabelen. Bij het aanspreken van de losse elementen moet je in een tweedimensionale array ook twee indexen opgeven, zoals in `blok[x,y]`. De lengtes van de aparte dimensies (6 en 8 in het voorbeeld) kun je opvragen door aanroep van `GetLength(0)` en `GetLength(1)`.

Arrays met meer dan twee dimensies zijn ook mogelijk, maar worden in de praktijk weinig gebruikt. Het geheugengebruik loopt ook wel erg op:

```
int[,,] kubus = new int[100][100][100];
```

maakt een object met een miljoen variabelen aan.

Als je goed kijkt naar de syntax van 'type' in figuur 27 dan zie je dat er ook meerdere hakenparen achter het type mogen staan. Dit is gedaan in:

```
int[][] rafelig = new int[3][];
rafelig[0] = new int[5];
rafelig[1] = new int[2];
rafelig[2] = new int[10];
```

Ook deze array geeft een tweedimensionaal gevoel, maar ditmaal zijn de rijen van `rafelig` niet allemaal even lang. In feite is het eigenlijk een array-van-arrays.

9.3 String: een immutable array characters

De klasse String

In de klasse `String` zitten onder andere de volgende methoden en properties:

- `int Length`: bepaalt de lengte van de string
- `string Substring(int x, int n)`: selecteert `n` letters van de string vanaf positie `x`, en levert die op als resultaat
- `string Concat(object s)`: plakt een tweede string erachter, en levert dat op als resultaat. Als de parameter iets anders is dan een `string` wordt er eerst de methode `ToString` van aangeroepen.
- `bool Equals(string s)`: vergelijkt de string letter-voor-letter met een andere string
- `int IndexOf(string s)`: bepaalt op welke plek `s` in `this` voor het eerst voorkomt (en levert `-1`) als dat nergens is)

- `string Insert(int p, string s: voegt s in op positie p`

In alle gevallen waar een `string` wordt opgeleverd, is dat een *nieuwe* string. De oorspronkelijke string wordt ongemoeid gelaten. Dat is een bewuste keuze van de ontwerpers van de klasse geweest: een `string`-object is *immutable*: eenmaal geconstrueerd wordt de inhoud van het object nooit meer veranderd. Wel kun je natuurlijk een aangepast kopie maken, en dat is wat `Substring`, `Concat` en `Insert` doen.

Behalve methoden worden er in de klasse `string` ook *operatoren* gedefinieerd:

- De operator `+` met als linker argument een string doet hetzelfde als methode `Concat`. Dit maakte het in onze allereerste programma's al mogelijk om strings samen te voegen: `"Hallo "+naam`.
- De operator `==` is hergedefinieerd en doet op strings hetzelfde als methode `Equals`. Dat is handig gedaan: zonder deze herdefinitie zou `==`, zoals altijd op objecten, de object-verwijzingen vergelijken. En er zijn situaties waarin twee string-verwijzingen naar verschillende objecten wijzen die dezelfde inhoud hebben. Dankzij de herdefinitie geeft `==` dan toch het antwoord `true`, zoals je waarschijnlijk ook zou verwachten. (In zuster-talen zoals Java en C gebeurt dit niet, dus daar moet je extra uitkijken bij het vergelijken van strings!).

Met de methode `Substring` kun je een deel van de string selecteren, bijvoorbeeld de eerste vijf letters:

```
string kop;
kop = s.Substring(0,5);
```

De telling van de posities in de string is, net als by arrays, merkwaardig: de eerste letter staat op positie 0, de tweede letter op positie 1, enzovoorts. Als parameters van de methode `Substring` geef je de positie van de eerste letter die je wilt hebben, en het aantal letters. Dus de aanroep `s.Substring(3,2)` geeft de letters op positie 3 en 4.

Je kunt de eerste letter van een string te pakken krijgen met een aanroep als:

```
string voorletter;
voorletter = s.Substring(0,1);
```

Het resultaat is dan een string met lengte 1.

Er is echter nog een andere manier om losse letters uit een string te pakken. De klasse `string` heeft namelijk de notatie om met vierkante haken, die in arrays ook gebruikt wordt om een element aan te spreken, opnieuw gedefinieerd. Dat kan; in feite gaat het om een bijzonder soort member, namelijk een *indexer*.

Hoewel een string niet een array is, begint het er voor de programmeur wel erg veel op te lijken, want je kunt een bepaalde letter van een string pakken zoals je ook een element van een array ophaalt. Je kunt in een string echter *niet* een letter veranderen: een string is een *immutable* object.

De losse letters van een string zijn van het primitieve type `char`, die je dus direct in een variabele kunt opslaan:

```
char eerste;
eerste = s[0];
```

Een van de voordelen van `char` boven string-objecten met lengte 1, is dat char-waarden direct in het geheugen staan, en dus niet de indirecte object-verwijzing nodig hebben.

Het primitieve type char

Net als alle andere primitieve types kun je char-waarden opslaan in variabelen, meegeven als parameter aan een methode, opleveren als resultaatwaarde van een methode, onderdeel laten uitmaken van een object, enzovoorts.

Er is een speciale notatie om constante char-waarden in het programma aan te duiden: je tikt gewoon het gewenste letterteken, en zet daar *enkele* aanhalingstekens omheen. Dit ter onderscheiding van string-constanten, waar dubbele aanhalingstekens omheen staan:

```
char sterretje;
string hekjes;
sterretje = '*';
hekjes    = "####";
```

Tussen enkele aanhalingstekens mag maar één symbool staan; tussen dubbele aanhalingstekens

mogen meerdere symbolen staan, maar ook één symbool, of zelfs helemaal geen symbolen.

Geschiedenis van char

Het aantal verschillende symbolen dat in een char-variabele kan worden opgeslagen is in de geschiedenis (en in verschillende programmeertalen) steeds groter geworden:

- In de jaren '70 van de vorige eeuw dacht men aan $2^6 = 64$ verschillende symbolen wel genoeg te hebben: 26 letters, 10 cijfers en 28 leestekens. Dat er op die manier geen ruimte was om hoofd- en kleine letters te onderscheiden nam men voor lief.
- In de jaren '80 werden meestal $2^7 = 128$ verschillende symbolen gebruikt: 26 hoofdletters, 26 kleine letters, 10 cijfers, 33 leestekens en 33 speciale tekens (einde-regel, tabulatie, piep, enzovoorts). De volgorde van deze tekens stond bekend als ASCII: de American Standard Code for Information Interchange. Dat was leuk voor Amerikanen, maar Françaises, Deutsche Mitbürger, en inwoners van España en de Fær-Øer denken daar anders over.
- In de jaren '90 kwamen er dan ook coderingen met $2^8 = 256$ symbolen in zwang, waarin ook de meest voorkomende land-specifieke letters een plaats vonden. De tekens 0–127 zijn hetzelfde als in ascii, maar de tekens 128–255 werden gebruikt voor bijzondere lettertekens die in een bepaalde taal voorkwamen. Het is duidelijk dat in Rusland hier een andere keuze werd gemaakt dan in Griekenland of India.

Er onstonden dus verschillende zogeheten *code pages*. In west-Europa werd de codepage *Latin1*, met tekens uit Duits, Frans, Spaans, Nederlands (de ij als één teken) en Scandinavische talen. Voor oost-Europa was er een andere codepage (Pools en Tsjechisch hebben veel bijzondere accenten waarvoor in Latin1 geen plaats meer was). Grieks, Russisch, Hebreeuws en het Indiase Devangari-alfabet hadden ieder een eigen codepage.

Daarmee kon je redelijk uit de voeten, maar het werd lastig als je in één file teksten in meerdere talen tegelijk wilde opslaan (woordenboeken!). Ook talen met meer dan 128 bijzondere tekens (Chinees!) hadden een probleem.

- In de jaren '00 werd daarom de codering opnieuw uitgebreid tot een tabel met $2^{16} = 65536$ verschillende symbolen. Daarin konden royaal alle alfabetten van de wereld, plus allerlei leestekens en symbolen, een plek krijgen. (Voor zeer bijzondere symbolen in bepaalde wetenschapsgebieden is er nog een uitbreiding met $2^{21} = 2$ miljoen posities mogelijk). Deze codering heet *Unicode*. De eerste 256 tekens van Unicode komen overeen met de Latin1-codering, dus we boffen maar weer in west-Europa.

In C# worden char-waarden opgeslagen via de Unicode-codering. Niet op alle computers en/of in alle fonts kun je al deze tekens daadwerkelijk weergeven, maar onze programma's hoeven tenminste niet te worden aangepast zodra dat wel het geval wordt.

Aanhalingstekens

Bij het gebruik van strings en char-waarden is het belangrijk om de aanhalingstekens pijnlijk precies op te schrijven. Als je ze vergeet, is wat er tussen staat namelijk geen letterlijke tekst meer, maar een stukje C#-programma. En er is een groot verschil tussen

- de letterlijke string "hallo" en de variabele-naam hallo
- de letterlijke string "bool" en de type-naam bool
- de letterlijke string "123" en de int-waarde 123
- de letterlijke char-waarde '+' en de optel-operator +
- de letterlijke char-waarde 'x' en de variabele-naam x
- de letterlijke char-waarde '7' en de int-waarde 7

Informatici hebben iets met aanhalingstekens. Grapjes en woordspelingen die gebaseerd zijn op de verwarring die door het weglaten van aanhalingstekens ontstaat, zijn dan ook erg populair in kringen van informatici (en worden daarbuiten vaak helemaal niet begrepen). In de casus met de geheime sleutel in sectie 7.2 stond zo'n woordspeling. Had je die gezien?

blz. 85

Speciale char-waarden

Speciale lettertekens zijn, juist omdat ze speciaal zijn, niet op deze manier aan te duiden. Voor een aantal speciale tekens zijn daarom aparte notaties in gebruik, gebruik makend van het omgekeerde-schuine-streep-teken (*backslash*):

- '\n' voor het einde-regel-teken,
- '\t' voor het tabulatie-teken.

Dat introduceert een nieuw probleem: hoe die backslash dan zelf weer aan te duiden. Dat gebeurt door twee backslashes achter elkaar te zetten (de eerste betekent: “er volgt iets speciaals”, de tweede betekent: “het speciale symbool is nu eens *niet* speciaal”). Ook het probleem hoe de aanhalingstekens *zelf* aan te duiden is op deze manier opgelost:

- `'\\'` voor het backslash-teken,
- `'\''` voor het enkele aanhalingsteken,
- `'\"'` voor het dubbele aanhalingsteken.

Er staan in deze gevallen in de programma-sourcetekst dus weliswaar twee symbolen tussen de aanhalingstekens, maar samen duiden die toch één char-waarde aan.

Rekenen met char

De symbolen in de Unicode-tabel zijn geordend; elk symbool heeft zijn eigen rangnummer. Het volgnummer van de letter `'A'` is bijvoorbeeld 65, dat van de letter `'a'` is 97. Let op dat de code van het symbool `'0'` niet 0 is, maar 48. Ook de spatie heeft niet code 0, maar code 32. Het symbool dat wel code 0 heeft, is een speciaal teken dat geen zichtbare representatie heeft.

Je kunt het code-nummer van een `char` te weten komen door de char-waarde toe te kennen aan een int-variabele:

```
char c; int i;
c = '*';
i = c;
```

of zelfs direct

```
i = '*';
```

Dit kan altijd; er zijn tenslotte maar 65536 verschillende symbolen, terwijl de grootse `int` meer dan 2 miljard is.

Andersom kan de toekenning ook, maar dan moet je voor de int-waarde nog eens extra garanderen dat je accoord gaat met onverwachte conversies, mocht de int-waarde te groot zijn. Die garantie geef je door voor de int-waarde tussen haakjes te schrijven dat je er een `char` van wilt maken:

```
c = (char) i;
```

Je kunt op deze manier ‘rekenen’ met symbolen: het symbool na de `'z'` is `(char)('z'+1)`, en de hoofdletter `c` is de `c-'A'+1`-de letter van het alfabet.

Deze “garantie”-notatie heet een *cast*. We hebben hem ook gebruikt om bij conversie van double naar int-waarde te garanderen dat we accoord gaan met afronding van het gedeelte achter de komma:

```
double d; int i;
d = 3.14159;
i = (int) d;
```

9.4 Arrays als turftabel

Arrays kris-kras indiceren

blz. 116

In sectie 9.1 hebben we de elementen van de arrays steeds op volgorde gebruikt: methode `klik` zette aan het eind een waarde erbij, en methode `teken` liep de elementen van begin tot eind langs. Maar arrays zijn veel flexibeler: je kunt de elementen naar willekeur eruit pikken: in het midden, dan weer eens aan het begin, dan op de 17e plaats. Een array kun je dus kris-kras benaderen, je hebt *random access*.

Turven

Random access is wat je in feite doet als je aantallen *turft*: je hebt een tabel waarin de turfjes staan, en je zet nu eens hier, dan weer daar een turfje erbij. In deze sectie bekijken we het programma `LetterTeller`. De gebruiker kan een tekst intikken, en het programma berekent dan hoe vaak elke letter in de tekst voorkomt. Met andere woorden: het programma turft de letters in de tekst. Het resultaat wordt op twee manieren gepresenteerd: als getallen, en als staafdiagram. In figuur 28 zie je het programma in werking.

Scheiden van inhoud en userinterface

blz. 132

In listing 26 staat het deel van het programma dat verantwoordelijk is voor de GUI. Omdat er



Figuur 28: Het programma LetterTeller in werking

maar een paar controls zijn is het met de hand geschreven, en er is dus geen aparte partial class met methode `InitializeComponent`. Met een abonnement op event `TextChanged` zorgen we ervoor dat het antwoord elke keer opnieuw wordt berekend zodra de gebruiker iets aan de tekst verandert. Zo blijft de telling steeds up-to-date.

De event-handler is expres zo kort mogelijk gehouden: al het eigenlijke werk wordt uitgevoerd in methodes. Hier wordt alleen de regie gedaan:

```
TurfTab tabel = new TurfTab();
tabel.Turf(invoer.Text);
uitvoer.Text = tabel.ToString();
```

We nemen dus aan dat er een klasse `TurfTab` bestaat (moeten we nog schrijven), die een methode `Turf` heeft waarmee de letters van een string (door de gebruiker ingetikt in textbox `invoer`) geturft kunnen worden. Verder moet er een methode `ToString` zijn die de hele inhoud van een turftabel omzet naar een handzame string, die aan de gebruiker getoond kan worden op een label `uitvoer`. De laatste regel is helemaal een staaltje van 'uitbesteden van werk':

```
diagram.Waardes = tabel.Waardes;
```

We nemen aan dat de turftabel een property `Waardes` heeft die we kunnen opvragen, en dat het object `diagram` zo'n property ook heeft, maar dan juist om te veranderen. Hierbij is `diagram` gedeclareerd als `StaaDiagram`, een (nog te schrijven) usercontrol waarmee we een staafdiagram gaan visualiseren.

Het werk is dus verdeeld over vier klassen:

- **Program**, de gebruikelijke klasse met alleen een methode `Main` die de boel moet opstarten (listing weggelaten)
- **LetterTeller**, de subklasse van `Form` waarin de GUI wordt opgebouwd (listing 26) blz. 132
- **StaaDiagram**, een usercontrol die uitsluitend verantwoordelijk is voor het tekenen van een staafdiagram (listing 28) blz. 134
- **TurfTab**, een klasse die niets te maken heeft met GUI's, en alleen het eigenlijke turfwerk hoeft te doen (listing 27). blz. 133

Het is een goede gewoonte om het werk op deze manier te scheiden. Niet alleen wordt het er

overzichtelijker van, maar ook hoeven we later alleen maar de klasse `LetterTeller` te veranderen als we nog eens een andere GUI zouden willen maken (web-interface? mail-gebaseerd?). De klassen `StaaDiagram` en `TurfTab` zijn algemeen bruikbaar, en komen misschien in andere programma's nog eens van pas.

De klasse `TurfTab`

Voor het eigenlijke turfwerk beschikt de klasse `TurfTab` (of liever gezegd: een object van die klasse) over een array waarin de 26 verschillende letters geturft kunnen worden. (Het programma is schaamteloos Nederlandstalig georiënteerd: letters met accenten, en andere –voor ons– bijzondere letters worden gewoon genegeerd). Deze variabelen zijn echter alleen voor ‘private’ gebruik binnen de klasse: de buitenwereld mag alleen de public methoden aanroepen.

Er zijn drie public methoden: de constructor die de array aanmaakt, de methode `Turf` die de letters van een string turft, en de methode `ToString` die het resultaat in een string inpakt. De methode `Turf` loopt de als parameter aangeboden string langs, en roept voor elke letter een andere methode `turf` aan. Door steeds weer een kleinere portie van het werk uit te besteden aan andere methoden, wordt het werk op een zeker moment zo simpel dat het direct uit te voeren is: de methode `turf` controleert of de letter binnen het bereik A–Z valt, berekent (door van het codegetal van de letter het codegetal van ‘A’ af te trekken) een getal tussen 0 en 26, en gebruikt dat getal als index in de array met tellers. De uitgekozen teller wordt opgehoogd, evenals de totaal-teller.

De methode `ToString` heeft geen random access op de array nodig, maar loopt de 26 waarden achtereenvolgens langs. De methode heeft de aanpak ‘bouw met een for-opdracht een resultaat op’: elke ronde wordt de resultaat-string een regel langer. Het lastigste is nog de expressie `(char)(t+'A')` waarmee van een getal tussen 0 en 26 weer een letter tussen ‘A’ en ‘Z’ wordt gemaakt.

In de header van de methode `ToString` staat de modifier `override`. Dit is nodig omdat de methode `ToString` al in een superklasse van `TurfTab` wordt gedefinieerd (namelijk in de superklasse aller superklassen: `object`). Omdat de methode hier een andere invulling krijgt wordt dat in de header aangekondigd.

In de header van de methode `turf` staat niet de zichtbaarheid `private`, maar `protected`. Dit maakt het mogelijk om deze methode ook in eventuele toekomstige subklassen van `TurfTab` te gebruiken. Bovendien staat er de modifier `virtual`. Dit maakt het mogelijk om deze methode ook in eventuele toekomstige subklasse ook een andere invulling te geven.

Zelf properties definiëren

Behalve public methoden heeft de klasse ook drie public properties. Dit is voor het eerst dat we zo’n property zelf definiëren in een klasse, dus let goed op de nieuwe syntax die hiervoor nodig is. De eerste regel van de property lijkt op de delaratie van een member-variabele:

```
public int Totaal
```

maar het wordt niet afgesloten met een puntkomma, want er volgt nog een body tussen accolades. Tussen de accolades kunnen een soort mini-methodes met de namen `get` en/of `set` worden gedefinieerd. In dit geval kiezen we voor alleen de mini-methode `get`, want we willen er een read-only attribuut van maken:

```
{
    get
    {   return totaal;
    }
}
```

De `get` mini-methode doet niets anders dan de private variabele `total` opleveren. Op deze manier heeft de gebruiker wel beschikking over het `Totaal` als hij dat wil raadplegen, maar hij kan het totaal niet veranderen; dat kan alleen met properties waarvoor ook een `set` mini-methode beschikbaar is.

Om te laten zien dat een property niet altijd 1-op-1 correspondeert met een private variabelen, maken we ook een property `Gemiddelde`:

```
public float Gemiddelde
{
    get
    {   return (float)totaal / tellers.Length;
    }
```

```
    }
}
```

Voor degene die de property **Gemiddelde** gebruikt mag het lijken of er een variabele wordt opgehaald, maar eigenlijk wordt de waarde snel even berekend uit twee andere waarden. Deze property wordt in het programma overigens niet gebruikt, maar wie weet komt hij later nog eens van pas. Een derde property geeft toegang tot de array met turf-waardes. Ook deze property heeft alleen een mini-methode **get**. Maar daarmee is onze private array nog niet veilig, want een array is in feite een verwijzing naar het eigenlijke object. Degene die de property opvraagt zou dus ook de waarden van de array kunnen veranderen, en dat is iets wat we liever niet willen hebben omdat het de integriteit van de turftabel zou aantasten. Daarom geven we niet de array zelf terug, maar een met **Clone** gemaakte kopie. In die kopie mag de gebruiker naar hartelust waarden veranderen, het origineel heeft daar geen last van.

De methode **Clone** wordt geërfd van **object**; het resultaat ervan is zonder verdere maatregelen ook weer **object**, maar omdat wij weten dat het een array is die we gekloond hebben, kunnen we het resultaat casten naar type **int[]**. Dat moet wel, want de **get** mini-methode wordt geacht een waarde te returnen zoals die in de header van de property wordt gedeclareerd.

De klasse **StaafDiagram**

De klasse **StaafDiagram** is een **UserControl** die een staafdiagram tekent. Daartoe is de **Paint**-eventhandler ingevuld met methode **teken**. Deze methode tekent de eigenlijke rechthoekjes. De af te beelden waardes moeten klaarstaan in een private array **waardes**.

De beschikbare hoogte wordt netjes verdeeld over het aantal benodigde balkjes, en de breedte zodanig dat het grootste balkje altijd precies past (tenzij de waarden < 10 zijn, want dan is het overdreven om er zo'n brede balk voor te gebruiken). Daartoe hebben we de grootste waarde van de array nodig, en het komt dus goed uit dat we daar een methode voor hebben klaarliggen in de eerder geschreven **ArrayBieb**. Merk op dat deze klasse gebruikt mag worden omdat de betreffende namespace (**CirkelKlikker**) in een **using**-directief wordt vermeld.

Om de array te kunnen veranderen stellen we een property beschikbaar waarmee de buitenwereld de waarde kan veranderen. In deze property is dus juist een mini-methode **set** nodig:

```
public int[] Waardes
{
    set
    {
        this.waardes = value;
        this.Invalidate();
    }
}
```

Als de mini-methode **set** aanwezig is, mag de property in een toekenningsopdracht worden gebruikt, precies wat we doen in op regel 35 van klasse **LetterTeller** (listing 26).

blz. 132

In de body van de mini-methode **set** mag je gebruikmaken van een speciale waarde **value**; de semantiek ervan is de nieuwe waarde die de property aan het krijgen is. Het woord **value** is dus een vast woord met een vaste betekenis (net zoals **this**), met als verschil dat **value** alleen in de body van een **set** mini-methode zinvol is. Daarbuiten is het woord niet gereserveerd en mag het ook als variabelenaam o.i.d. worden gebruikt.

Hier is goed te zien wat het voordeel van een property is boven een public membervariabele: aanpassen van de property heeft automatisch tot gevolg dat ook de methode **Invalidate** wordt aangeroepen (die via het event-mechanisme weer een aanroep van **teken** forceert. Zo blijft het plaatje dus up-to-date met de toegekende property, terwijl de collega-programmeur denkt dat hij 'alleen maar' de waarde van de property wijzigt.

Deze zelfde aanpak wordt intern ook gehanteerd in de klasse **Label**. Je dacht misschien dat met het wijzigen van de property **Text** van een label alleen maar een variabele wordt veranderd, maar wat er verder allemaal nog gebeurt zien we nu ook eens van de andere kant.

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace LetterTeller
{
    public class LetterTeller : Form
    {
        private TextBox invoer;
        10 private Label uitvoer;
        private StaafDiagram diagram;

        public LetterTeller()
        {
            15 this.Text = "LetterTeller";
            this.ClientSize = new Size(320, 481);
            invoer = new TextBox(); invoer.Multiline = true;
            uitvoer = new Label();
            diagram = new StaafDiagram();
            20 invoer.Location = new Point(10, 10); invoer.Size = new Size(300, 100);
            uitvoer.Location = new Point(10, 120); uitvoer.Size = new Size( 70, 351);
            diagram.Location = new Point(90, 120); diagram.Size = new Size(210, 338);
            this.Controls.Add(invoer);
            this.Controls.Add(uitvoer);
            25 this.Controls.Add(diagram);

            invoer.TextChanged += berekenAntwoord;
        }

        30 void berekenAntwoord(object o, EventArgs ea)
        {
            TurfTab tabel = new TurfTab();
            tabel.Turf(invoer.Text);
            uitvoer.Text = tabel.ToString();
            35 diagram.Waardes = tabel.Waardes;
        }
    }
}
```

Listing 26: LetterTeller/LetterTeller.cs

```
namespace LetterTeller
{
    public class TurfTab
    {
        5      protected int[] tellers;
        protected int totaal;

        public TurfTab()
        {
            10      tellers = new int[26];
        }
        protected virtual void turf(char ch)
        {
            if (ch>='A' && ch<='Z')
            15      {    tellers[ch-'A']++;
                        totaal++;
                    }
            if (ch>='a' && ch<='z')
            {    tellers[ch-'a']++;
            20      totaal++;
                }
        }
        public void Turf(string s)
        {
            25      for (int t = 0; t < s.Length; t++)
                        this.turf(s[t]);
        }
        public override string ToString()
        {
            30      string res = "";
            for (int t = 0; t < 26; t++)
                res += (char)(t + 'A') + ": " + tellers[t] + " keer\n";
            res += "totaal: " + totaal;
            return res;
        }
        35      public int Totaal
        {
            get
            {    return totaal;
            40      }
        }
        public float Gemiddelde
        {
            get
            45      {    return (float)totaal / tellers.Length;
                }
        }
        public int[] Waardes
        {    get
        50      {    return (int[]) tellers.Clone();
                }
        }
    }
}
```

Listing 27: LetterTeller/TurfTab.cs

```
using System.Drawing;
using System.Windows.Forms;
using CirkelKlikker;

5 namespace LetterTeller
{
    public partial class StaafDiagram : UserControl
    {
        private int[] waardes;

10     public StaafDiagram()
        {
            this.Paint += teken;
        }

15     public int[] Waardes
        {
            set
            {
                this.waardes = value;
                this.Invalidate();
20            }
        }

        private void teken(object o, PaintEventArgs pea)
        {
25            Graphics gr = pea.Graphics;
            if (waardes != null)
            {
                int max = ArrayBieb.Grootste(waardes);
                if (max < 10)
30                    max = 10;
                float balkH = (float)this.Height / waardes.Length;
                float balkUnit = (float)this.Width / max;

                for (int t = 0; t < waardes.Length; t++)
35                    gr.FillRectangle(Brushes.Blue, 0, t * balkH, balkUnit * waardes[t], balkH-1);
            }
        }
    }
}
```

Listing 28: LetterTeller/StaafDiagram.cs

Hoofdstuk 10

Libraries / taalconcepten / toepassingen

10.1 Menu's / lambda-expressies / Bitmap-editor

Bitmaps

Een *bitmap* is een tekening die uit kleine puntjes is opgebouwd. Bitmaps worden veel gebruikt, omdat beeldschermen (bijna) altijd uit losse beeldpunten bestaan, waarop een bitmap goed kan worden weergegeven. De naam 'bitmap' geeft aan dat elk punt door een 'bit' (0 of 1), of bool-waarde (false of true) kan worden afgebeeld. Een '0' of 'false' duidt een wit punt aan, een '1' of 'true' een zwart punt.

Eigenlijk is het een woord uit het zwartwit-tijdperk, want plaatjes zijn tegenwoordig natuurlijk over het algemeen in kleur. De correcte term voor een kleurenplaatje dat uit losse punten is opgebouwd is eigenlijk 'pixmap': een plaatje dat uit 'pixels' oftewel beeldpunten is opgebouwd. In de forms-library is een klasse `Bitmap` gedefinieerd die wel degelijk een kleurenplaatje bevat. Maar wij gaan zelf een klasse `Bitmap` schrijven, die een simpel zwartwit-plaatje bevat.

Functies van de bitmap-editor

We gaan in deze sectie een bitmap-editor ontwikkelen. Het is *geen* pixmap-editor: met dit programma kun je alleen zwartwit-plaatjes maken. Of preciezer gezegd: *monochrome* plaatjes, want de kleuren hoeven niet per se zwart en wit te zijn. In feite gaan we in de GUI rood en wit gebruiken. Het programma toont een vergrote weergave van een bitmap, die de gebruiker met de muis kan veranderen: met de linker muisknop worden punten ingekleurd, met de rechter muisknop weer blanco.

Bovenaan het window bevindt zich een menu waarmee de gebruiker een aantal operaties op het plaatje kan uitvoeren. Deze operaties zijn:

- schoonmaken van het hele plaatje
- inverteren van het plaatje (wit wordt zwart en zwart wordt wit)
- het plaatje één beeldpunt opschuiven naar links, rechts, boven of beneden
- het plaatje 'massiever' maken, of juist 'hol'
- het zogenaamde 'game of life' spelen, waarbij een 'nieuwe generatie' wordt berekend, of een continue animatie wordt getoond

Opdeling in klassen

We verdelen het programma over een aantal verschillende klassen:

- een klasse `Program`, waarin alleen een methode `Main` staat die een window aanmaakt en runt. Dit hebben we nu al zo vaak gezien dat de listing niet in dit boek is opgenomen.
- het hoofdscherm wordt zoals altijd een subklasse van `Form`, die we `Hoofdscherm` zullen noemen (listing 33 en verder).
- De belangrijkste interface-component die in het programma gebruikt worden is de weergave van de tekening. De tekening is een subklasse van `UserControl`, waarin de `Paint`-eventhandler is ingevuld. Onze subklasse van `UserControl` zullen we `BitmapControl` noemen (zie listing 29 en listing 30).
- De eigenlijke bitmap gaan we modelleren in een aparte klasse `Bitmap`, zonder daarbij de precieze visuele representatie vast te leggen (zie listing 31 en listing 32).

blz. 142

blz. 138

blz. 139

blz. 140

blz. 141

Methoden van Hoofdscherm

In de klasse `Hoofdscherm` wordt de grafische userinterface (GUI) gemodelleerd. Het belangrijkste wat daar gebeurt is het creëren en zichtbaar maken van het menu en één `BitmapControl`-object.

Alle items uit de menu's zijn gekoppeld aan event-handlers. De meeste items worden door dezelfde event-handler afgehandeld: **uitvoeren**, die zich in de klasse **BitmapControl** bevindt. Alleen de items 'start' en 'stop' voor het starten en stoppen van de animatie hebben een aparte handler, en het item 'close' heeft een aparte handler die zich in de klasse **Hoofdscherm** bevindt.

Muiskliks worden hier niet afgehandeld: dat doet de **BitmapControl**, die deel uitmaakt van de GUI, zelf.

Methoden van **BitmapControl**

In de klasse **BitmapControl** wordt de grafische afbeelding van de tekening gemodelleerd. De tekening zelf wordt in in deze klasse niet gemodelleerd, uitsluitend het weergeven ervan op het scherm. Daartoe wordt het **Paint**-event afgehandeld. Hierin wordt een lijnenstelsel getekend, waarmee de individuele beeldpunten afgebakend worden. Verder worden de gekleurde beeldpunten als vierkantje ingetekend. Hiertoe wordt aan een **Bitmap**-object gevraagd welke punten gekleurd zijn.

Het indrukken van de muis wordt afgevangen door een event-handler voor **MouseClicked** en **MouseMove**. Als reactie op het indrukken van de muis, of het "dargen" (bewegen met ingedrukte muisknop) wordt het betreffende punt in het **Bitmap**-object gekleurd (of juist blanco) gemaakt, en het hele plaatje opnieuw getekend. Bij het tekenen wordt zoveel mogelijk het hele scherm gevuld. Om te bepalen hoe groot de beeldpunten kunnen worden afgebeeld, wordt de beschikbare ruimte gedeeld door het aantal beeldpunten. Om de beeldpunten vierkant te houden, wordt het minimum van de uitkomsten van deze delingen voor de breedte en de hoogte gebruikt.

Methoden van **Bitmap**

In een object van de klasse **Bitmap** worden de bits van het plaatje opgeslagen. Er komen methoden **vraagKleur** en **veranderKleur** om de status van een beeldpunt op te vragen, respectievelijk te veranderen. (Die kunnen vanuit **BitmapControl** worden aangeroepen als de **bitmap** getekend moet worden, of met de muis veranderd wordt).

Daarnaast komen er de volgende methoden in de klasse **Bitmap**:

- Twee constructor-methoden: de ene maakt een blanco **bitmap** met gegeven breedte en hoogte, de andere maakt een kopie van een bestaande **bitmap**.
- Zes methoden die het hele plaatje tegelijk veranderen: **Clear** om te wissen, **Invert** om het te inverteren, en **Left**, **Right**, **Up** en **Down** om het beeld te verschuiven. Deze operaties werken op de **bitmap** die onderhanden is; na aanroep van deze methoden is de **bitmap** dus veranderd.
- Drie methoden die het plaatje veranderen volgens bepaalde regels: **Bold** maakt het plaatje massiever, **Outline** bepaalt de omtrek van het plaatje, en **Life** bepaalt een nieuwe generatie volgens de regels van het 'game of life'.
- Een hulpmethode die van pas komt om **Clear**, **Bold** en **Outline** te implementeren: een methode **combineer** die de punten van de **bitmap** combineert met beeldpunten van een andere **bitmap**, volgens nog nader te bepalen regels, en een hulpmethode **buren** die gebruikt wordt in **Life**.

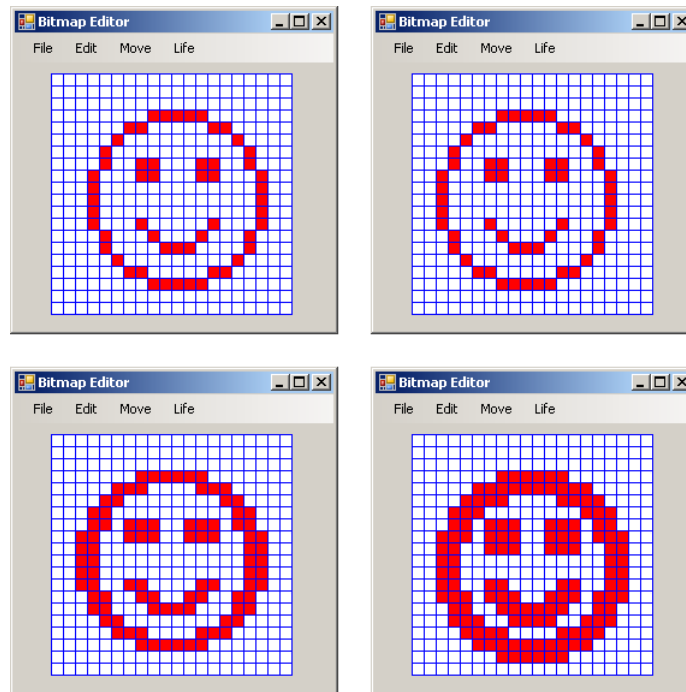
Beeldbewerkingsoperaties

In de klasse **Bitmap** komt een methode **Bold** die een plaatje massiever maakt. Een makkelijke manier om dat te doen is de volgende. We maken eerst een kopie van het plaatje. De kopie schuiven we één beeldpunt naar links. Daarna kleuren we de beeldpunten die in het oorspronkelijke plaatje of in het verschoven plaatje zwart zijn. Het plaatje krijgt er daardoor een extra rand bij. Vervolgens herhalen we het procedé in verticale richting.

Om de **Outline** van een plaatje te bepalen gaan we als volgt te werk. We maken weer een kopie, en verschuiven die één beeldpunt naar links en naar beneden. Vervolgens laten we alleen die beeldpunten zwart, die in het oorspronkelijke plaatje zwart zijn, maar in het verschoven plaatje wit; die liggen immers aan de rand. Ook de beeldpunten die wit zijn, maar in het verschoven plaatje zwart liggen aan de rand, en moeten dus zwart gemaakt worden.

Het 'Game of Life'

Nee, dit is geen actiespelletje met high score en verborgen extra levels. Het 'Game of Life' is een simulatieproces, in de jaren zestig van de vorige eeuw bedacht door John Conway. Het bijzondere ervan is dat door hele simpele regels toch complexe patronen kunnen ontstaan. Het wordt daarom soms gebruikt als metafoor voor biologische processen: op grond van eenvoudige biochemische processen kunnen ingewikkelde organismen, en zelfs intelligentie en bewustzijn ontstaan.



Figuur 29: De Bitmap Editor in werking: vier stappen in het ‘bold’ maken van een plaatje

De wereld in het Game of Life bestaat uit een twee-dimensionaal raster, waarin elke cel bewoond kan worden door een beestje. Je kunt deze wereld dus weergeven met een bitmap: een zwart vakje duidt op de aanwezigheid van een beestje, een wit vakje is onbewoond.

Het bewoningspatroon in de volgende ‘generatie’ wordt bepaald door de acht buur-vakjes van elk vakje:

- Een vakje *blijft bewoond* als *twee of drie buurvakjes* bewoond zijn (bij minder burens sterft het beestje van eenzaamheid, en bij meer burens door verstikking).
- In een leeg vakje wordt een *nieuw beestje geboren* als er *precies drie* buurvakjes bewoond zijn.

De vakjes aan de randen hebben minder buurvakjes dan de rest. Om deze vakjes ook eerlijk te behandelen, doen we alsof de linker- en de rechterrands aan elkaar grenzen, en de onder- en bovenrand ook. In figuur 30 is een aantal opeenvolgende generaties uit het Game of Life getekend. Opvallend is dat bepaalde clusters van beestjes stabiel zijn, of zich zelfs over de wereld kunnen verplaatsen! De individuele beestjes sterven en worden geboren, maar een soort ‘hogere organismen’ blijven leven en kunnen bewegen...

Implementatie van Bitmap met een array

Alle genoemde beeldbewerkingsmethoden kunnen de individuele beeldpunten van het plaatje aanspreken door de methoden `vraagKleur` en `veranderKleur` aan te roepen. Maar om die twee methoden te schrijven, zullen we een beslissing moeten nemen over de variabelen waarmee een bitmap-object wordt gemodelleerd.

De meest voor de hand liggende implementatie is een array van `bool`-waarden: voor elk beeldpunt een `bool`. Twee extra integers bewaren de breedte en hoogte van het plaatje. De array wordt geïnitieerd in de constructor, en methoden `vraagKleur` en `veranderKleur` kiezen het juiste element er uit:

```

using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;
5
namespace BitmapEditor
{
    public partial class BitmapControl : UserControl
    {
10        private Bitmap model;

        public BitmapControl()
        {
            model = new Bitmap(20, 20);
15            this.Paint += teken;
            this.Resize += vergroot;
            this.MouseClick += klik;
            this.MouseMove += beweeg;
        }

20        public int Diameter
        {
            get
            {
                Size s = this.ClientSize;
                return Math.Min(s.Width / model.Breedte, s.Height / model.Hoogte);
            }
25        }

        private void teken(object sender, PaintEventArgs e)
        {
            int w = model.Breedte;
            int h = model.Hoogte;
30            int d = this.Diameter;
            for (int y = 0; y <= h; y++)
                e.Graphics.DrawLine(Pens.Blue, 0, y * d, w * d, y * d);
            for (int x = 0; x <= w; x++)
                e.Graphics.DrawLine(Pens.Blue, x * d, 0, x * d, h * d);
35            for (int y = 0; y < h; y++)
            {
                for (int x = 0; x < w; x++)
                {
40                    Brush b;
                    if (model.vraagKleur(x, y))
                        b = Brushes.Red;
                    else b = Brushes.White;
                    e.Graphics.FillRectangle(b, x * d + 1, y * d + 1, d - 1, d - 1);
                }
45            }

            int rx = w * d + 1;
            int ry = h * d + 1;

            Brush bg = new SolidBrush(this.BackColor);
50            e.Graphics.FillRectangle(bg, rx, 0, this.Width - rx, this.Height);
            e.Graphics.FillRectangle(bg, 0, ry, this.Width, this.Height - ry);
        }

        override protected void OnPaintBackground(PaintEventArgs e)
        {
55        }

        private void vergroot(object sender, EventArgs e)
        {
            this.Invalidate();
        }
    }
}

```

```
private void klik(object sender, MouseEventArgs mea)
60 {
    int d = this.Diameter;
    int x = mea.X / d;
    int y = mea.Y / d;
    if (x >= 0 && x < model.Breedte && y >= 0 && y < model.Hoogte)
65     model.veranderKleur(x, y, mea.Button == MouseButton.Left);
    this.Invalidate();
}
private void beweeg(object sender, MouseEventArgs mea)
{
70     if (mea.Button == MouseButton.Left || mea.Button == MouseButton.Right)
        this.klik(sender, mea);
}
public void uitvoeren(object sender, EventArgs e)
{
75     switch (sender.ToString())
    {
        case "Clear":    this.model.Clear();    break;
        case "Invert":   this.model.Invert();   break;
        case "Bold":     this.model.Bold();     break;
80     case "Outline":    this.model.Outline();   break;
        case "Left":     this.model.Left();     break;
        case "Right":    this.model.Right();    break;
        case "Up":       this.model.Up();       break;
        case "Down":     this.model.Down();     break;
85     case "Step":       this.model.Life();     break;
    }
    this.Invalidate();
}

90 private Thread animatie;

public void starten(object sender, EventArgs e)
{
    animatie = new Thread(animatieFunctie);
95     animatie.Start();
}
public void stoppen(object sender, EventArgs e)
{
    animatie = null;
100 }
private void animatieFunctie()
{
    while (animatie != null)
    {
105         this.model.Life();
        this.Invalidate();
        Thread.Sleep(50);
    }
}
110 }
}
```

Listing 30: BitmapEditor/BitmapControl.cs, deel 2 van 2

```

using System;

namespace BitmapEditor
{
5   public class Bitmap
    {
        private bool[,] vakjes;

        public Bitmap(int br, int h)
10        {   vakjes = new bool[br, h];
        }
        public Bitmap(Bitmap ander)
        {   vakjes = new bool[ander.Breedte, ander.Hoogte];
            this.Kopieer(ander);
15        }
        public int Breedte
        {   get { return vakjes.GetLength(0); }
        }
        public int Hoogte
20        {   get { return vakjes.GetLength(1); }
        }
        public void veranderKleur(int x, int y, bool b)
        {   vakjes[x, y] = b;
        }
25        public bool vraagKleur(int x, int y)
        {   return vakjes[x, y];
        }
        private void combineer(Bitmap ander, Func<bool,bool,bool> comb)
        {   for (int x = 0; x < this.Breedte; x++)
30            for (int y = 0; y < this.Hoogte; y++)
                this.veranderKleur(x, y, comb(this.vraagKleur(x, y) , ander.vraagKleur(x, y)));
        }
        private void Kopieer(Bitmap ander)
        {   this.combineer(ander, (a, b) => b);
35        }
        public void Clear()
        {   this.combineer(this, (a, b) => false);
        }
        public void Invert()
40        {   this.combineer(this, (a, b) => !a);
        }
        public void Bold()
        {   Bitmap ander = new Bitmap(this);
            ander.Left();
            this.combineer(ander, (a, b) => a || b);
45            ander.Kopieer(this);
            ander.Down();
            this.combineer(ander, (a, b) => a || b);
        }
50        public void Outline()
        {   Bitmap ander = new Bitmap(this);
            ander.Left();
            ander.Down();
            this.combineer(ander, (a, b) => a != b);
55        }
    }

```

```

public void Left()
{
    for (int y = 0; y < this.Hoogte; y++)
    {
        for (int x = 1; x < this.Breedte; x++)
            this.veranderKleur(x - 1, y, this.vraagKleur(x, y));
60         this.veranderKleur(this.Breedte - 1, y, false);
    }
}

public void Right()
{
    for (int y = 0; y < this.Hoogte; y++)
65     {
        for (int x = this.Breedte-1; x >0; x--)
            this.veranderKleur(x, y, this.vraagKleur(x-1, y));
        this.veranderKleur(0, y, false);
    }
}

70 public void Up()
{
    for (int x = 0; x < this.Breedte; x++)
    {
        for (int y = 1; y < this.Hoogte; y++)
            this.veranderKleur(x, y-1, this.vraagKleur(x, y));
        this.veranderKleur(x, this.Hoogte - 1, false);
75     }
}

public void Down()
{
    for (int x = 0; x < this.Breedte; x++)
    {
        for (int y = this.Hoogte-1; y >0; y--)
80         this.veranderKleur(x, y, this.vraagKleur(x, y-1));
        this.veranderKleur(x, 0, false);
    }
}

public void Life()
85 {
    Bitmap old = new Bitmap(this);
    for (int y=0; y<Hoogte; y++)
        for (int x = 0; x < Breedte; x++)
        {
            int n = old.buren(x,y);
            this.veranderKleur(x, y, n == 3 || (old.vraagKleur(x, y) && n == 2));
90        }
}

private int buren(int x, int y)
{
    int x0 = x - 1; if (x0 < 0) x0 += Breedte;
    int y0 = y - 1; if (y0 < 0) y0 += Hoogte;
95    int x1 = x + 1; if (x1 >= Breedte) x1 -= Breedte;
    int y1 = y + 1; if (y1 >= Hoogte ) y1 -= Hoogte;
    int n = 0;
    if (this.vraagKleur(x0,y0)) n++;
    if (this.vraagKleur(x ,y0)) n++;
100    if (this.vraagKleur(x1,y0)) n++;
    if (this.vraagKleur(x0,y )) n++;
    if (this.vraagKleur(x1,y )) n++;
    if (this.vraagKleur(x0,y1)) n++;
    if (this.vraagKleur(x ,y1)) n++;
105    if (this.vraagKleur(x1,y1)) n++;
    return n;
}
}
}

```

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace BitmapEditor
{
    public class Hoofdscherm : Form
    {
        MenuStrip menuStrip;
10        BitmapControl viewer;

        public Hoofdscherm()
        {
            this.Text = "Bitmap Editor";
            this.Resize += this.vergroten;
15            viewer = new BitmapControl();
            menuStrip = new MenuStrip();

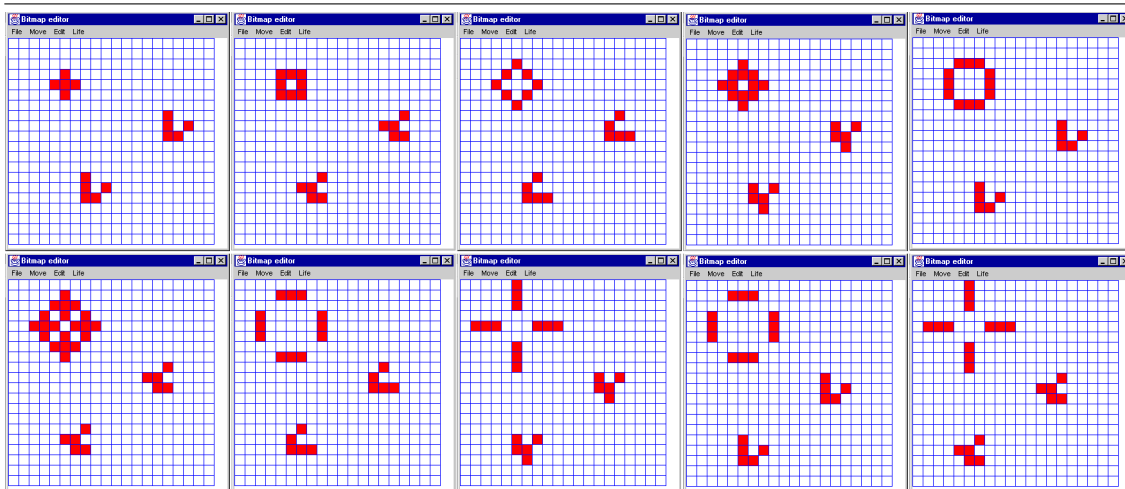
            ToolStripDropDownItem menu;
            menu = new ToolStripDropDownButton("File");
20            menu.DropDownItems.Add("Close", null, this.afsluiten);
            menuStrip.Items.Add(menu);
            menu = new ToolStripMenuItem("Move");
            menu.DropDownItems.Add("Left", null, viewer.uitvoeren);
            menu.DropDownItems.Add("Right", null, viewer.uitvoeren);
25            menu.DropDownItems.Add("Up", null, viewer.uitvoeren);
            menu.DropDownItems.Add("Down", null, viewer.uitvoeren);
            menuStrip.Items.Add(menu);
            menu = new ToolStripMenuItem("Edit");
            menu.DropDownItems.Add("Clear", null, viewer.uitvoeren);
30            menu.DropDownItems.Add("Invert", null, viewer.uitvoeren);
            menu.DropDownItems.Add("Bold", null, viewer.uitvoeren);
            menu.DropDownItems.Add("Outline", null, viewer.uitvoeren);
            menuStrip.Items.Add(menu);
            menu = new ToolStripMenuItem("Life");
35            menu.DropDownItems.Add("Step", null, viewer.uitvoeren);
            menu.DropDownItems.Add("Start", null, viewer.starten);
            menu.DropDownItems.Add("Stop", null, viewer.stoppen);
            menuStrip.Items.Add(menu);

40            this.Controls.Add(menuStrip);
            this.Controls.Add(viewer);
            this.vergroten(null, null);
        }

45        private void vergroten(object sender, EventArgs e)
        {
            int w = this.ClientSize.Width - 20;
            int h = this.ClientSize.Height - 50;
            viewer.Size = new Size(w, h);
            viewer.Location = new Point(10, 40);
50        }

        private void afsluiten(object sender, EventArgs e)
        {
            this.Close();
        }

55    }
}
```



Figuur 30: Tien stappen tijdens het 'Game of Life'

```
class Bitmap
{
    int breed, hoog;
    bool [] punten;
    Bitmap(int b, int h)
    {
        breed=b; hoog=h;
        punten = new bool[breed*hoog];
    }
    void veranderKleur(int x, int y, bool b)
    {
        punten[y*breed+x] = b;
    }
    bool vraagKleur(int x, int y)
    {
        return punten[y*breed+x];
    }
}
```

Als een bitmap bijvoorbeeld 10 beeldpunten breed is, worden array-elementen 0 t/m 9 gebruikt voor de eerste rij, 10 t/m 19 voor de tweede rij, enzovoort. Daarom wordt in de methoden `vraagKleur` en `veranderKleur` het nummer van de rij, `y`, vermenigvuldigd met de breedte van de bitmap.

Implementatie van Bitmap met een tweedimensionale array

Bij het modelleren van tweedimensionale structuren, zoals plaatjes, is het eigenlijk gemakkelijker om een tweedimensionale array te gebruiken. Berekeningen zoals `y*breed+x` zijn dan niet meer nodig: je kunt de twee coördinaten in een tweedimensionale array apart specificeren.

Tweedimensionale arrays worden op dezelfde manier gebruikt als eendimensionale arrays, alleen staan er nu bij initialisatie en gebruik twee getallen tussen de vierkante haken achter de naam in plaats van één, gescheiden door de komma. Bij de declaratie staan er geen getallen, maar wel de komma. De implementatie van Bitmap komt er dan als volgt uit te zien:

```
class Bitmap
{
    int breed, hoog;
    bool [,] punten;
    Bitmap(int b, int h)
    {
        breed=b; hoog=h;
        punten = new bool[breed,hoog];
    }
    void veranderKleur(int x, int y, bool b)
    {
        punten[x,y] = b;
    }
    bool vraagKleur(int x, int y)
    {
        return punten[x,y];
    }
}
```

In feite is het zelfs niet nodig om de breedte en hoogte apart op te slaan, want een array kent een methode `GetLength`, waarmee de lengte van de array in elke dimensie kan worden opgevraagd. Daarmee kunnen we properties `Breedte` en `Hoogte` maken:

```
public int Breedte
{
    get { return vakjes.GetLength(0); }
}
public int Hoogte
{
    get { return vakjes.GetLength(1); }
}
```

10.2 Panels / switch&break / Calculator

Beschrijving van de casus

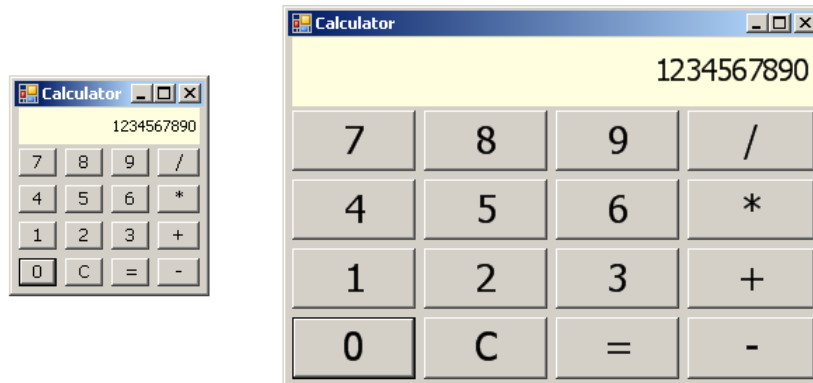
We gaan een eenvoudige 4-functie calculator maken. Voorlopig werkt de calculator alleen met integer getallen (maar dat kan eenvoudig aangepast worden). De gebruiker kan de calculator bedienen met buttons op het scherm, zoals in figuur 31. Bijzonder aan dit programma is dat de GUI meeschaalt met het window. Als de gebruiker het window groter maakt, blijft de calculator het hele scherm vullen: de knoppen en de ‘display’ worden groter, en krijgen ook grotere letters. De opbouw van het programma demonstreert verder hoe je de vorm en de inhoud kunt scheiden, in aparte klassen:

- een klasse `Calculator` waar de userinterface wordt opgebouwd, en het indrukken van de knoppen wordt afgehandeld (zie listing 34 en listing 35);
- een klasse `Processor` waar al het inhoudelijke werk gebeurt, maar die juist niets met de userinterface te maken wil hebben (zie listing 36);
- een klasse `Program` waarin alleen de een-regelige methode `Main` staat.

blz. 147

blz. 148

blz. 149



Figuur 31: Het programma Calculator in werking, voor en na resizen van het window.

De processor van de rekenmachine

Voor het schrijven van de klasse `Proc` moeten we ons eerst afvragen welke variabele er in deze klasse nodig zijn. Met andere woorden: in welke ‘toestand’ de processor zich kan bevinden. Om daar achter te komen bekijken we in detail wat er gebeurt als de gebruiker de rekenmachine bedient.

De gebruiker begint met een getal in te toetsen. Alle ingedrukte cijfertoeetsen worden geaccumuleerd in een steeds groter wordend getal. Is het getal 12 al ingevoerd, en drukt de gebruiker op de 3-toets, dan wordt de nieuwe waarde 123; dat is 10 maal de oude waarde plus het ingedrukte cijfer.

Daarna drukt de gebruiker een operator-toets in. Op het scherm verandert ogenschijnlijk niet (er blijft bijvoorbeeld 123 staan), maar de ‘huidige waarde’ is blijkbaar anders geworden. Na het indrukken van de 4-toets verschijnt namelijk niet 1234, maar 4 op het scherm.

De gebruiker gaat door met het intoetsen van het tweede getal, bijvoorbeeld 456, en drukt tenslotte op de =-toets. Op dat moment is de oude waarde (123) ineens weer van belang, want die moet worden opgeteld/vermenigvuldigd/enz. bij de nieuwe waarde.

Bovendien is de eerder gekozen operator nodig, want na het indrukken van de `=`-toets moet de machine weten of er opgeteld of vermenigvuldigd moet worden.

Voor de toestand van de rekenmachine zijn dus van belang:

- de ‘huidige’ waarde
- de ‘vorige’ waarde
- de tekst op het scherm (niet altijd gelijk aan de ‘huidige’ waarde!)
- de laatst gekozen operator

De klasse **Processor** krijgt daarom vier membervariabelen: twee getallen, een string, en een **char**. Voor de getallen gebruiken we long-waarden in plaats van int-waarden, zodat de calculator getallen tot 18 cijfers aankan.

Verder maken we methoden die corresponderen met de diverse acties die op de calculator kunnen worden toegepast:

- De methode **Cijfer** accumuleert een cijfer bij het huidige getal.
- De methode **Reken** voert de berekening volgens de bewaarde operator uit op het vorige getal en de huidige waarde. Het resultaat komt in de variabele `scherm`, en wordt ook de nieuwe “vorige” waarde, zodat het resultaat meteen als linker operand van de volgende operatie gebruikt kan worden.
- De methode **Operatie** correspondeert met het indrukken van een operator-toets. Eerst wordt de eventueel nog hangende berekening uitgevoerd; vervolgens wordt de nieuwe operator opgeslagen.
- De methode **Schoon** initialiseert de toestand. Door het toekennen van `+` aan de variabele die de operator bevat, kan de `=`-toets ook veilig ingedrukt worden als er alleen maar cijfers zijn ingevoerd. De **Schoon**-methode correspondeert met de `C`-toets, maar wordt ook in de constructormethode aangeroepen.

De buitenwereld mag alleen via de methoden de toestand aanpassen; de member-variabelen zijn `private`, en kunnen vanuit de GUI niet direct veranderd worden. Wel definiëren we een `public` property om de scherm-waarde te kunnen opvragen: die moet de GUI immers in beeld brengen.

De userinterface van de rekenmachine

In de constructormethode **Calculator** bouwen we de userinterface op. In een `for`-opdracht worden 16 buttons gecreëerd.

Met een index tussen vierkante haken achter een constante string (!) kiezen we voor elke button het juiste opschrift uit een string met alle opschriften. De buttons worden met **Add** toegevoegd: niet aan de controls van de hele form, maar aan de controls van een voor dat doel aangemaakt **TableLayoutPanel**-object. Daardoor komen de knoppen netjes in een raster met 4 kolommen te staan. Ook het Label waarin het resultaat getoond wordt komt in dat paneel te staan, waarbij we aan het paneel vragen om dit specifieke control meerdere kolommen te laten beslaan. Het paneel op zijn beurt wordt aan de controls van het hele form toegevoegd.

Alle buttons krijgen dezelfde **Click**-event-handler. In die methode **klik** kunnen we er via de **object**-parameter achter komen welke button verantwoordelijk was voor de klik.

We kunnen in dit geval dat object *niet* met `==` vergelijken met alle aangemaakte buttons, omdat we de verwijzingen naar de 16 aparte buttons niet hebben bewaard in een array-variabele. Bij het aanmaken van de buttons werd die alleen tijdelijk eventjes bewaard in de lokale variabele **knop**, maar die is in de event-handler niet meer zichtbaar. Maar er is een andere uitweg. Omdat we zeker weten dat de aanroep van **klik** werd veroorzaakt door een **Button**, kunnen we de **object**-variabele met een *cast* converteren naar het type **Button**. Van dat button-object kunnen we vervolgens de **Text**-property opvragen, en daarvan weer het eerste karakter: dat is precies het opschrift van de button. De verdere verwerking vindt plaats in de methode **verwerk**: afhankelijk van dit opschrift wordt één van de methodes van **Proc** aangeroepen. Het resultaat wordt vervolgens op de ‘display’ van de calculator getoond.

De switch-opdracht

Bij het onderscheiden van de mogelijke knop-opschriften wordt een nieuw soort opdracht gebruikt: de **switch**-opdracht.

De switch-opdracht kan gebruikt worden als vereenvoudiging van een veel voorkomend soort if-opdracht. Vaak wordt een if-opdracht gebruikt om een variabele met een aantal alternatieven te vergelijken:

```
if      (x==1)      een();
else if (x==2)      { twee(); ookTwee(); }
else if (x==3 || x==4) drieOfVier();
else          meer();
```

Met een switch-opdracht kan dit worden geschreven als:

```
switch(x)
{
  case 1:  een();
           break;
  case 2:  twee();
           ookTwee();
           break;
  case 3:
  case 4:  drieOfVier();
           break;
  default: meer();
           break;
}
```

Bij uitvoering van een switch-opdracht wordt de waarde tussen de haakjes uitgerekend. Vervolgens wordt de verwerking van opdrachten voortgezet achter het woord **case** met de betreffende waarde. Is dit er niet, dan worden de opdracht(en) achter **default** uitgevoerd. De waarden achter de diverse **cases** moeten constanten zijn (getallen, characters of strings tussen aanhalingstekens, of als **const** gedeclareerde variabelen).

De break-opdracht

Als we niet uitkijken, worden in een switch-opdracht niet alleen de opdrachten achter betreffende **case** uitgevoerd, maar ook de opdrachten achter de volgende cases. Alleen de plaatsing van de speciale **break**-opdracht verhindert dit.

De betekenis van de **break**-opdracht is: ‘stop de verwerking van de body van de switch-, while- of for-opdracht waarmee je bezig bent’. Zonder de plaatsing van deze opdrachten zou in bovenstaand voorbeeld in het geval dat **x** de waarde 2 heeft niet alleen methode **twee** worden aangeroepen, maar ook **drieOfVier** en **meer**.

In bijzondere gevallen zou je daar misschien gebruik van willen maken, dus dat de verschillende cases als het ware in elkaar overlopen. In de met C# verwante talen als C, C++ en Java kan dat inderdaad. Maar dat was ook een bron van akelige fouten, want als een programmeur zo’n **break**-opdracht vergat op te schrijven gebeuren er rare dingen. In C# is daarom met de traditie gebroken, en moeten de cases *verplicht* gescheiden worden met een **break**-opdracht. Wel mag je meerdere case-labels voor één (groepje) opdrachten zetten, zoals in het voorbeeld met 3 en 4 wordt gedaan.

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace Calculator
{
    public class Calculator : Form
    {
        Label resultaat;
10        TableLayoutPanel paneel;
        Processor proc;

        public Calculator()
        {
15            proc = new Processor();
            this.Text = "Calculator";

            paneel = new TableLayoutPanel();
            paneel.Dock = DockStyle.Fill;
20            paneel.ColumnCount = 4;
            for (int t = 0; t < 4; t++)
                paneel.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 25));
            for (int t = 0; t < 5; t++)
                paneel.RowStyles.Add(new RowStyle(SizeType.Percent, 20));
25

            resultaat = new Label();
            resultaat.BackColor = Color.LightYellow;
            resultaat.Text = proc.ScherM;
            resultaat.TextAlign = ContentAlignment.MiddleRight;
30            resultaat.Dock = DockStyle.Fill;
            resultaat.AutoSize = true;
            resultaat.Resize += veranderFont;

            paneel.Controls.Add(resultaat);
35            paneel.SetColumnSpan(resultaat, 4);

            for (int n = 0; n < 16; n++)
            {
                Button knop = new Button();
40                knop.Dock = DockStyle.Fill;
                knop.Text = "789/456*123+0C=-"[n].ToString();
                knop.Click += this.klik;
                knop.KeyPress += this.toets;
                knop.Resize += this.veranderFont;
45                paneel.Controls.Add(knop);
            }
            this.Controls.Add(paneel);
        }
    }
}
```

Listing 34: Calculator/Calculator.cs, deel 1 van 2

```
50     private void verwerk(char c)
    {
        switch (c)
        {
            case 'c':
            case 'C': proc.Schoon(); break;
55         case '=': proc.Reken(); break;
            case '+':
            case '-':
            case '*':
60         case '/': proc.Operatie(c); break;
            default: if (c>='0' && c<='9')
                        proc.Cijfer(c - '0');
                        break;
        }
65     resultaat.Text = proc.Scherf;
    }

    private void klik(object o, EventArgs ea)
    {
70     verwerk( ((Button)o).Text[0] );
    }

    private void toets(object o, KeyPressEventArgs kpea)
    {
75     verwerk(kpea.KeyChar);
    }

    private void veranderFont(object o, EventArgs ea)
    {
80     Control c = (Control)o;
        int h = c.Height / 2;
        if (c == resultaat) h = c.Height / 3;
        c.Font = new Font("Tahoma", h);
    }
85 }
}
```

Listing 35: Calculator/Calculator.cs, deel 2 van 2

```
namespace Calculator
{
    class Processor
    {
5        private long waarde, vorige;
        private char oper;
        private string scherm;

        public Processor()
10        {    this.Schoon();
        }

        public string Scherm
        {    get { return scherm; }
15        }

        public void Schoon()
        {    waarde = 0;
            vorige = 0;
20            oper = '+';
            scherm = "0";
        }

        public void Reken()
25        {    switch(oper)
            {
                case '+': vorige += waarde; break;
                case '-': vorige -= waarde; break;
                case '*': vorige *= waarde; break;
30                case '/': vorige /= waarde; break;
            }
            scherm = vorige.ToString();
            waarde = 0;
        }

35        public void Cijfer(int n)
        {
            waarde = 10 * waarde + n;
            scherm = waarde.ToString();
40        }

        public void Operatie(char c)
        {    this.Reken();
            oper = c;
45        }
    }
}
```

Listing 36: Calculator/Processor.cs

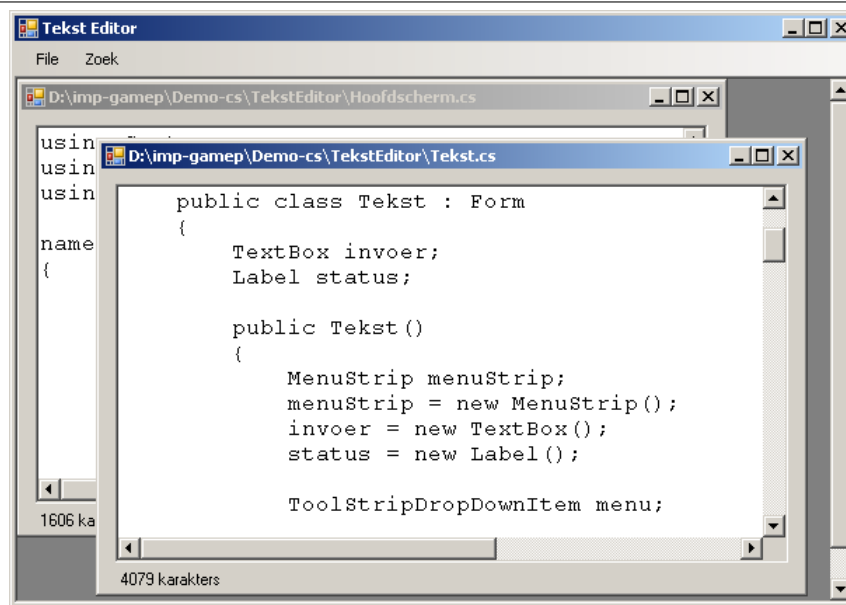
10.3 Files / abstracte klassen / Tekst-editor

Beschrijving van de casus

In deze sectie bespreken we een tekst-editor. In figuur 32 is het programma in werking te zien. Het programma heeft een zogeheten *multiple document interface* (MDI), dat wil zeggen dat er meerdere teksten tegelijk mee verwerkt kunnen worden, ieder in een eigen deelwindow. De deelwindows zwerven rond binnen het hoofdwindow, kunnen apart geopend en gesloten worden, en geminimaliseerd en gemaximaliseerd. Al deze functionaliteit krijgen we overigens grotendeels kado; er hoeft weinig extra voor geprogrammeerd te worden.

Je hebt niets aan een tekst-editor als je de teksten niet kunt bewaren voor later gebruik. Daarom heeft het programma menu-items voor ‘Open’ en ‘Save’. We zullen dus zien hoe je files kunt lezen en schrijven, en hoe je aan de gebruiker om een filenaam vraagt.

Zoals dat hoort in een tekst-editor zijn er menu-items voor ‘Zoeken’ en ‘Vervangen’ van tekst. Als de gebruiker die menukeuzes maakt, verschijnt er een dialoog waarin om zoek- en vervang-tekst wordt gevraagd. Daarmee is dit programma ook een voorbeeld van het voeren van een dialoog.



Figuur 32: Het programma TekstEditor in actie om twee van z'n eigen source-files te bewerken.

Een MDI-programma

Een programma met een multiple document interface heeft aparte klassen voor het hoofdwindow (de MDI-container) en de deelwindows (de MDI-children). In het voorbeeldprogramma zijn dat **Hoofdscherm** (listing 37) en **Tekst** (listing 38 en listing 39).

In de constructor van **Hoofdscherm** wordt het menu opgebouwd dat in het hoofdscherm te zien is: ‘File’, met de opties ‘New’, ‘Open...’ en ‘Exit’. Elk van deze items krijgt uiteraard een eigen event-handler: **nieuw**, **open** en **afsluiten**. Bij de toekenningen aan attributen zijn er behalve de gebruikelijke titeltekst en de afmetingen twee bijzondere attributen:

```
this.IsMdiContainer = true;
this.MainMenuStrip = menuStrip;
```

De eerste maakt dat dit window zich als een MDI-container gaat gedragen. Met de tweede toekenning maken we het menu dat we gemaakt hebben tot **MainMenuStrip**. Dat maakt het mogelijk dat de menu's van de deelwindows automatisch samengevoegd worden met het hoofdmenu.

In de event-handler **nieuw** wordt een MDI-childwindow aangemaakt. Het is simpelweg een kwestie van construeren van een object van de klasse **Tekst** en dat meegeven aan **Show**. Met als enige extra bijzonderheid dat in het attribuut **MdiParent** een verwijzing naar de MDI-container wordt opgeslagen.

blz. 156
blz. 157
blz. 158

```
t.MdiParent = this;
```

In de constructor van het MDI-child **Tekst** wordt de menu-items aangemaakt die alleen voor het child-window van toepassing zijn: 'Opslaan', 'Opslaan als' en 'Sluiten' in het 'File'-menu, en 'Zoek' en 'Vervang' in het 'Zoek'-menu. Dit menu wordt gewoon aan **this.Controls** toegevoegd, maar het wordt wel onzichtbaar gemaakt:

```
menuStrip.Visible = false;
```

Anders zou er toch nog een aparte menu-strip in de deelwindows zichtbaar worden, en dat is niet de bedoeling, want de inhoud van het deelwindow-menu wordt automatisch samengevoegd met het hoofdmenu: het 'Zoek'-menu komt er als tweede menu bij, en de items van het 'File'-menu worden samengevoegd. De volgorde waarin dat samenvoegen gebeurt is desgewenst nog aan te passen, maar hier doen we dat niet, waardoor de 'File'-menu-items van het child-window onder die van het container-window worden toegevoegd.

Het MDI-childwindow

In de constructor van **Tekst** worden twee controls aan het **Form** toegevoegd. Elk child-window krijgt dus die twee controls: een **Textbox** om de tekst in te tikken, een **Label** om status-informatie te tonen. We hangen een event-handler aan het **Resize**-event, die er voor zorgt dat de **TextBox** altijd vrijwel het gehele window gebruikt, met vrijlating van een kleine rand rondom en wat extra ruimte voor het status-label. In het status-label tonen we het aantal ingetikte karakters: die wordt in een event-handler voor het **TextChanged**-event van de invoer-textbox steeds aangepast.

We gebruiken de **Text**-property van het childwindow om de naam van de file aan de gebruiker te tonen. Zolang die naam nog niet bekend is (in de tijd tussen menukeuze 'Nieuw' en de eerste keer 'Save') is die string leeg. Let op het verschil tussen **this.Text** (de titel van het window met de naam van de file) en **this.invoer.Text** (de ingetikte tekst in de invoer-textbox).

Opslaan en lezen van de tekst

In de event-handlers **opslaan** en **opslaanAls** wordt de methode **schrijfNaarFile** aangeroepen, die het eigenlijke werk doet. In **opslaanAls** wordt eerst nog aan de gebruiker gevraagd wat de filenaam moet worden; in **opslaan** gebeurt dat niet, maar als er nog geen filenaam bekend was wordt daar alsnog **opslaanals** aangeroepen.

De methode **schrijfNaarFile** verradt hoe je een tekst naar een file kunt wegschrijven: maak een **StreamWriter**-object aan (geef de naam van de file mee aan de constructor), en roep van dat object de methode **Write** aan. Roep daarna de methode **Close** aan, anders blijft de file geblokkeerd voor andere programma's.

De methode **LeesVanFile** doet het omgekeerde. Hier is een **StreamReader**-object nodig, waarvan je de methode **ReadToEnd** kunt aanroepen om de gehele inhoud te lezen. Alternatieven zijn **Read** om één letter, of **ReadLine** om één regel te lezen. De methode **LeesVanFile** wordt vanuit het hoofdscherm aangeroepen als reactie op de menukeuze 'Open'. Daarom moet deze methode **public** zijn.

Vragen om een filenaam

Je kunt de gebruiker om een filenaam vragen met de standaard filenaam-dialoog. Maak daarvoor een **OpenFileDialog**-object, en roep daarvan de methode **ShowDialog** aan. Als het resultaat **DialogResult.OK** is heeft de gebruiker op 'OK' gedrukt, en kun je de property **FileName** inspecteren om de door de gebruiker uitgekozen filenaam te zien. Op het eerste gezicht zien de **OpenFileDialog** en de **SaveFileDialog** er hetzelfde uit, maar de eerste accepteert alleen bestaande filenamen, en de tweede ook nog-niet bestaande filenamen.

Vragen om de zoek/vervang-teksten

De dialoog voor de zoek- en vervang-teksten verloopt op dezelfde manier: maak een **ZoekDialog**-object aan, roep **ShowDialog** aan, en als het resultaat **DialogResult.OK** is kun je aan de gang. Het verschil is dat we klasse **ZoekDialog** zelf moeten schrijven. Deze klasse (zie listing 38) bestaat grotendeels uit een constructormethode waarin de benodigde controls worden neergezet. Zoiets is natuurlijk het gemakkelijkst met de interactieve form-designer, maar het voorbeeld in de listing is voor de overzichtelijkheid met de hand geschreven. Bovendien kunnen we zo door slim gebruik te maken van de bool-parameter **ookVervangen** dezelfde klasse zowel voor de zoek- als de zoek-en-vervang-dialoog gebruiken.

Twee toekennigen aan bijzondere properties maken dat de dialoog zich dialoog-achtig gedraagt:

```
this.AcceptButton = ok;
this.CancelButton = cancel;
```

De property `AcceptButton` maakt dat deze button de default-waarde is, en dus met een druk op de Enter-toets gekozen kan worden. De property `CancelButton` maakt dat deze button de dialoog automatisch afbreekt. Daarom heeft de cancel-button ook geen eigen `Click`-eventhandler nodig; de ok-button heeft dat gek genoeg nog wel.

Klassen voor file input/output

In het programma gebruikten we de methode `Write` van een object van de klasse `StreamWriter`, en de methode `ReadToEnd` van een object van de klasse `StreamReader`. Voor een beter begrip van de mogelijkheden van de standaardklassen voor input en output (afgekort I/O) bespreken we hier ook een aantal andere klassen. Ze worden in het voorbeeldprogramma niet gebruikt, maar kunnen in andere omstandigheden handig zijn. Een aantal veelgebruikte I/O-klassen staan (met hun subklasse-samenhang) in figuur 33. De meeste klassen staan in namespace `System.IO`, maar sommige staan in aparte namespaces voor networking, XML, cryptografie of compressie.

Stream versus Reader/Writer

Het belangrijkste verschil tussen de klassen in het schema is dat tussen de `Stream`-klassen enerzijds, en de `Reader`- en `Writer`-klassen anderzijds.

- In een `Stream` worden de aparte bytes van een file aangesproken. In een `Reader` en `Writer` gaat het om grotere eenheden: characters en strings in een `TextWriter`, ints, doubles en andere standaardtypen in een `BinaryWriter`, en samenhangende gedeelten uit een XML-beschrijving in een `XMLWriter`.
- In een `Stream` kun je afwisselend lezen en schrijven uit dezelfde file, en dus ook delen van een bestaande file overschrijven. Bij een `Reader` en een `Writer` kun je alleen maar lezen, respectievelijk schrijven.

Subklassen van Stream

Er zijn verschillende subklassen van `Stream`. Allemaal kennen ze een methode `ReadByte` om één byte te lezen. Ze verschillen in herkomst van die byte: bij een `FileStream` komt de byte uit een file, bij een `MemoryStream` komt hij uit het geheugen, en bij een `NetworkStream` komt hij via het internet binnen. Deze methode is essentieel: je zou hem zelf nooit kunnen maken.

Gek genoeg levert `ReadByte` niet een `byte` op, maar een `int`. Dat is om te kunnen signaleren dat er geen bytes meer beschikbaar zijn in de stream: dan levert de methode de waarde `-1` op.

Daarnaast is er een methode `Read` die een hele array van bytes tegelijk inleest. Dat is handig, maar als hij er niet was geweest had je altijd nog zelf `ReadByte` herhaaldelijke kunnen aanroepen in een `for`-opdracht.

Bij de constructie krijgt elke klasse de benodigde informatie mee. Bijvoorbeeld voor een `FileStream` geef je een string met de filenaam, en een parameter van type `FileMode` die aangeeft of er gelezen, geschreven of allebei gaat worden.

Abstracte klassen

Het zou de programmeurs van al die klassen nodeloos belasten als ze alledrie zelf de methode `Read` moesten schrijven, terwijl die in alle gevallen hetzelfde doet: herhaaldelijk `ReadByte` aanroepen. Gelukkig was dat niet nodig, want alle drie de klassen zijn een subklasse van `Stream`, en de methode `Read` wordt daar gedefinieerd. In de drie subklassen wordt de methode geërfd. Toch is dat wel raar, want `Read` moet `ReadByte` aanroepen, en die wordt in de diverse subklassen nou juist verschillend ingevuld. Hoe kan de klasse `Read` dan weten welke methode hij moet aanroepen?

Dat kan hij niet, en daarom is de klasse `Stream` een *abstracte klasse*. In het schema is dat aangegeven met een parallelogram-vormig kadertje. In een abstracte klasse hebben niet alle methoden een body. De klasse `Stream` zal er ongeveer als volgt uitzien:

```
abstract class Stream
{
    public abstract int ReadByte();    // geen body!
    public int Read(byte[] doel, int aantal)
    {
        for (int t=0; t<aantal; t++)
        {
            b = this.ReadByte();
```



```

        if (b==-1) return t;
        doel[t] = (byte) b;
    }
    return aantal;
}
}

```

In de subclasses wordt de methode `ReadByte` alsnog ingevuld. Bijvoorbeeld:

```

class FileStream : Stream
{
    public override int ReadByte()
    {
        int b;
        b = ...doet zijn magische ding...
        return b;
    }
}

```

Uiteindelijk kun je van een `FileStream`-object dus zowel `ReadByte` als `Read` aanroepen.

Van de abstracte klasse kun je geen object aanmaken: het zou immers niet duidelijk zijn hoe dat object de `ReadByte`-methode moet uitvoeren. Er zijn dan ook geen constructormethoden in een abstracte klasse. Maar je kunt wel variabelen van een abstracte klasse definiëren, en die een waarde geven van één van zijn subclasses. Bijvoorbeeld:

```

Stream s;
if (eenVoorwaarde)
    s = new FileStream("test", FileMode.Create);
else s = new NetworkStream(...);

```

Daarna kun je de de voorgeschiedenis vergeten, en de variabele vrolijk gebruiken. Het enige wat vanaf dit moment belangrijk is, is dat het een `Stream` betreft, die methodes `Read` en `ReadByte` kent:

```
int b = s.ReadByte();
```

Decorator Streams

Laten we aannemen dat we een stream hebben gemaakt. Of dat een file-, memory- of network-stream is, is voor het vervolg niet van belang, en dankzij de abstracte klasse kunnen we dus inderdaad zeggen: ‘laten we aannemen dat we een `Stream` hebben gemaakt’.

Dat `Stream`-object kunnen we vervolgens meegeven aan de constructor van een van de drie klassen die in het schema als ‘decorator’ is aangemerkt: `BufferedStream`, `GZipStream` of `CryptoStream`. Dat zijn zelf ook subclasses van `Stream`, en dus kennen ze de methoden `Read` en `ReadByte`. Uiteindelijk zullen ze bytes opvragen bij hun onderliggende stream, maar in de tussentijd doen ze nog iets extra’s: `BufferedStream` vraagt een groot aantal bytes tegelijk aan zijn onderliggende stream en bewaart die, zodat hij niet voor elke volgende `ReadByte` de disk weer hoeft te laten draaien; `CryptoStream` vertaalt de bytes van en naar geheimschrift, en `GZipStream` (de)comprimeert een stream.

Je kunt decorators op elkaar stapelen: zo kun je een `FileStream`-object maken, die meegeven aan de constructor van `GZipStream`, en de resulterende stream meegeven aan de constructor van `CryptoStream`.

Reader

Met een *reader* kun je grotere eenheden tegelijk lezen. Een `TextReader` kan teksten, dus strings bestaande uit characters, lezen. Er zijn onder andere de volgende methoden:

```

int    Read();        // leest een char, of geeft -1 als er niets meer is
String ReadLine();    // leest de eerstvolgende regel
String ReadToEnd();   // leest de rest van de file

```

Een `BinaryReader` kan de ingebouwde datatypes in hun binaire codering lezen. Een `int` of een `float` kost dus altijd 4 bytes, een `long` of een `double` altijd 8 bytes. Er zijn onder andere de volgende methoden:

```

byte    ReadByte();
short   ReadInt16();

```

```
int    ReadInt32();
long   ReadInt64();
uint   ReadUInt32();
double ReadDouble();
```

Readers zijn *geen* streams! Hoewel sommigen een methode `Read` of `ReadByte` hebben, zijn dit andere methodes, met een ander type resultaat ook, dan de gelijknamige methoden in `Stream`.

Vaak hebben readers een onderliggende stream; in deze zin lijken ze op decorator-streams. Zo kun je bij de constructor van een `BinaryReader` een `Stream` als parameter geven. Bij elke lees-operatie op de `BinaryReader` zal deze zijn onderliggende stream aanspreken om de eigenlijke bytes op te halen.

Maar niet *elke* reader heeft een onderliggende stream: sommige readers hebben in plaats van een onderliggende stream een onderliggende string waar ze de gegevens vandaan halen.

TextReader

De klasse `TextReader` is een abstracte klasse. Deze klasse heeft dus geen constructor-methode, en je kunt er daarom geen nieuwe objecten van maken. Maar wel van zijn subklassen, en dat is wat je dus moet doen als je een `TextReader` nodig hebt. Er is keus uit twee subklassen: een `StreamReader` is een textreader met een onderliggende stream, een `StringReader` is een textreader met een onderliggende string. In de praktijk zul je de `StreamReader` het meest nodig hebben. Dit is ook de klasse die in het voorbeeldprogramma wordt gebruikt.

Let op dat de naamgeving van de klassen inconsequent is, en daardoor verwarrend kan zijn:

- een `BinaryReader` leest *binaire dingen* uit een onderliggende stream
- een `XMLReader` leest *XML-dingen* uit een onderliggende stream
- een `TextReader` leest *tekst-dingen* uit iets onbekends onderliggends
- een `StreamReader` leest tekst-dingen uit een onderliggende *stream*
- een `StringReader` leest tekst-dingen uit een onderliggende *string*

Het inconsequente er aan is dat de naam van de reader soms aangeeft *wat* er gelezen wordt, en soms *waarvandaan*.

StreamReader

De meest gebruikte manier om een file te lezen is via een `StreamReader`. Dat is een `TextReader` met een onderliggende `Stream`, en om een `StreamReader` te kunnen maken moet je dus eerst een `Stream` hebben. Daar is volop keuze uit, maar het meest gebruikt wordt de `FileStream`. Dat levert bijvoorbeeld de volgende code op:

```
FileStream s = new FileStream("test.txt", FileMode.Open);
StreamReader r = new StreamReader(s);
String t = r.ReadToEnd();
```

Juist omdat deze combinatie zo vaak voorkomt is er voor het gemak een tweede constructor van `StreamReader`, die de filenaam direct als parameter krijgt. Een aparte `FileMode`-parameter is dan ook niet meer nodig, want bij readers wil je altijd een bestaande file openen (en bij writers juist een nieuwe file creëren). Dan blijft er dus het volgende over:

```
StreamReader r = new StreamReader("test.txt");
String t = r.ReadToEnd();
```

Dit is de aanpak die in het voorbeeldprogramma wordt gebruikt.

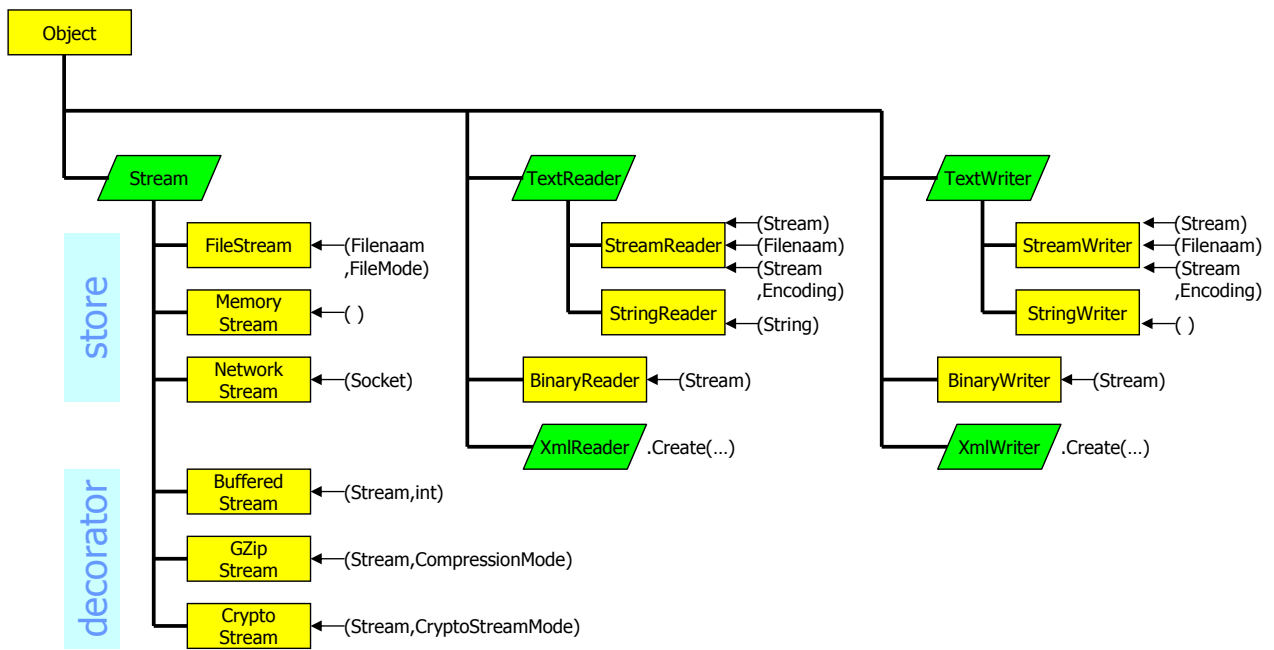
Er is overigens nog een kortere manier om alle tekst uit een file te lezen, want er is een statische methode `ReadAllText` in de klasse `File` die het maken van een `StreamReader` en de aanroep van `ReadToEnd` voor zijn rekening neemt. Je schrijft dan:

```
String t = File.ReadAllText("test.txt");
```

Zo zit de library vol met handige afkortingen, maar als je eens een keer net iets anders wilt zul je toch de lange route moeten nemen, dus daarom is het belangrijk om ook (of zelfs: juist) die te kennen.

Encoding

Het standaardtype `char` neemt 16 bits in het geheugen in, en er kunnen dus $2^{16} = 65536$ verschillende tekens mee worden gerepresenteerd. Dat is genoeg voor alle gangbare (en dode!) talen van de wereld. In het geheugen worden characters, en daarmee ook strings, gecodeerd volgens



Figuur 33: Klassen voor lezen en schrijven van files.

de Unicode-standaard. Alle letters, cijfers, symbolen, letters met accenten, en anders bijzondere samengestelde tekens hebben daarin een vaste plaats. Normaalgesproken merk je daar weinig van, want alles wat je kunt intikken (tussen aanhalingstekens in de broncode, of door de gebruiker in een textbox) wordt automatisch goed opgeslagen. Maar hoe komen deze tekens in een file terecht? Het meest voordehand liggend zou het zijn dat elke character (16 bits) als twee bytes (van ieder 8 bits) in de file worden gezet. Soms wordt dat ook wel gedaan, maar deze codering heeft twee nadelen:

- het neemt erg veel ruimte in beslag: tekst in westerse talen maakt voor een groot deel gebruik van de characters met unicode 32–127; de tweede byte is in die gevallen 0, en de file bestaat uiteindelijk voor bijna de helft uit nullen
- het is niet compatibel met files van vroeger, toen characters nog als 1 byte werden weggeschreven

Ziehier de noodzaak voor een *encoding*: de manier waarop 16-bits characters worden vertaald van en naar de bytes die in een file zijn opgeslagen. Het is de taak van een **TextReader** en een **TextWriter** om deze vertaling uit te voeren. De encoding die gebruikt moet worden kan als tweede parameter aan **StreamReader/Writer** worden meegegeven.

Hier zijn een paar mogelijke encodings:

- **Encoding.BigEndianUnicode**: de bovengenoemde voordehand liggende codering: elke character wordt als 2 bytes weggeschreven. De letter 'a' wordt 0x00 0x61, met trema 'ä' wordt het 0x00 0xE4, en de griekse 'α' wordt 0x03 0xB1.
- **Encoding.Unicode**: net zo, maar de twee bytes worden in omgekeerde volgorde opgelagen: het 'kleine eind' eerst. De letter 'a' wordt 0x61 0x00, met trema 'ä' wordt het 0xE4 0x00, en de griekse 'α' wordt 0xB1 0x03.

Dat begint dus al goed: zelfs voor de voordehand liggende codering zijn er toch al twee mogelijkheden: little-endian (gebruikelijk op Intel-gebaseerde machines, en daarom ook in Microsoft Windows, en daarmee ook in C#), en big-endian (gebruikelijk op Apple- en Unix-computers). Als je een unicode-file van het internet plukt, hoe weet je dan welk van de twee coderingen gebruikt is?

In principe is dat onmogelijk om te weten, het moet er eigenlijk altijd bij vermeld worden. Helaas is dat niet altijd het geval, en daarom is het gebruik ontstaan om *in* de file aan het begin een extra character neer te zetten. Dat is het karakter met code 0xFFFE. Ziet de reader aan het begin van

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace TekstEditor
{
    public class Hoofdscherm : Form
    {
        public Hoofdscherm()
10     {
            MenuStrip menuStrip;
            menuStrip = new MenuStrip();
            ToolStripDropDownItem menu;
            menu = new ToolStripMenuItem("File");
15     menu.DropDownItems.Add("Nieuw", null, this.nieuw);
            menu.DropDownItems.Add("Open...", null, this.open);
            menu.DropDownItems.Add("Exit", null, this.afsluiten);
            menuStrip.Items.Add(menu);
            this.Controls.Add(menuStrip);

20     this.Text = "Tekst Editor";
            this.ClientSize = new Size(600, 400);
            this.IsMdiContainer = true;
            this.MainMenuStrip = menuStrip;
25     }

    private void nieuw(object sender, EventArgs e)
    {
        Tekst t = new Tekst();
30     t.MdiParent = this;
        t.Show();
    }

    private void open(object sender, EventArgs e)
    {
35     OpenFileDialog dialoog = new OpenFileDialog();
        dialoog.Filter = "Tekstfiles|*.txt|Alle files|*.*";
        dialoog.Title = "Tekst openen...";
        if (dialoog.ShowDialog() == DialogResult.OK)
        {
40     Tekst t = new Tekst();
            t.MdiParent = this;
            t.LeesVanFile(dialoog.FileName);
            t.Show();
        }
45     }

    private void afsluiten(object sender, EventArgs e)
    {
        this.Close();
    }
50 }
}
```

Listing 37: TekstEditor/Hoofdscherm.cs

```
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

5 namespace TekstEditor
{
    public class Tekst : Form
    {
10         TextBox invoer;
        Label status;

        public Tekst()
        {
15             MenuStrip menuStrip;
            menuStrip = new MenuStrip();
            invoer = new TextBox();
            status = new Label();

20             ToolStripDropDownItem menu;
            menu = new ToolStripMenuItem("File");
            menu.MergeAction = MergeAction.MatchOnly;
            menuStrip.Items.Add(menu);
            menu.DropDownItems.Add("Opslaan", null, this.opslaan);
25             menu.DropDownItems.Add("Opslaan &als...", null, this.opslaanAls);
            menu.DropDownItems.Add("Sluiten", null, this.sluiten);
            menu = new ToolStripMenuItem("Zoek");
            menu.DropDownItems.Add("Zoek", null, this.zoek);
            menu.DropDownItems.Add("Vervang", null, this.vervang);
30             menuStrip.Items.Add(menu);
            menuStrip.Visible = false;
            this.Controls.Add(menuStrip);

            invoer.Multiline = true;
35             invoer.WordWrap = false;
            invoer.ScrollBars = ScrollBars.Both;
            invoer.Font = new Font("Courier New", 12);
            invoer.TextChanged += verander;
            this.Controls.Add(invoer);
40             this.Controls.Add(status);

            this.ClientSize = new Size(500, 300);
            this.Resize += vergroot;
            this.vergroot(null, null);
45         }

        private void vergroot(object o, EventArgs ea)
        {
            invoer.Location = new Point(10, 10);
            invoer.Size = this.ClientSize - new Size(20, 30);
            status.Location = new Point(10, this.ClientSize.Height - 15);
50         }

        private void verander(object o, EventArgs ea)
        {
            status.Text = invoer.Text.Length.ToString() + " karakters";
        }

        private void sluiten(object sender, EventArgs e)
55         {
            this.Close();
        }
    }
}
```

```

private void zoekOfVervang(bool ookVervangen)
{
    ZoekDialogoog dialoog;
    dialoog = new ZoekDialogoog(ookVervangen);
60    if (dialoog.ShowDialog(this) == DialogResult.OK)
    {
        string alles = this.invoer.Text;
        string zoek = dialoog.ZoekText.Text;
65        int pos = alles.IndexOf(zoek);
        if (pos >= 0)
        {
            if (ookVervangen)
            {
                string vervang = dialoog.VervangText.Text;
70                this.invoer.Text = alles.Replace(zoek, vervang);
                zoek = vervang;
            }
            this.invoer.Select(pos, zoek.Length);
        }
75    }
}

private void zoek(object o, EventArgs ea)
{
    this.zoekOfVervang(false);
80 }

private void vervang(object o, EventArgs ea)
{
    this.zoekOfVervang(true);
}

85 private void opslaan(object o, EventArgs ea)
{
    if (this.Text == "")
        opslaanAls(o, ea);
    else schrijfNaarFile();
}

90 private void opslaanAls(object o, EventArgs ea)
{
    SaveFileDialogoog dialoog = new SaveFileDialogoog();
    dialoog.Filter = "Tekstfiles|*.txt|Alle files|*.*";
    dialoog.Title = "Tekst opslaan als...";
    if (dialoog.ShowDialog() == DialogResult.OK)
95    {
        this.Text = dialoog.FileName;
        this.schrijfNaarFile();
    }
}

100 private void schrijfNaarFile()
{
    StreamWriter writer = new StreamWriter(this.Text);
    writer.Write(this.invoer.Text);
    writer.Close();
}

105 public void LeesVanFile(string naam)
{
    StreamReader reader = new StreamReader(naam);
    this.invoer.Text = reader.ReadToEnd();
    reader.Close();
    this.Text = naam;
110 }
}
}

```

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace TekstEditor
{
    public class ZoekDialog : Form
    {
        public TextBox ZoekText, VervangText;
        10 Button ok, cancel;
        Label zoekLabel, vervangLabel;

        public ZoekDialog(bool ookVervang)
        {
            15 ZoekText = new TextBox();
            VervangText = new TextBox();
            ok = new Button();
            cancel = new Button();
            zoekLabel = new Label();
            20 vervangLabel = new Label();

            zoekLabel.Text = "Zoek:"; zoekLabel.AutoSize = true;
            vervangLabel.Text = "Vervang:"; vervangLabel.AutoSize = true;
            ok.Text = "OK";
            25 cancel.Text = "Cancel";

            zoekLabel.Location = new Point(10, 10);
            vervangLabel.Location = new Point(10, 40);
            ZoekText.Location = new Point(70, 10); ZoekText.Size = new Size(130, 20);
            30 VervangText.Location = new Point(70, 40); VervangText.Size = new Size(130, 20);
            ok.Location = new Point(50, 80);
            cancel.Location = new Point(140, 80);

            this.Controls.Add(zoekLabel);
            35 this.Controls.Add(ZoekText);
            if (ookVervang)
            {
                this.Controls.Add(vervangLabel);
                this.Controls.Add(VervangText);
            }
            40 this.Controls.Add(ok);
            this.Controls.Add(cancel);

            this.ClientSize = new Size(220, 120);
            45 this.AcceptButton = ok;
            this.CancelButton = cancel;
            ok.Click += ok_Click;
        }
        void ok_Click(object o, EventArgs ea)
        50 { this.DialogResult = DialogResult.OK;
            this.Close();
        }
    }
}
```

een file de bytes `0xFF 0xFE` staan dan kan hij aannemen dat de file big-endian is gecodeerd, ziet de reader `0xFE 0xFF` staan dan is de file little-endian gecodeerd. Ziet de reader geen van beide, dan weet je dat de maker van de file zich niet aan deze conventie heeft gehouden en is er een probleem. Eventueel zou de reader een steekproefje kunnen nemen om te zien aan welke kant de nullen staan, maar dan moet het natuurlijk geen griekse tekst zijn.

Omgekeerd, als onze file in handen valt van een programma dat deze conventie niet kent, dan zal dat programma het karakter `0xFFFF` als ‘echt’ karakter beschouwen. Typografisch heeft dat gelukkig geen grote gevolgen, want dit karakter is een wel heel bijzonder karakter: een ‘niet-afbreekbare spatie met breedte nul’. De C# reader en writer kennen de conventie echter allebei wel, dus als je een tekst wegschrijft en weer terugleest zul je dit bijzondere karakter niet terugzien. Je ziet ze wel als je met een hexadecimale file-viewer de eerste twee bytes van een file bekijkt.

Deze coderingen zijn echter, vanwege de al genoemde nadelen, niet de meest gebruikte. Dat is:

- **Encoding.UTF8.** In deze codering worden de karakters met Unicode 0 t/m 127 als één byte weggeschreven. Dus de letter ‘a’ wordt de enkele byte `0x61`. De prijs hiervan is dat andere karakters niet meer als hun eigen code worden weggeschreven. De a-met-trema ‘ä’ wordt bijvoorbeeld `0xC3 0xA4`. Griekse, Russische, Arabische en Hebreeuwse karakters hebben allemaal een encoding van 2 bytes, maar voor nog exotischere alfabetten (Indiaas, Thais) worden dat er 3. Volgens eerdergenoemde conventie wordt ook het karakter ‘nul-brede spatie’ aan het begin van de file neergezet, wat in deze codering resulteert in 3 bytes: `0xEF 0xBB 0xBF`. Daaraan zou je een file in UTF8-codering kunnen herkennen.

Het leuke van deze codering is dat hij compatibel is met de oeroude ascii-codering: de eerste 128 karakters hebben een 1-byte codering. Als je een oude ascii-file inleest volgens de UTF8-codering, dan gaat dat dus vanzelf goed. Maar als je een file wegschrijft volgens UTF8-codering en hem kado doet aan een programma dat ascii-files verwacht, dan zal die zich verslikken in de eerste drie bytes. Je zou kunnen hopen dat zulke ouderwetse programma’s niet meer bestaan, maar de C#-libraries stellen zich minder arrogant op, en willen de oudjes desgevraagd wel terwille zijn. Je kunt daartoe een codering gebruiken die speciaal voor dit soort *backward compatibility* is gemaakt:

- **Encoding.ASCII.** Alleen karakters met code kleiner dan 128 worden weggeschreven, alle andere karakters worden vervangen door een vraagteken. Geen speciale start-markers, geen gedoe verder.

Deze codering is ook bruikbaar om files te schrijven die gebruikt gaan worden door programma’s (bijvoorbeeld webbrowsers en L^AT_EX) die weliswaar speciale tekens kunnen verwerken, maar daarvoor een eigen representatie in ascii hebben.

In de tijd die verstreken is tussen 1980 (ten tijde van de emancipatie van niet-Amerikaanse computergebruikers) en 2000 (toen Unicode begon op te komen) zijn er talloze systemen bedacht om de 1-byte-karakters met codes 128–255 nuttig te gebruiken. Coderingen die in onze streken werden gebruikt, en die je nog steeds veel kunt tegenkomen, zijn de volgende. Als je aan een C#-reader/writer de goede Encoding meegeeft, kun je ze zo nodig nog steeds gebruiken:

- **Encoding.GetEncoding("iso-8859-1"),** ook wel bekend als ‘Latin1’. Een codering waarin naast de ascii-tekens 0–127 ook de karakters 160–255 gebruikt worden, vooral voor klinkers met accenten en een paar andere speciale tekens die in west-Europa (Frankrijk, Spanje, IJsland, Denemarken) worden gebruikt.
- **Encoding.GetEncoding("Windows-1252").** Een uitbreiding van Latin1 die in gebruik was in de west-Europese versies van Windows95/98, waarin ook de karakters 128–159 worden gebruikt: het euro-teken op positie 128, een heus gulden-teken op positie 131 en nog wat Tsjechische tekens.

Latin1 is een deelverzameling van Unicode, maar Windows-1252 is dat niet (het Unicode-symbool voor het euro-teken is `0x20AC`). Er zijn nog enkele tientallen andere verouderde coderingen mogelijk, onder andere de oost-Europese windows-1250 en de Griekse windows-1253. Met **Encoding.GetEncodings()** kun je de complete lijst opvragen.

Niet voor *backward*, maar juist voor *forward compatibility* is tenslotte nog vermeldenswaard:

- **Encoding.GetEncoding("GB18030"),** de door de Chinese overheid voorgeschreven codering.

Wil je dit allemaal niet weten, dan gebruik je **StreamReader/Writer** zonder expliciete **Encoding**-parameter. Standaard wordt dan **Encoding.UTF8** gebruikt. Als je met een **TextWriter** een file wegschrijft, en hem later met een **TextReader** met dezelfde encoding weer terugleest gaat alles sowieso vanzelf goed; je hoeft je alleen maar druk te maken over encodings als je files wilt lezen en schrijven die compatibel zijn met andere programma’s.

Einde van de regel

Een ander aspect van tekstfile-compatibiliteit is de manier waarop het einde van de regel wordt gecodeerd. Hiervoor zijn maar liefst drie conventies in gebruik:

- Op Unix/Linux-computers: het einde van de regel wordt aangeduid met het ‘newline’-karakter, ascii-code 10 (0x0A).
- Op oudere Apple-computers: het einde van de regel wordt aangeduid met het ‘carriage return’-karakter, ascii-code 13 (0x0D).
- Op Windows-computers: het einde van de regel wordt aangeduid met het ‘carriage return’-èn een ‘newline’-karakter, (0x0D 0x0A).

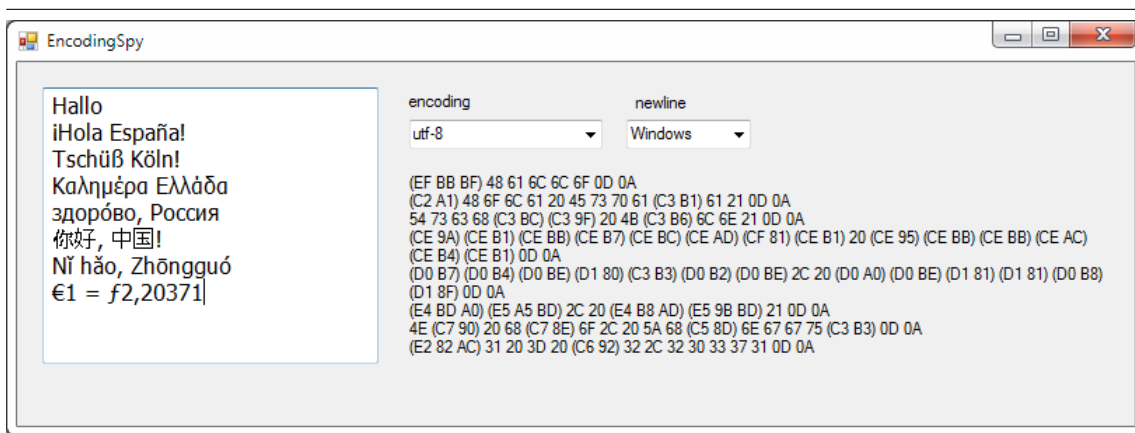
Een `TextWriter` volgt de Windows-stijl, dus methode `WriteLine` zal twee bytes aan de string toevoegen. Een `TextReader` herkent alle drie de stijlen, en voegt het ontbrekende teken automatisch toe. De methode `ReadLine` levert de ingelezen regel af zonder de regelscheidings-teken, dus dan maakt het ook niet uit. Maar als je `ReadToEnd` gebruikt en daarna de `Length` van de string bekijkt, zul je zien dat de gereleindes als twee karakteres tellen.

Gevolg is dat als je een tekstfile inleest en weer wegschrijft, de lengte kan veranderen als de stijl nog niet de Windows-stijl was. Wil je een file wegschrijven volgens een niet-Windows stijl, dan kun je de property `NewLine` van een `TextWriter` aanpassen.

Experimenteren met encodings

Om te kunnen experimenteren met de verschillende encodings, hebben we een klein demonstratieprogramma `EncodingSpy` gemaakt. De gebruiker hiermee een tekst intikken, die door het programma met een `TextWriter` wordt weggeschreven naar een file. De losse bytes van die file worden daarna met een `FileStream` teruggelezen en in hexadecimale notatie aan de gebruiker getoond. In figuur 34 is het programma in werking te zien. In listing 41 staat het relevante deel van de broncode (de userinterface is gemaakt met de visual designer, en de daardoor gegenereerde code is weggelaten).

blz. 162



Figuur 34: Het programma `EncodingSpy` schrijft een tekst naar file, en leest hem terug als bytes.

```

public partial class EncodingSpy : Form
10 {   private TextBox invoer;
      private ComboBox codering, regeleinde;
      private Label uitvoer;

      public EncodingSpy()
15 {   InitializeComponent();
          foreach (EncodingInfo info in Encoding.GetEncodings())
              codering.Items.Add(info.Name);
          invoer.TextChanged += verander;
          codering.TextChanged += verander;
20          regeleinde.TextChanged += verander;
          this.verander(null, null);
      }

      private void verander(object o, EventArgs ea)
      {   Encoding e = Encoding.GetEncoding(codering.Text);
25          Stream s1 = new FileStream("test1.txt", FileMode.Create);
          StreamWriter w1 = new StreamWriter(s1, e);
          switch (regeleinde.Text)
          {   case "Unix":   w1.NewLine = "\n";   break;
              case "Apple": w1.NewLine = "\r";   break;
30              default:     w1.NewLine = "\r\n"; break;
          }
          foreach(string regel in invoer.Lines)
              w1.WriteLine(regel);
          w1.Close();

35          Stream s2 = new FileStream("test1.txt", FileMode.Open);
          uitvoer.Text = "";
          int a=0, n = 0, b;
          while ((b = s2.ReadByte()) != -1)
40          {   if (n == 0)
                  {   a = b;
                      switch(codering.Text)
                      {case "utf-8":   if (b>=0xE0) n=3; else if (b>=0x80) n=2; break;
                        case "utf-16":
45                        case "utf-16BE": n = 2; break;
                        case "utf-32":
                        case "utf-32BE": n = 4; break;
                      }
                      if (n > 0) uitvoer.Text += "(";
50                  }
                  uitvoer.Text += String.Format("{0:X2}", b);
                  if (n > 0)
                  {   n--;
                      if (n == 0)
55                      {   uitvoer.Text += ")";
                          if (codering.Text=="utf-16BE" || codering.Text=="utf-32BE") a = b;
                      }
                  }
                  uitvoer.Text += " ";
60                  if (n==0 && (a==10 || (a==13&& regeleinde.Text=="Apple"))) uitvoer.Text += "\n";
              }
          s2.Close();
      }
}

```

10.4 Collections / interfaces

De beperkingen van arrays

Als je grote hoeveelheden gegevens van hetzelfde type in een programma wilt verwerken, kun je die opslaan in een *array*. Met een tellertje in een **for**-opdracht kun je de elementen van een array langslopen, maar je kunt ook naar believen de elementen van een array kris-kras door elkaar benaderen, om hun waarde te bekijken of eventueel te veranderen.

Dit is een zeer krachtig hulpmiddel, en de array is dan ook al zo oud als de geschiedenis van programmeertalen. Toch is vanuit modern object-georiënteerd perspectief een array eigenlijk een onding. Dit vanwege het feit dat de mogelijkheden van een array precies vastliggen, en ingebouwd zijn in de taal:

- je kunt 'm declareren: `String [] a;`
- je kunt 'm creëren: `a = new String[100];`
- je kunt een waarde op een bepaalde plaats veranderen: `a[n] = s;`
- je kunt de waarde op een bepaalde plaats bekijken: `s = a[n];`
- je kunt de bij creatie vastgelegde lengte later nog eens opvragen: `x = a.Length;`

Soms is dit precies wat je nodig hebt, en dan is er geen probleem. Maar soms zou je nog andere dingen met een array willen kunnen doen, bijvoorbeeld een extra element tussenvoegen, of de array langer maken dan bij z'n oorspronkelijke creatie is vastgelegd. Je zou voor dat soort dingen eigenlijk methoden willen kunnen toevoegen in een subklasse. Maar een array-object heeft geen bijbehorende klasse, waarvan je een subklasse zou kunnen maken.

In andere gevallen biedt een array juist meer dan je nodig hebt. Soms wil je bijvoorbeeld een gegevensverzameling opbouwen (strings die een gebruiker intikt ofzo), en die later nog eens in dezelfde volgorde langlopen (om ze op het scherm te tekenen ofzo). Je kunt daarvoor natuurlijk een array gebruiken, maar van de facilititeit dat je zo'n array kriskras kunt benaderen maak je helemaal geen gebruik: je wilt hem immers alleen maar op volgorde langslopen. Voor de facilititeit die je niet eens nodig hebt betaal je wel een dure prijs, namelijk dat je de lengte van de array vooraf moet vastleggen.

Sommige beperkingen van een array zijn wel te omzeilen. De vaste lengte bijvoorbeeld: als je die dreigt te overschreiden, kun je snel een nieuwe array creëren met de dubbele lengte van de vorige, en de oude elementen daarnaartoe kopiëren. Of de mogelijkheid om elementen tussen te voegen: als je eerste alle overige elementen een plaatsje opschuift, kun je dat in een array wel voor elkaar krijgen.

Het wordt al snel een heel gedoe met indexen en hulptellertjes. Maar als je dat netjes opbergt in methoden, kun je een klasse maken waar 'achter de schermen' weliswaar een array wordt gebruikt, maar waar je bij het gebruik van die klasse geen last van hebt, omdat dat door de diverse methoden wordt afgeschermd.

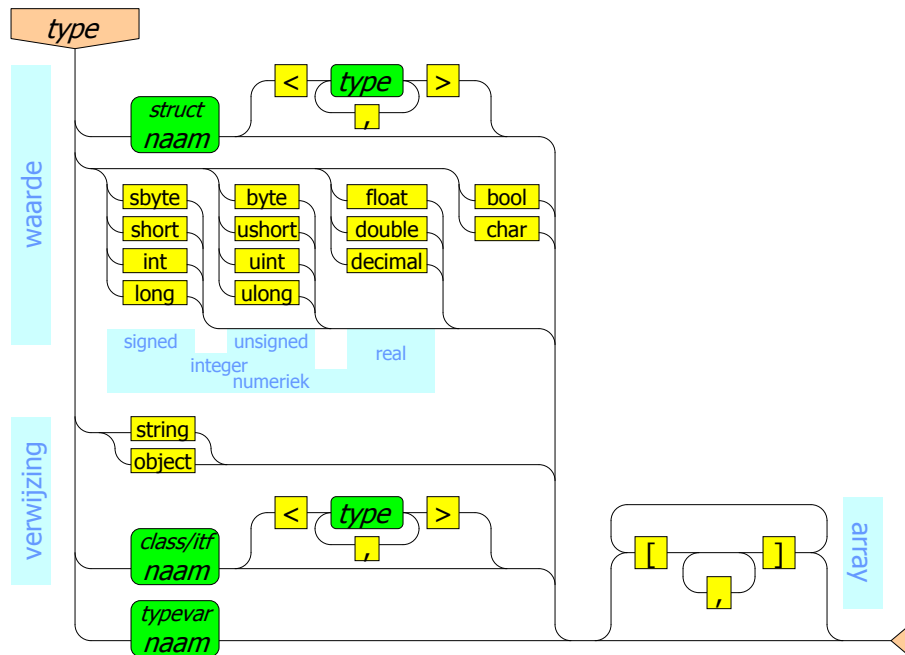
De klasse List

Die klasse bestaat natuurlijk al: hij heet **List**, en bevindt zich in de namespace **System.Collections.Generic**. Er kan hetzelfde soort dingen mee als met een gewone array:

- je kunt 'm declareren: `List<String> a;`
- je kunt 'm creëren: `a = new List<String>();`
- je kunt een waarde op een bepaalde plaats veranderen: `a[n] = s;`
- je kunt de waarde op een bepaalde plaats bekijken: `s = a[n];`
- je kunt de huidige lengte opvragen: `x = a.Count;`
- je kunt een element op een bepaalde plaats invoegen: `a.Add(n, s);`
- je kunt een element aan het eind achtervoegen: `a.Add(s);`

Opmerkelijk is dat het ophalen/veranderen van een waarde op een bepaalde plaats toch weer met vierkante haken gebeurt. Dit is mogelijk omdat het in C# mogelijk is om operatoren en indexing-met-vierkante-haken zelf te definiëren in een klasse. In de klasse **String** is dat ook al gebruikt om de characters uit een string via indexing op te vragen. Daarmee lijkt een list, net als een string, voor de onoplettende programmeur net een array, maar het is eigenlijk een echte klasse. Dat blijkt onder andere uit de aanwezigheid van andere methoden, zoals **Add**, die bij een array niet bestaan. Anders dan bij een array hoeven we bij een **List** niet de maximale lengte op te geven bij creatie: een list kan groeien op het moment dat dat nodig is. Bij de declaratie en de creatie is echter wel iets anders aan de hand: we moeten opgeven wat het type is van de elementen die in de list zullen worden opgeslagen (in het voorbeeld is dat **String**).

Dit wordt mogelijk gemaakt door de syntax van ‘type’:



Achter een klassenaam kunnen tussen punthaken nog een of meer types vermeld worden. Let op het nieuwe soort haakjes: geen ronde haakjes, geen accolades, geen vierkante haakjes, maar punthakjes. Tussen de punthakjes staat de *type-parameter* van de klasse `List`. De type-parameter maakt `List` tot een zogeheten *generiek type*: het is een type dat in meerdere situaties gebruikt kan worden (zoals `List<String>` en `List<Color>`).

blz. 168

Een voorbeeld van het gebruik van `List` staat in listing 42. De constructormethode maakt een eenvoudige userinterface waar de gebruiker een tekstveld kan intikken. Bovendien creëert deze methode een `List` waarin strings kunnen worden opgeslagen. Event-handler `klik` pakt de ingetikte string en voegt die toe aan de `List`. In event-handler `teken` worden alle tot nu toe ingetikte strings op het scherm gezet.

De interface `IList`

Het is denkbaar dan iemand in de toekomst nog eens een klasse schrijft die precies dezelfde methoden heeft als `List`, maar die bepaalde voordelen heeft: de methoden werken sneller, of de objecten kosten minder geheugenruimte (of zelfs wel allebei, maar dat is misschien te veel gevraagd). Laten we zeggen dat er in 2015 ineens een klasse `SnelleLijst` beschikbaar komt. Omdat die zoveel beter is, willen we tegen die tijd natuurlijk ons programma `ListDemo` ook aanpassen. Dat kan: we zullen dan bij de membervariabelen de declaratie moeten vervangen door

```
SnelleLijst<String> alles;
```

en in de constructor de initialisatie door

```
alles = new SnelleLijst<String>();
```

De rest van het programma kan hetzelfde blijven, als een `SnelleLijst` inderdaad precies dezelfde methodes aanbied als een `List`.

Maar niet altijd is het zo gemakkelijk om het programma aan te passen als in dit voorbeeld. Als de lijst als parameter wordt meegegeven aan methodes, moeten ook de declaraties van de parameters van die methoden worden aangepast. En dan moeten we ineens ook bij alle andere aanroepen van die methode zorgen dat ook daar de parameter een `SnelleLijst` is in plaats van een `List`. Dat gaat een hoop gedoe geven.

Met een vooruitziende blik naar de toekomst nu, kunnen we ons in 2015 echter de hoofdpijn besparen. We gaan bij de declaratie van de variabelen en de parameters niet vastleggen dat het een `List` betreft (want daar krijgen we later misschien spijt van), maar ook niet dat het een `SnelleLijst` is (want we weten nu nog niet dat die in 2015 op de markt komt). In plaats daarvan declareren we de variabelen met het type `IList`.

Het type `IList` is geen klasse, maar een zogeheten *interface*. Een interface is een specificatie van welke methoden en properties objecten van dit type moeten hebben, maar zonder daar al een implementatie van te geven. Je kunt van een interface-type dan ook geen `new` objecten maken, want in de interface is nog niet vast gelegd *hoe* de objecten in elkaar zitten, alleen *wat* ze moeten kunnen. Bij de initialisatie moeten we dus wel een concrete keus maken. We krijgen dan bijvoorbeeld:

```
IList<String> alles;
alles = new List<String>();
```

In 2015 kunnen we de declaratie laten zoals hij is, en hoeft alleen de initialisatie aangepast te worden.

Zoals je aan het voorbeeld kunt zien is het toegestaan om aan de interface-variabele van het type `IList` een object van het type `List` toe te kennen. Of, desgewenst, een object van het type `SnelleLijst`. Dit mag, omdat de klasse `List` in zijn header belooft dat het een *implementatie van de interface `IList`* is. Dat wil zeggen: dat alle methoden die in de interface beloofd zijn, ook inderdaad bestaan. Dit wordt door de compiler gecontroleerd.

Interfaces versus abstracte klassen

In feite hadden we ditzelfde effect ook kunnen bereiken met een abstracte klasse, waarvan de verschillende implementaties dan subklassen zijn. De verschillen zijn:

- In een interface hoeven de te implementeren methodes niet `virtual` gedeclareerd te worden
- Een klasse kan *meerdere* interfaces implementeren, maar slechts van één klasse de subklasse zijn.

In de library is ervoor gekozen om `IList` ene interface te maken, om hun implementaties de vrijheid te laten om een subklasse van iets anders te zijn.

De interface Collection

In werkelijkheid zit de library `System.Collection.Generic` nog iets subtieler in elkaar. In figuur 35 staat een overzicht van de interfaces en klassen die hieronder worden besproken. Er wordt onderscheid gemaakt tussen twee interfaces:

- `ICollection`, voor gegevens waarbij de volgorde er niet toe doet;
- `IList`, voor gegevens die op een lineaire volgorde staan.

Dit zijn een aantal (niet alle) methodes die gespecificeerd worden in de interface `ICollection`:

```
interface ICollection<E>
{
    void Add      (E x);
    bool Remove   (E x);
    bool Contains (E x);
    int  Count    {get;}; // een read-only property
    void Clear    ();
}
```

Alle methodes waarin een index-nummer een rol speelt, staan in de sub-interface `IList`:

```
interface IList<E> : ICollection<E>
{
    E      this [int n] {get; set;}; // dit is de notatie voor herdefinitie indicering
    int  IndexOf (E x);
    void Insert  (int n, E x);
    void RemoveAt (int n);
}
```

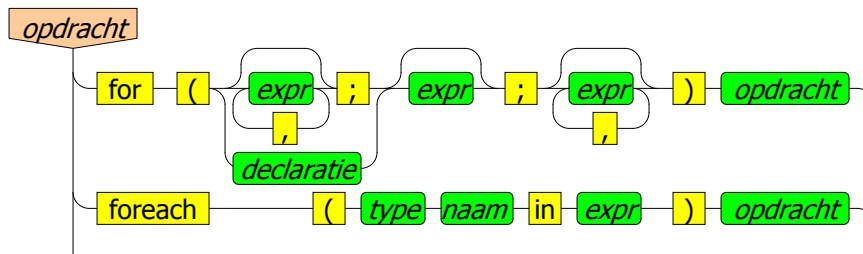
Er zijn klassen, zoals `HashSet`, die wel een implementatie zijn van `ICollection`, maar niet van `IList`. Met een object van `HashSet` kun je dus elementen toevoegen, elementen verwijderen en kijken of een bepaald element aanwezig is. Maar je kunt bijvoorbeeld niet elementen ‘aan het eind’ toevoegen, want er is niet een aanwijsbaar ‘eind’ als de spullen op een grote hoop liggen in plaats van op een rijtje.

Het langslopen van een Collection

Kun je de elementen van een `ICollection` langslopen, zoals dat in listing 42 met de elementen van een `List` gebeurde? Op het eerste gezicht niet, want je kunt niet met een tellertje via en de vierkante-haken-notatie de achtereenvolgende elementen opvragen: er is immers geen vierkante-haken-notatie in een `ICollection`.

Toch is dit wel mogelijk. Het is zelfs zo vaak nodig (en in concurrerende programmeertalen zo

eenvoudig), dat de C#-ontwerpers de verleiding niet konden weerstaan om hier een speciale notatie voor te introduceren:



In plaats van de vertrouwde twee puntkomma's in de header van de for-opdracht, ziet de header van `foreach` er anders uit, bijvoorbeeld:

```
foreach ( String s in coll )
```

De betekenis is eigenlijk gewoon zoals je het leest: 'voor elke string `s` in de collectie `coll`'. In de body kunnen we vervolgens direct gebruikmaken van de waarde van `s`. Het mooie is dat variabele `s`, die hier ter plaatse ook wordt gedeclareerd, bij elke herhaling automatisch een andere waarde heeft. Geen gedoe met tellertjes, en vierkante haken zijn helemaal niet meer nodig.

In listing 43 staat een voorbeeld van het gebruik. Het programma heeft dezelfde opzet als listing 42. Nu declareren we echter in plaats van een `List` een `ICollection`. Er hoeft maar op één plaats een keuze gemaakt te worden voor een concrete implementatie (hier in de constructormethode). Voor de rest van het programma maakt de keuze niet uit: die kan met elke `ICollection` werken. In de methode `teken` wordt een `foreach`-opdracht gebruikt om de elementen langs te lopen.

Itereren met een Iterator

Voor wie morele bezwaren heeft tegen speciale syntax voor een bijzondere situatie, is er nog een andere manier om de elementen van een `ICollection` langs te lopen. Voor dit doel is er namelijk nog een methode beschikbaar in `ICollection`. Of eigenlijk: in de super-interface `IEnumerable` waarvan `ICollection` erft:

```
interface IEnumerable<E>
{
    ...
    IEnumerator<E> GetEnumerator ();
}
```

deze methode `GetEnumerator` levert een `IEnumerator`-object op. Van dat object kun je vervolgens aanroepen: de property `Current` en de methode `MoveNext`. Oftewel: dat object is zelf weer een implementatie van de interface `IEnumerator`:

```
interface IEnumerator<E>
{
    E Current {get;}; // een read-only property
    bool MoveNext (); // geeft false als er geen volgende meer is
}
```

Deze twee methoden zijn precies wat je nodig hebt in een while-opdracht:

```
IEnumerator<String> enum;
enum = coll.GetEnumerator();
while (enum.MoveNext())
    doeIetsMet( enum.Current );
```

of nog compacter met een for-opdracht:

```
for (IEnumerator<String> enum=coll.GetEnumerator(); enum.MoveNext(); )
    doeIetsMet( enum.Current );
```

In een `ICollection` staan de gegevens niet in een bepaalde volgorde. Het is dus niet zeker dat de `IEnumerator` die door `GetEnumerator` wordt teruggegeven de elementen in dezelfde volgorde oplepelt als waarin ze zijn toegevoegd, maar je krijgt ze wel gegarandeerd allemaal te zien.

Dit is overigens precies wat er achter de schermen gebeurt als je de `foreach`-syntax gebruikt. Het programma wordt dus niet sneller of langzamer van het gebruik van `IEnumerator` in plaats van de `foreach`-syntax.

De interface ISet

Een andere sub-interface van `ICollection` is `ISet`. Hierin gedraagt methode `Add` zich hierin bijzonder. In een `ISet` garandeert `Add` dat elk element hoogstens één keer in de collectie wordt opgenomen. Zit het element er al in, dan wordt het niet nogmaals toegevoegd. Behalve dit veranderde gedrag van `Add` zitten er in `ISet` nog enkele methodes die de overlap met een andere `ISet` kunnen bepalen.

Gelijkheid wordt hierbij getest met behulp van de methode `Equals` die wordt gespecificeerd in de klasse `IEqualityComparer`. Veel klassen, onder andere `string`, implementeren `IEqualityComparer` door middel van de operator `==`.

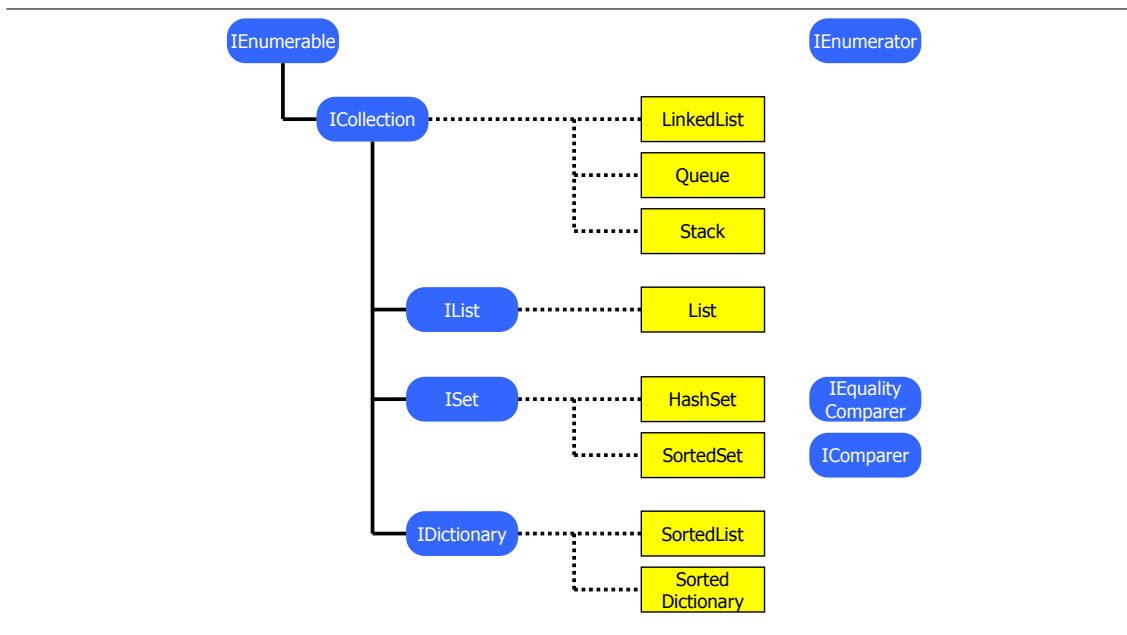
De interface IDictionary

Stel je eens voor dat in een array de elementen niet aangeduid zouden worden met een nummer, maar met een `String`. Dat zou handig zijn: je zou dan bijvoorbeeld een vertaal-tabel kunnen maken tussen het Nederlands en het Engels, door als 'index' de Nederlandse woorden te gebruiken, en als bijbehorende waarde de Engelse vertaling. Om een woord te vertalen hoef je alleen nog maar op de goede plaats de tabel te raadplegen.

Zo'n vertaaltabel is precies wat er wordt gespecificeerd door de interface `IDictionary`. Het lijkt op een `IList`, maar waar een `IList` de elementen nummert met `int`-waarden, mag als plaatsbepaling in een `IDictionary` een willekeurig objecttype gebruikt worden. In de praktijk zijn dat vaak `String`-waarden.

De interface `IDictionary` is niet alleen generiek in het element-type `E`, maar ook in het key-type `K`. De interface specificeert (onder andere) de volgende methoden:

```
interface IDictionary<K,E>
{
    E Remove (K key);
    bool Add (K key, E value)
}
```



Figuur 35: Klassen en interfaces voor collections.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
5
namespace ListDemo
{
    public class ListDemo : Form
    {
10        TextBox invoer;
        List<String> alles;

        public ListDemo()
        {
15            alles = new List<String>();

            invoer = new TextBox();    invoer.Location = new Point( 10, 10); invoer.Size = new Size(1
            Button knop = new Button(); knop.Location = new Point(130, 10); knop.Text = "Toevoegen";
            this.Controls.Add(invoer);
20            this.Controls.Add(knop);
            knop.Click += klik;
            this.Paint += teken;
        }
        private void klik(object o, EventArgs ea)
25        {
            alles.Add(invoer.Text);
            invoer.Text = "";
            this.Invalidate();
        }
        private void teken(object o, PaintEventArgs pea)
30        {
            int y=40;
            Font font = new Font("Tahoma", 12 );
            for (int t=0; t<alles.Count; t++)
35            {
                pea.Graphics.DrawString( alles[t], font, Brushes.Black, new Point(10, y));
                y+=20;
            }
        }
40    }
}
```

Listing 42: ListDemo/ListDemo.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
5
namespace CollectionDemo
{
    public class CollectionDemo : Form
    {
10        TextBox invoer;
        ICollection<String> alles;

        public CollectionDemo()
        {
15            // Kies een mogelijke implementatie:...
            //alles = new List<String>();
            //alles = new HashSet<String>();
            alles = new SortedSet<String>();

20            invoer = new TextBox(); invoer.Location = new Point(10, 10); invoer.Size = new Size(100,
            Button knop = new Button(); knop.Location = new Point(130, 10); knop.Text = "Toevoegen";
            this.Controls.Add(invoer);
            this.Controls.Add(knop);
            knop.Click += klik;
25            this.Paint += teken;
        }
        private void klik(object o, EventArgs ea)
        {
            alles.Add(invoer.Text);
30            invoer.Text = "";
            this.Invalidate();
        }
        private void teken(object o, PaintEventArgs pea)
        {
35            int y = 40;
            Font font = new Font("Tahoma", 12);
            foreach( String s in alles )
            {
                pea.Graphics.DrawString(s, font, Brushes.Black, new Point(10, y));
40                y += 20;
            }
        }
    }
}
```

Listing 43: CollectionDemo/CollectionDemo.cs

10.5 Het tekenprogramma: “Schets”

Beschrijving van het programma

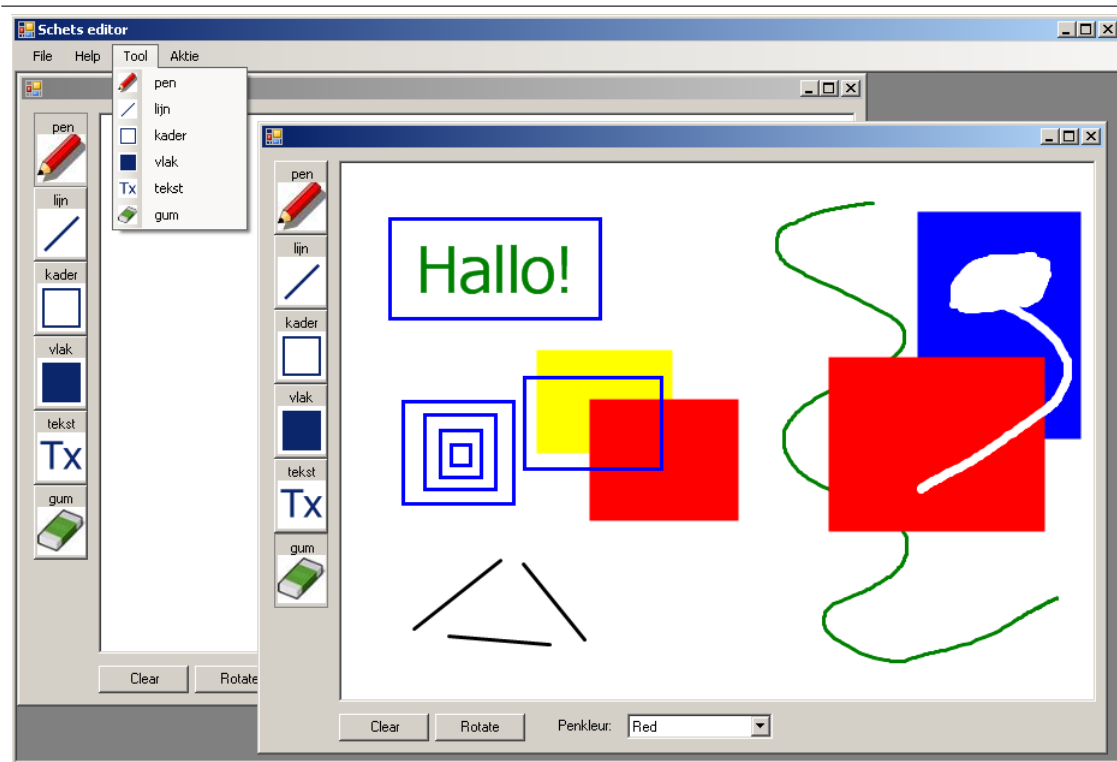
In deze sectie schrijven we een compleet tekenprogramma. In figuur 36 is het programma in werking te zien. Anders dan de bitmap-editor in sectie 10.1 kent dit programma *tools* waarmee je verschillende figuren kunt tekenen: pennetje, lijntje, open en gesloten rechthoekje, en een gummetje om weer te kunnen wissen. Ook is er een tool om tekst toe te voegen aan de tekening.

Behalve de tools zijn er onderin het window nog wat bedieningselementen, die we ter onderscheid maar *akties* zullen noemen: een knop om het plaatje leeg te maken, eentje om hem te draaien, en een combobox om de penkleur uit te kiezen.

Als de gebruiker de rechthoek-tool uitkiest kan hij/zij een blok trekken op het canvas. Het gekozen gebied wordt met een grijze contour aangegeven, en pas bij het loslaten van de muis wordt de rechthoek definitief getekend.

In plaats van met de knoppen langs de linkerrand kunnen de tools ook worden uitgekozen via het uitklapmenu ‘Tool’; de functie van de akties is ook beschikbaar via het menu ‘Aktie’. Zo’n dubbele bediening komt in professionelere programma’s vaak voor, en het is een uitdaging aan de programmeur om dit zonder code-duplicatie voor elkaar te krijgen. In dit geval zijn de methoden die de tool-buttons en de tool-menuitems maken beide geparametriseerd met een collection van de gewenste tools. De methoden die de aktie-button/-menuitems maken zijn geparametriseerd met een array van de gewenste kleuren.

Op deze manier is het gemakkelijk om in een latere versie van het programma nieuwe tools en kleuren toe te voegen, die dan automatisch op twee plaatsen in de userinterface opduiken.

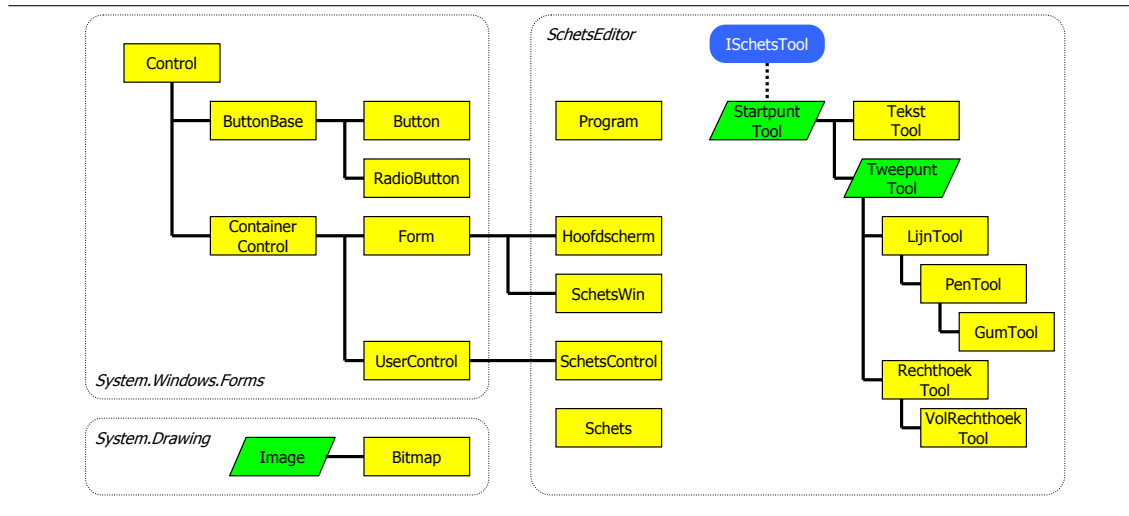


Figuur 36: Het programma Schets in werking

Opzet van het programma

In dit programma zijn we nu eens niet zuinig met klassen. Het programma bestaat uit maar liefst 14 verschillende klassen. In figuur 37 is de subklasse-hiërarchie van deze klassen getekend, en ook hoe ze samenhangen met de bestaande klassen in de library.

Zo'n groot aantal klassen is typerend voor een object-georiënteerd programma. Als je de listings van de aparte klassen leest, zul je zien dat er nergens echt opzienbarende dingen gebeuren. Veel van de klassen zijn ook erg kort, en bevatten maar een paar methoden met een handjevol opdrachten. De magie zit hem in de samenwerking van de klassen.



Figuur 37: De klasse-hiërarchie van het programma Schets

Globale opzet van de klassen

Het programma heeft een MDI-userinterface, en heeft dus dezelfde opbouw als het teksteditor-programma in sectie 10.3.

- De klasse **Program** (listing 54) bevat de methode **Main** waarmee het programma begint. Hier wordt een object gemaakt van het MDI-containerwindow. blz. 186
- De klasse **Hoofdscherm** (listing 44) modelleert het MDI-containerwindow. De belangrijkste taak hier is het afhandelen van de File-Nieuw menukeuze, waarmee een nieuw MDI-childwindow wordt aangemaakt. blz. 177
- De klasse **SchetsWin** (listing 45 t/m listing 48) modelleert het MDI-childwindow. Hierin wordt de tekening getoond, maar ook de buttons en menuitems voor de diverse tools en acties. blz. 178
blz. 181

In het teksteditor-programma werd de eigenlijke tekst getoond in een **TextBox**. Zo gemakkelijk komen we dit keer niet weg, want er is niet een standaard control om een ‘schets’ te tonen. Daarom maken we een aparte klasse voor dit doel:

- De klasse **SchetsControl** (listing 49) is een subklasse van **Usercontrol**. Het modelleert een control waarmee de gebruiker een ‘schets’ kan bewerken. blz. 182

Zoals een **TextBox** gebruikt wordt om een string te bewerken, zo wordt een **SchetsControl** gebruikt om een schets te bewerken. Voor een string bestaat een standaardklasse, maar voor een schets natuurlijk niet. Dus ook hier maken we een eigen klasse:

- De klasse **Schets** (listing 50) modelleert de eigenlijke schets. Dus niet hoe die zichtbaar wordt gemaakt, maar het ding zelf. In de huidige situatie is het een tamelijk kleine klasse, omdat hij dankbaar gebruik maakt van een membervariabele van het type **Bitmap**. blz. 183

Een hiërarchie van Tool-klassen

Bij het werken met het programma zal de gebruiker een tool uitkiezen, en vervolgens handelingen uitvoeren in de **SchetsControl**. Het indrukken van de muis is belangrijk, daarna kan de muis bewogen worden (met ingedrukte knop), en op een zeker moment wordt de muisknop weer losgelaten. Afhankelijk van de gekozen tool gebeurt er dan iets met de schets.

Wat er precies gebeurt is afhankelijk van de gekozen tool. We maken daarom voor elke tool een aparte klasse: `LijnTool`, `PenTool`, `RechthoekTool`, `VolRechthoekTool`, `TekstTool` en `GumTool` (listing 51 t/m listing 53).

Deze klassen staan in een subklasse-hiërarchie, omdat ze soms functionaliteit van een andere klasse kunnen hergebruiken. Zo is `VolRechthoekTool` een subklasse van `RechthoekTool`, omdat het tekenen van de contour tijdens het *draggen* met de muis hetzelfde is. En `GumTool` is een subklasse van `PenTool` omdat een gum in feite een soort witte pen is.

Sommige klassen in de hiërarchie zijn *abstract*, zoals `StartpuntTool` en `TweepuntTool`. Deze klassen hebben een hiërarchisch nut, maar kunnen niet zelfstandig als tool optreden.

Er is een aparte *interface* gedefinieerd waarin we specificeren wat elke tool moet kunnen: reageren op de muis (indrukken, slepen en loslaten) en reageren op het intikken van een letter. De `SchetsControl`-klasse rekent er op dat elke tool voor deze vier dingen methoden beschikbaar heeft. De interface `ISchetsTool` legt vast wat de parameters van die methoden moeten zijn. De klasse `StartpuntTool` komt de belofte van de interface na, al is van sommige methoden de body nog niet ingevuld (die methoden zijn nog *abstract*). Dieper in de hiërarchie worden echter alle methoden uiteindelijk gedefinieerd, en sommige onderweg ook weer met *override* opnieuw ingevuld.

De tool-klassen

De abstracte klasse `StartpuntTool` werkt een gemeenschappelijk kenmerk uit dat voor alle tools van belang is: met de muis klik je een startpunt aan, dat bewaard moet worden om de figuur later definitief te kunnen tekenen. Deze klasse is *abstract*, want *wat* er dan precies op dat startpunt getekend wordt is in deze klasse nog niet uitgewerkt.

In het programma hebben we twee soorten tools die voortborduren op het idee van de `StartpuntTool`: `TekstTool`, die een tekst toont op het startpunt, en `TweepuntTool` voor de tools die behalve een startpunt ook nog een tweede punt nodig hebben. Die laatste klasse is weer *abstract*, omdat we nog niet hebben uitgewerkt wat er dan met die twee punten gebeurt.

De klassen `LijnTool` en `RechthoekTool` zijn twee concrete invullingen van het abstracte `TweepuntTool`.

De klasse `VolRechthoekTool` werkt voor de contour hetzelfde als `RechthoekTool`, maar voor het definitief tekenen een beetje anders. Door hem een subklasse van `RechthoekTool` te maken vermijden we duplicatie van de code voor het contour-tekenen.

Zelfs `PenTool` blijkt nog weer wat gemeenschappelijk te hebben met `LijnTool`: een pentekening bestaat immers uit een serie (korte) lijnen. Daarom herdefinieert `PenTool` de methode die correspondeert met muis-slepen: in deze methode simuleert de `PenTool` het loslaten en onmiddellijk weer indrukken van de muistoets, door de overeenkomstige methoden aan te roepen.

Een gum-spoor, tenslotte, is in feite niets anders dan een dikke witte lijn, dus wordt `GumTool` op zijn beurt een subklasse van `PenTool`.

De klasse `SchetsWin`

De klasse `SchetsWin` is de grootste van het hele project. Dit komt voornamelijk omdat hier de userinterface (tool- en actie-buttons, menu-items) moet worden opgebouwd.

De constructormethode van deze klasse staat in listing 46. Zoals altijd heeft deze methode als taak om alle controls te maken en te configureren. De belangrijkste control (de `SchetsControl` waarin het plaatje wordt getoond) wordt direct in de constructor gemaakt; voor het aanmaken van de drie menu's en de twee rijen buttons worden hulpmethoden aangeroepen die in de volgende twee listings staan uitgewerkt.

Als eerste wordt in de constructormethode echter een array met `ISchetsTool`-objecten aangeemaakt. Het type van de elementen van deze array is `ISchetsTool`, dat wil zeggen de interface die alle tools in de hiërarchie implementeren. De elementen van de array zijn objecten van de diverse subklassen uit de hiërarchie: van elke concrete subklasse eentje, om precies te zijn.

Als membervariabele in de klasse (regel 14 in listing 45) wordt gedeclareerd:

```
ISchetsTool huidigeTool;
```

Het is de bedoeling dat deze variabele steeds de op dat moment uitgekozen tool zal bevatten. De variabele krijgt een andere waarde als de gebruiker een van de knoppen langs de linkerrand indrukt (en ook als de gebruiker een item uit het Tool-menu kiest). Hoe we dat voor elkaar krijgen kun je zien in methode `maakToolMenu` in listing 47 en methode `maakToolButtons` in listing 48.

Het Tool-menu in de klasse SchetsWin

De methode `maakToolMenu` (in het midden van listing 47) verwacht als parameter een collection met tools. Hij wordt vanuit de constructor aangeroepen met een *array* van tools als parameter. (Mag dat zomaar? Ja, dat mag: elke array implementeert automatisch de interface `ICollection`). In de body van de methode worden alle 6 de elementen van de array/collection achtereenvolgens verwerkt met een `foreach`-herhaling. Die regel code leest als een gewone Engelse zin: voor elke tool in de collection van tools wordt een menu-item aangemaakt. Belangrijk is hier de regel

blz. 180

```
item.Click += this.klikToolMenu;
```

Daarmee wordt geregeld dat, als het menuitem wordt aangeklikt, de event-handler `klikToolMenu` wordt aangeroepen.

Die methode bevindt zich een stuk eerder in de klasse, afgebeeld in listing 45:

blz. 178

```
private void klikToolMenu(object obj, EventArgs ea)
{
    this.huidigeTool = (ISchetsTool)((ToolStripMenuItem)obj).Tag;
}
```

Hier wordt dus inderdaad een nieuwe waarde aan membervariabele `huidigeTool` toegekend. Maar hoe kunnen we in de event-handler nog achterhalen welke tool correspondeert met het gekozen menuitem? Dat zit subtiel in elkaar. In de event-handler is altijd het `object` dat het event heeft veroorzaakt beschikbaar. In dit geval weten we zeker dat dat een `ToolStripMenuItem` moet zijn, dus kunnen we het object met een *cast* naar dat type converteren: `(ToolStripMenuItem)obj`. Nu hebben we dus het menuitem te pakken, maar wat is de bijbehorende tool? Kijk nog eens in de methode die het menuitem aanmaakt. Daar staat ook nog:

```
item.Tag = tool;
```

De gewenste tool is dus bewaard in de `Tag`-property van het menuitem. (Wat is dat voor rare property? De property heeft het type `object`, en kan dus gebruikt worden om een willekeurig te kiezen object bij het menuitem te bewaren. Hier komt dat dus goed van pas, want we willen een tool-object bij het menuitem bewaren).

Kijk nu weer naar de event-handler. Daar hebben we met de cast het menuitem te pakken gekregen. Van dat menuitem vragen we de `Tag`-property weer op `((ToolStripMenuItem)obj).Tag`. Let op het extra paar haakjes: de binnenste zijn van de cast, de buitenste omdat we van het *geheel* de `Tag`-property willen pakken. De compiler weet niet beter dan dat die `Tag` van het type `object` is. Maar wij weten dat we daar een `ISchetsTool` in hebben opgeslagen. Met opnieuw een cast kunnen we de compiler daarvan overtuigen, en al met al leidt dat dus tot de toekenning:

```
this.huidigeTool = (ISchetsTool)((ToolStripMenuItem)obj).Tag;
```

Gebruik van de huidige tool in klasse SchetsWin

In de constructor van `SchetsWin` wordt een membervariabele `schetscontrol` van het type `SchetsControl` geconfigureerd. Hier wordt geregeld dat bij interessante activiteiten van de gebruiker in de `SchetsControl` (bewegen en klikken van de muis, toets indrukken), de momenteel gekozen tool geactiveerd wordt. Dat gebeurt door de aanroep van een van de vier afgesproken methoden, die elke tool beloofd heeft te bezitten: `MuisVast`, `MuisDrag`, `MuisLos` of `Letter`. Het abonneren op zo'n event ziet er exotisch uit:

```
schetscontrol.MouseDown += (object o, MouseEventArgs mea) =>
{
    vast=true;
    huidigeTool.MuisVast(schetscontrol, mea.Location);
};
```

We hadden het ook op een andere manier kunnen opschrijven, en dan ziet het er bekender uit:

```
schetscontrol.MouseDown += this.muisIngedrukt;
```

Dan had er natuurlijk wel een methode `muisIngedrukt` moeten zijn, met de bekende event-handler parameters:

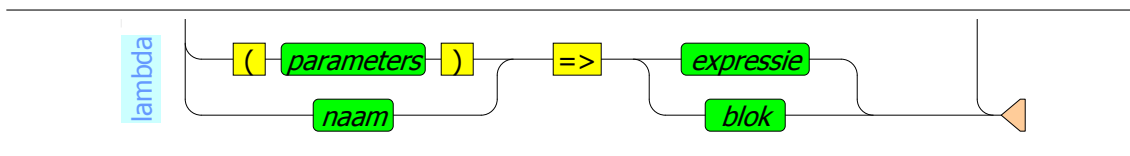
```
void muisIngedrukt(object o, MouseEventArgs mea)
{
    vast=true;
    huidigeTool.MuisVast(schetscontrol, mea.Location);
}
```

Die aanpak hebben we hier echter niet gebruikt, want dan wordt het zo'n zoekplaatje met al die aparte methoden. De gehanteerde aanpak is veel leuker: de hele body van de event-handler staat direct achter het `+=` teken, zonder dat de methode zelfs maar een naam hoeft te krijgen.

Syntax van een lambda-expressie

Zo'n *anonieme methode* staat ook wel bekend als een *lambda-expressie*. Het is, syntactisch gesproken, inderdaad een expressie, en mag daarom op de plek achter `+=` staan. De syntax van een lambda-expressie staat samengevat in figuur 38. Hij is te herkennen aan het speciale symbool `=>` in het midden. Daarachter staat in principe een *blok*, dus een rij opdrachten tussen accolades. Als de enige opdracht in dat blok een `return`-opdracht is, dan mag je de expressie die ge-returnd wordt ook direct achter de `=>` schrijven, dus zonder accolades en zonder `return`. Die notatie hebben we gebruikt in sectie 10.1, waar een anonieme methode als parameter werd meegegeven aan de methode `combineer` (listing 33 op blz. 143).

Voor de `=>` staat in principe de parameterlijst van de methode. Als de methode precies één parameter heeft, mag je de haakjes en het type weglaten. Daarmee kun je de kwadraat-functie super-compact opschrijven als anonieme functie: `x=>x*x`.



Figuur 38: Syntax van een lambda-expressie

Aanroep van tool-methoden in klasse SchetsWin

We gaan verder met de beschouwing van de constructor van `SchetsWin`. De vier events die van belang zijn in het `SchetsControl` (`MouseDown`, `MouseMove`, `MouseUp` en `KeyPress`) worden in feite vertaald in aanroepen van corresponderende tool-methoden (`MuisVast`, `MuisDrag`, `MuisLos` en `Letter`). Maar let op: de `Muis`-methoden zijn *geen* event-handlers; ze hebben immers niet de voor event-handlers karakteristieke twee parameters. Het zijn zelfbedachte methoden, die vanaf vandaag bekend staan als ‘schetstool-methoden’, want het zijn de methoden die samen de interface `ISchetsTool` vormen. Of deze library een wereldwijde doorbraak gaat beleven valt te bezien, maar de bekendheid is in ieder geval een feit binnen dit project.

De methoden zijn zo ontworpen dat ze ook de `SchetsControl` als parameter krijgen, zodat de tools –als deze methoden te zijner tijd uitgevoerd gaan worden– inderdaad een effect op de tekening kunnen uitoefenen.

Verder is er nog een subtiliteit in de vertaling van `MouseMove`: deze triggert alleen een aanroep van `MuisDrag`, als de muisknop nog ingedrukt is. Dit wordt gecontroleerd door een `bool` variabele `vast`, die bij het afhandelen van `MouseDown` op `true` wordt gezet, en bij `MouseUp` weer op `false`.

Resources

Een laatste aandachtspunt in de klasse `SchetsWin` is nog de manier waarop de icoontjes op de knoppen en menu-items worden gezet. Dat is niet heel moeilijk, want zowel een `Button` als een `MenuItem` hebben een property `Image`, waaraan je een image-object (of een `Bitmap`-object, want dat is een subklasse van `Image`) kunt toekennen. Maar hoe kom je aan een `Image` met een toepasselijk plaatje?

Het eenvoudigste is om de constructor van `Bitmap` te gebruiken, die naar keuze een string met de filenaam of een `Stream` met een reeds geopende file meekrijgt. Dan moeten de files met de plaatjes wel beschikbaar zijn in de directory waarin het programma runt. Dat betekent dat je de plaatjes moet meeleveren als je het programma aflevert. Dat is riskant, want gebruikers zijn gewend dat programma's uitsluitend uit een `.exe`-file bestaan, en raken dit soort extra hulp-bestanden nog wel eens kwijt.

In het programma is daarom een aanpak gehanteerd waarbij de plaatjes worden meeverpakt (*embedded*) in de assembly, en dus uiteindelijk in de `.exe`-file. Zo'n meeverpakt hulpbestand heet een *resource*.

Met Visual Studio kun je eenvoudig interactief resources toevoegen aan een project. Je krijgt dan een bestand `Resources.resx`. Dit is een XML-bestand, waarin de koppeling wordt gemaakt tussen de naam van de resources en de files met de plaatjes. De compiler zoekt de plaatjes op (tijdens het compileren moeten de files van de plaatjes dus wel beschikbaar zijn!) en pakt ze in de assembly in. Vanuit het programma kun je de resources aanspreken via een `ResourceManager` object. Zo’n ding wordt inderdaad geïnitieerd als membervariabele in `SchetsWin`:

```
ResourceManager resourcemanager
    = new ResourceManager("SchetsEditor.Properties.Resources"
        , Assembly.GetExecutingAssembly()
    );
```

Daarna kun je de gewenste resources terugvinden via de naam die je er, bij het aanmaken van het `.resx`-bestand, voor gekozen hebt. In dit geval hebben we de 6 resources precies dezelfde naam gegeven als de opschriften van de buttons en menu-items: ‘pen’, ‘lijn’, ‘kader’, ‘vlak’, ‘tekst’ en ‘gum’. Dit blijkt uit de twee regels bij de opbouw van het menu:

```
item.Text = tool.ToString();
item.Image = (Image)resourcemanager.GetObject(tool.ToString());
```

Je ziet hier dat de string die als `Text` op het menuitem wordt gezet, tevens de naam is van de resource die via `GetObject` bij de `resourcemanager` wordt opgevraagd. De cast naar `Image` is nodig omdat `GetObject` ook resources van andere types kan ophalen.

De teksten die corresponderen met de 6 soorten tools zijn afkomstig van een aanroep van `ToString`. In listing 53 kun je zien dat elke tool-klasse inderdaad de methode `ToString` herdefinieert om z’n eigen naam bekend te maken.

blz. 186

Visual Studio genereert naast het bestand `Resources.resx` ook nog een bestand `Resources.Designer.cs`. Dit is C#-code, waarin voor elke resource een static property wordt aangemaakt met de naam van de resource. In plaats van de opdracht

```
Image plaatje = (Image)resourcemanager.GetObject("pen");
```

is het dan mogelijk om te schrijven:

```
Image plaatje = SchetsEditor.Properties.pen;
```

In het programma laten we die mogelijkheid echter ongebruikt, omdat we in een `for`-opdracht steeds een ander plaatje willen ophalen, waarbij de naam van de resource in een string-variabele beschikbaar is. Dan is de universele methode `GetObject` handiger dan de zes aparte properties.

De klasse `SchetsControl`

De klasse `SchetsControl` is niet zo groot: hij staat in zijn geheel in listing 49. Het belangrijkste zijn de twee member-variabelen: een `Schets` waarin de eigenlijke schets wordt bewaard, en een `Color` waarin de huidige penkleur wordt bewaard.

blz. 182

De klasse bevat verder een aantal event-handlers. Twee daarvan (`VeranderKleur`) veranderen de penkleur, en vier andere (`Schoon`, `Roteer`, `veranderAfmeting` en `teken`) vertrouwen erop dat het onderliggende `Schets`-object het vuile werk opknapt.

De mysterieuze herdefinitie van `OnPaintBackground` met een lege body maakt dat de achtergrond van de control niet meer uitgewist wordt voorafgaand aan het tekenen. Dat is gewenst, omdat het beeld daardoor stabiel wordt. Dit is alleen verantwoord als het tekenen helemaal ‘dekkend’ gebeurt, maar dat is hier inderdaad het geval. Het tekenen wordt uitbesteed aan de klasse `Schets`, en die verzorgt het tekenen door een schermvullende bitmap neer te zetten.

Voor het vervolg is nog van belang dat elke `Control` een methode `GetGraphics` heeft, waarmee direct toegang tot het scherm verkregen kan worden. Deze methode wordt geërfd door `UserControl` en dus ook door `SchetsControl`. In hoofdstuk 9 hadden we streng verboden om deze methode te gebruiken, behalve voor het ‘tijdelijk’ tekenen van figuren. Dat is in dit programma het geval: het grijze kader dat tijdens het ‘draggen’ met de muis wordt getekend, gebeurt op deze manier.

Dit staat in contrast met het ‘definitieve’ tekenen. De definitieve tekening moet in de bitmap terecht komen. Voor dit doel schrijven we een methode `MaakBitmapGraphics` die een `Graphics` maakt, die aan de bitmap is gekoppeld. Bij gebruik van die graphics komen de getekende dingen niet op het scherm, maar in de bitmap terecht.

Als dan later het `Paint`-event optreedt, wordt de inhoud van de bitmap alsnog op het scherm gezet.

De klasse **Schets**

blz. 183 In de klasse **Schets** wordt pas echt de beslissing genomen om de schets als een bitmap te modeleren. In deze klasse (listing 50) wordt uitgewerkt hoe met behulp van deze bitmap de benodigde methoden **Schoon**, **Roteer**, **Teken** en **VeranderAfmeting** gerealiseerd worden. De enige methode die wat toelichting behoeft is **VeranderAfmeting**. Als de gewenste afmeting groter is dan de huidige afmeting van de bitmap, dan wordt een grotere bitmap aangemaakt, waarin de huidige gekopieerd wordt. Als de afmeting kleiner wordt, laten we de bitmap echter intact. Dit maakt dat als het window kleiner wordt gemaakt, het deel van het plaatje ‘buiten beeld’ toch nog bewaard wordt.

De Tool-hiërarchie

blz. 184 In listing 51 staat om te beginnen de definitie van de interface **ISchetsTool**. Deze zegt in feite dat een tool pas een echte tool is als hij deze vier methoden implementeert. Dit maakt dat we op regel
blz. 179 64, 68, 72 en 75 van listing 46 deze methoden blindelings mogen aanroepen op de **huidigeTool**.

Van deze interface kunnen we vervolgens diverse implementaties gaan maken. Het begint met een abstracte klasse **StartpuntTool** die alvast twee methodes invult: **MuisVast** en **MuisLos**. Op het indrukken van de muis reageert zo’n startpunt-tool door dat punt voor later gebruik te bewaren in een speciaal daarvoor gedeclareerde membervariabele, bij **MuisLos** neemt de membervariabele **kwast** de in de **SchetsControl** geldende penkleur aan. De overige twee methoden worden later in de hiërarchie ingevuld, en worden dus voorlopig **abstract** gedefinieerd. Dit is in C# (anders dan in Java) verplicht.

De klasse **TekstTool** is een concrete subklasse van **StartpuntTool**. Bij het slepen van de muis hoeft er bij deze tool niets te gebeuren, maar bij het intikken van een letter moet deze letter worden afgebeeld op de positie van het eerder bewaarde startpunt. Daarna moet de x-coördinaat van het startpunt worden verhoogd. Met hoeveel precies hangt af van de eigenschappen van het font, die met **MeasureString** kunnen worden opgevraagd.

blz. 185 In listing 52 gaat het verder met de klasse **TweepuntTool**, een andere uitbreiding van **StartpuntTool**. Deze reageert op het slepen van de muis met het tekenen van een grijze contour. Bij het loslaten van de muis wordt de figuur definitief getekend. Toetsindrukken worden genegeerd. Daarmee zijn alle vier de methoden van de interface **ISchetsTool** ingevuld, maar er zijn twee nieuwe bijgekomen: **Bezig** voor het tekenen van de contour, en **Compleet** voor het tekenen van de definitieve figuur. We kunnen **Compleet** alvast een default-invulling geven, namelijk dat hij ook alleen de contour tekent. Maar hoe je een contour tekent (lijn? rechthoek? cirkel?) kunnen we nog niet vastleggen. Deze methode wordt daarom **abstract** gedeclareerd, en hoeft dus nog geen body te hebben. De prijs is dat ook de klasse daarmee nog **abstract** is.

Wel concreet zijn de subklassen **LijnTool** en **RechthoekTool**. Deze geven ieder een eigen invulling aan de ontbrekende methode **Bezig**. De default-invulling dat het definitieve tekenen net zo gebeurt als het voorlopige, is voor deze twee tools in orde.

De klasse **VolRechthoekTool** erft het tekenen van de contour van **RechthoekTool**, maar geeft het definitieve tekenen van een **Compleet** figuur een nieuwe invulling, die het default-gedrag vervangt. De figuur wordt nu ingevuld getekend met behulp van **FillRectangle**.

blz. 186 De implementatie van **PenTool** in listing 53 is subtiel. Deze herdefinieert de methode **MuisDrag**: bij elk verslepen van de muis wordt nu zogenaamd de muis even losgelaten en weer ingedrukt. Die methoden waren al ingevuld met het tekenen van een lijn tot hier, en het beginnen van een nieuw lijnsegment. Het gevolg is dat de pen-tool bij elke muisbeweging (althans met ingedrukte muisknop) een klein lijnstukje tekent en aan een nieuw lijntje gaat beginnen. Dat is precies wat we nodig hebben voor een pen.

De klasse **GumTool** tenslotte is weer een verfijning van **PenTool**. Bij het tekenen van de contour (en daarmee ook van de definitieve figuur, want voor lijnen, en dus ook voor pennen, en dus ook voor gummen, is die hetzelfde als de contour) wordt eerst de pendikte 7 uitgekozen, en de tekenkleur wit. Op deze manier doet een gum hetzelfde als een pen, maar dan met een dikke witte lijn in plaats van een dunne gekleurde lijn.

```

using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace SchetsEditor
{
    public class Hoofdscherm : Form
    {
        MenuStrip menuStrip;

10        public Hoofdscherm()
        {
            this.ClientSize = new Size(800, 600);
            menuStrip = new MenuStrip();
            this.Controls.Add(menuStrip);
15            this.maakFileMenu();
            this.maakHelpMenu();
            this.Text = "Schets editor";
            this.IsMdiContainer = true;
            this.MainMenuStrip = menuStrip;

20        }
        private void maakFileMenu()
        {
            ToolStripDropDownItem menu;
            menu = new ToolStripMenuItem("File");
            menu.DropDownItems.Add("Nieuw", null, this.nieuw);
25            menu.DropDownItems.Add("Exit", null, this.afsluiten);
            menuStrip.Items.Add(menu);
        }
        private void maakHelpMenu()
        {
            ToolStripDropDownItem menu;
30            menu = new ToolStripMenuItem("Help");
            menu.DropDownItems.Add("Over \"Schets\"", null, this.about);
            menuStrip.Items.Add(menu);
        }
        private void about(object o, EventArgs ea)
35        {
            MessageBox.Show("Schets versie 1.0\n(c) UU Informatica 2010"
                            , "Over \"Schets\""
                            , MessageBoxButtons.OK
                            , MessageBoxIcon.Information
                            );
40        }

        private void nieuw(object sender, EventArgs e)
        {
            SchetsWin s = new SchetsWin();
            s.MdiParent = this;
45            s.Show();
        }
        private void afsluiten(object sender, EventArgs e)
        {
            this.Close();
        }

50    }
}

```

Listing 44: SchetsEditor/Hoofdscherm.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
5 using System.Reflection;
using System.Resources;

namespace SchetsEditor
{
10     public class SchetsWin : Form
    {
        MenuStrip menuStrip;
        SchetsControl schetscontrol;
        ISchetsTool huidigeTool;
15     Panel paneel;
        bool vast;
        ResourceManager resourcemanager
            = new ResourceManager("SchetsEditor.Properties.Resources"
20                                     , Assembly.GetExecutingAssembly()
                                        );

        private void veranderAfmeting(object o, EventArgs ea)
        {
            schetscontrol.Size = new Size ( this.ClientSize.Width - 70
25                                     , this.ClientSize.Height - 50);
            paneel.Location = new Point(64, this.ClientSize.Height - 30);
        }

        private void klikToolMenu(object obj, EventArgs ea)
30     {
            this.huidigeTool = (ISchetsTool)((ToolStripMenuItem)obj).Tag;
        }

        private void klikToolButton(object obj, EventArgs ea)
35     {
            this.huidigeTool = (ISchetsTool)((RadioButton)obj).Tag;
        }

        private void afsluiten(object obj, EventArgs ea)
40     {
            this.Close();
        }
    }
}
```

Listing 45: SchetsEditor/SchetsWin.cs, deel 1 van 4

```
public SchetsWin()
45 {
    ISchetsTool[] deTools = { new PenTool()
                                , new LijnTool()
                                , new RechthoekTool()
                                , new VolRechthoekTool()
50                                , new TekstTool()
                                , new GumTool()
                                };
    String[] deKleuren = { "Black", "Red", "Green", "Blue"
55                        , "Yellow", "Magenta", "Cyan"
                        };

    this.ClientSize = new Size(700, 500);
    huidigeTool = deTools[0];

60    schetscontrol = new SchetsControl();
    schetscontrol.Location = new Point(64, 10);
    schetscontrol.MouseDown += (object o, MouseEventArgs mea) =>
        {    vast=true;
            huidigeTool.MuisVast(schetscontrol, mea.Location);
65        };
    schetscontrol.MouseMove += (object o, MouseEventArgs mea) =>
        {    if (vast)
            huidigeTool.MuisDrag(schetscontrol, mea.Location);
        };
70    schetscontrol.MouseUp    += (object o, MouseEventArgs mea) =>
        {    if (vast)
            huidigeTool.MuisLos (schetscontrol, mea.Location);
            vast = false;
        };
75    schetscontrol.KeyPress += (object o, KeyPressEventArgs kpea) =>
        {    huidigeTool.Letter (schetscontrol, kpea.KeyChar);
        };
    this.Controls.Add(schetscontrol);

80    menuStrip = new MenuStrip();
    menuStrip.Visible = false;
    this.Controls.Add(menuStrip);
    this.maakFileMenu();
    this.maakToolMenu(deTools);
85    this.maakAktieMenu(deKleuren);
    this.maakToolButtons(deTools);
    this.maakAktieButtons(deKleuren);
    this.Resize += this.veranderAfmeting;
    this.veranderAfmeting(null, null);
90 }
```

Listing 46: SchetsEditor/SchetsWin.cs, deel 2 van 4

```
private void maakFileMenu()
{
    ToolStripMenuItem menu = new ToolStripMenuItem("File");
95    menu.MergeAction = MergeAction.MatchOnly;
    menu.DropDownItems.Add("Sluiten", null, this.afsluiten);
    menuStrip.Items.Add(menu);
}

100 private void maakToolMenu(ICollection<ISchetsTool> tools)
{
    ToolStripMenuItem menu = new ToolStripMenuItem("Tool");
    foreach (ISchetsTool tool in tools)
    {
        ToolStripItem item = new ToolStripMenuItem();
105        item.Tag = tool;
        item.Text = tool.ToString();
        item.Image = (Image)resourceManager.GetObject(tool.ToString());
        item.Click += this.klikToolMenu;
        menu.DropDownItems.Add(item);
110    }
    menuStrip.Items.Add(menu);
}

private void maakAktieMenu(String[] kleuren)
115 {
    ToolStripMenuItem menu = new ToolStripMenuItem("Aktie");
    menu.DropDownItems.Add("Clear", null, schetscontrol.Schoon );
    menu.DropDownItems.Add("Roteer", null, schetscontrol.Roteer );
    ToolStripMenuItem submenu = new ToolStripMenuItem("Kies kleur");
120    foreach (string k in kleuren)
        submenu.DropDownItems.Add(k, null, schetscontrol.VeranderKleurViaMenu);
    menu.DropDownItems.Add(submenu);
    menuStrip.Items.Add(menu);
}
```

Listing 47: SchetsEditor/SchetsWin.cs, deel 3 van 4

```
125     private void maakToolButtons(ICollection<ISchetsTool> tools)
    {
        int t = 0;
        foreach (ISchetsTool tool in tools)
130     {
            RadioButton b = new RadioButton();
            b.Appearance = Appearance.Button;
            b.Size = new Size(45, 62);
            b.Location = new Point(10, 10 + t * 62);
135            b.Tag = tool;
            b.Text = tool.ToString();
            b.Image = (Image)resourcemanager.GetObject(tool.ToString());
            b.TextAlign = ContentAlignment.TopCenter;
            b.ImageAlign = ContentAlignment.BottomCenter;
140            b.Click += this.klikToolButton;
            this.Controls.Add(b);
            if (t == 0) b.Select();
            t++;
        }
145    }

    private void maakAktieButtons(String[] kleuren)
    {
        paneel = new Panel();
150        paneel.Size = new Size(600, 24);
        this.Controls.Add(paneel);

        Button b; Label l; ComboBox cbb;
        b = new Button();
155        b.Text = "Clear";
        b.Location = new Point( 0, 0);
        b.Click += schetscontrol.Schoon;
        paneel.Controls.Add(b);

        b = new Button();
160        b.Text = "Rotate";
        b.Location = new Point( 80, 0);
        b.Click += schetscontrol.Roteer;
        paneel.Controls.Add(b);

165        l = new Label();
        l.Text = "Penkleur:";
        l.Location = new Point(180, 3);
        l.AutoSize = true;
170        paneel.Controls.Add(l);

        cbb = new ComboBox(); cbb.Location = new Point(240, 0);
        cbb.DropDownStyle = ComboBoxStyle.DropDownList;
        cbb.SelectedValueChanged += schetscontrol.VeranderKleur;
175        foreach (string k in kleuren)
            cbb.Items.Add(k);
        cbb.SelectedIndex = 0;
        paneel.Controls.Add(cbb);
    }
180 }
```

```

using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

5 namespace SchetsEditor
{
    public class SchetsControl : UserControl
    {
        private Schets schets;
        private Color penkleur;

10
        public Color PenKleur
        { get { return penkleur; }
        }

        public Schets Schets
15 { get { return schets; }
        }

        public SchetsControl()
        {
            this.BorderStyle = BorderStyle.Fixed3D;
            this.schets = new Schets();
            this.Paint += this.teken;
20
            this.Resize += this.veranderAfmeting;
            this.veranderAfmeting(null, null);
        }

        protected override void OnPaintBackground(PaintEventArgs e)
25 {
        }

        private void teken(object o, PaintEventArgs pea)
        {
            schets.Teken(pea.Graphics);
        }

30
        private void veranderAfmeting(object o, EventArgs ea)
        {
            schets.VeranderAfmeting(this.ClientSize);
            this.Invalidate();
        }

        public Graphics MaakBitmapGraphics()
35 {
            Graphics g = schets.BitmapGraphics;
            g.SmoothingMode = SmoothingMode.AntiAlias;
            return g;
        }

        public void Schoon(object o, EventArgs ea)
40 {
            schets.Schoon();
            this.Invalidate();
        }

        public void Roteer(object o, EventArgs ea)
        {
            schets.VeranderAfmeting(new Size(this.ClientSize.Height, this.ClientSize.Width));
45
            schets.Roteer();
            this.Invalidate();
        }

        public void VeranderKleur(object obj, EventArgs ea)
        {
            string kleurNaam = ((ComboBox)obj).Text;
50
            penkleur = Color.FromName(kleurNaam);
        }

        public void VeranderKleurViaMenu(object obj, EventArgs ea)
        {
            string kleurNaam = ((ToolStripMenuItem)obj).Text;
55
            penkleur = Color.FromName(kleurNaam);
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Drawing;

5 namespace SchetsEditor
{
    public class Schets
    {
        private Bitmap bitmap;

10        public Schets()
        {
            bitmap = new Bitmap(1, 1);
        }

15        public Graphics BitmapGraphics
        {
            get { return Graphics.FromImage(bitmap); }
        }

        public void VeranderAfmeting(Size sz)
20        {
            if (sz.Width > bitmap.Size.Width || sz.Height > bitmap.Size.Height)
            {
                Bitmap nieuw = new Bitmap( Math.Max(sz.Width,  bitmap.Size.Width)
                                           , Math.Max(sz.Height, bitmap.Size.Height)
25                                           );
                Graphics gr = Graphics.FromImage(nieuw);
                gr.FillRectangle(Brushes.White, 0, 0, sz.Width, sz.Height);
                gr.DrawImage(bitmap, 0, 0);
                bitmap = nieuw;

30            }
        }

        public void Teken(Graphics gr)
        {
            gr.DrawImage(bitmap, 0, 0);

35        }

        public void Schoon()
        {
            Graphics gr = Graphics.FromImage(bitmap);
            gr.FillRectangle(Brushes.White, 0, 0, bitmap.Width, bitmap.Height);

40        }

        public void Roteer()
        {
            bitmap.RotateFlip(RotateFlipType.Rotate90FlipNone);

45        }
    }
}
```

Listing 50: SchetsEditor/Schets.cs

```

using System;
using System.Drawing;
using System.Drawing.Drawing2D;

5 namespace SchetsEditor
{
    public interface ISchetsTool
    {
        void MuisVast(SchetsControl s, Point p);
10    void MuisDrag(SchetsControl s, Point p);
        void MuisLos(SchetsControl s, Point p);
        void Letter(SchetsControl s, char c);
    }

15    public abstract class StartpuntTool : ISchetsTool
    {
        protected Point startpunt;
        protected Brush kwast;

20        public virtual void MuisVast(SchetsControl s, Point p)
        {
            startpunt = p;
        }
        public virtual void MuisLos(SchetsControl s, Point p)
        {
25            kwast = new SolidBrush(s.PenKleur);
        }
        public abstract void MuisDrag(SchetsControl s, Point p);
        public abstract void Letter(SchetsControl s, char c);
    }

30    public class TekstTool : StartpuntTool
    {
        public override string ToString() { return "tekst"; }

        public override void MuisDrag(SchetsControl s, Point p) { }

35        public override void Letter(SchetsControl s, char c)
        {
            if (c >= 32)
            {
40                Graphics gr = s.MaakBitmapGraphics();
                Font font = new Font("Tahoma", 40);
                string tekst = c.ToString();
                SizeF sz =
                    gr.MeasureString(tekst, font, this.startpunt, StringFormat.GenericTypographic);
45                gr.DrawString(tekst, font, kwast,
                                this.startpunt, StringFormat.GenericTypographic);
                // gr.DrawRectangle(Pens.Black, startpunt.X, startpunt.Y, sz.Width, sz.Height);
                startpunt.X += (int)sz.Width;
                s.Invalidate();

50            }
        }
    }
}

```

Listing 51: SchetsEditor/Tools.cs, deel 1 van 3


```
public abstract class TweepuntTool : StartpuntTool
55 {
    public static Rectangle Punten2Rechthoek(Point p1, Point p2)
    {
        return new Rectangle( new Point(Math.Min(p1.X,p2.X), Math.Min(p1.Y,p2.Y))
                               , new Size (Math.Abs(p1.X-p2.X), Math.Abs(p1.Y-p2.Y))
                               );
60    }
    public static Pen MaakPen(Brush b, int dikte)
    {
        Pen pen = new Pen(b, dikte);
        pen.StartCap = LineCap.Round;
        pen.EndCap = LineCap.Round;
65        return pen;
    }
    public override void MuisVast(SchetsControl s, Point p)
    {
        base.MuisVast(s, p);
        kwast = Brushes.Gray;
70    }
    public override void MuisDrag(SchetsControl s, Point p)
    {
        s.Refresh();
        this.Bezig(s.CreateGraphics(), this.startpunt, p);
    }
75    public override void MuisLos(SchetsControl s, Point p)
    {
        base.MuisLos(s, p);
        this.Compleet(s.MaakBitmapGraphics(), this.startpunt, p);
        s.Invalidate();
    }
80    public override void Letter(SchetsControl s, char c)
    {
    }
    public abstract void Bezig(Graphics g, Point p1, Point p2);

85    public virtual void Compleet(Graphics g, Point p1, Point p2)
    {
        this.Bezig(g, p1, p2);
    }
}

90 public class RechthoekTool : TweepuntTool
{
    public override string ToString() { return "kader"; }

    public override void Bezig(Graphics g, Point p1, Point p2)
95    {
        g.DrawRectangle(MaakPen(kwast,3), TweepuntTool.Punten2Rechthoek(p1, p2));
    }
}

public class VolRechthoekTool : RechthoekTool
100 {
    public override string ToString() { return "vlak"; }

    public override void Compleet(Graphics g, Point p1, Point p2)
    {
        g.FillRectangle(kwast, TweepuntTool.Punten2Rechthoek(p1, p2));
105    }
}
```

```

    public class LijnTool : TweepuntTool
    {
110         public override string ToString() { return "lijn"; }

        public override void Bezig(Graphics g, Point p1, Point p2)
        {
            g.DrawLine(MaakPen(this.kwast,3), p1, p2);
        }
115     }

    public class PenTool : LijnTool
    {
        public override string ToString() { return "pen"; }
120
        public override void MuisDrag(SchetsControl s, Point p)
        {
            this.MuisLos(s, p);
            this.MuisVast(s, p);
        }
125     }

    public class GumTool : PenTool
    {
        public override string ToString() { return "gum"; }
130
        public override void Bezig(Graphics g, Point p1, Point p2)
        {
            g.DrawLine(MaakPen(Brushes.White, 7), p1, p2);
        }
    }
135 }

```

Listing 53: SchetsEditor/Tools.cs, deel 3 van 3

```

using System;
using System.Windows.Forms;

namespace SchetsEditor
5 {
    static class Program
    {
        [STAThread]
        static void Main()
10     {
            Application.Run(new Hoofdscherm());
        }
    }
}

```

Listing 54: SchetsEditor/Program.cs

Hoofdstuk 11

Algoritmen

11.1 Een console-tool om tekst te zoeken

Console input/output

Niet elk programma hoeft een window te maken om zijn resultaten in te tonen. Hoewel op PC's het gebruik van DOS-programma's nauwelijks meer gangbaar is, is het vooral op Unix/Linux-computers vrij gebruikelijk dat programma's tekstuitvoer genereren in hetzelfde window waarin ze (door het intikken van een commando) werden gestart. Dit soort programma's kunnen de muis niet gebruiken, en krijgen al hun input van de gebruiker dus vanaf het toetsenbord.

De klasse `Console`

In de klasse `Console` zitten twee interessante variabelen, die nodig zijn bij console-programma's: `Console.In` en `Console.Out`. Dit zijn variabelen met als type respectievelijk `TextReader` en `TextWriter`.

De variabele `Console.In` stelt de tekst voor die afkomstig is van het toetsenbord. Alle tekst die je schrijft naar `Console.Out` verschijnt als tekst op de console.

Bij het starten van het programma kan de gebruiker er echter voor kiezen om de standaardinvoer ergens anders vandaan te laten komen dan het toetsenbord, bijvoorbeeld uit een file, of uit de uitvoer van een ander programma. Evenzo kan de gebruiker de standaarduitvoer *redirecten*: naar een file, of naar de input van een ander programma. Binnen in het C#-programma merk je daar niets van: je treft hoe dan ook een `Console` met een `In` en een `Out` aan.

De statische methode `Console.WriteLine` die we in hoofdstuk 2 gebruikten is eigenlijk alleen maar een hulpmethode, die direct `Console.In.WriteLine` aanroept.

blz. 15

Voorbeeld: `grep`-programma

Een voorbeeld van een typisch console-programma is het programma `grep`. Onder Unix is dit een standaard aanwezig programma, waarvan de naam betekent: "get regular expression pattern". We gaan dit programma, althans een eenvoudige versie ervan, zelf even maken.

Bij het starten van het programma moet op de commandoregel een tekst worden gespecificeerd (het zogenaamde *patroon*), en een of meer filenamen. Het programma zal nu alle regels uit de genoemde files tonen waarin het tekstpatroon voorkomt. In figuur 39 toont het programma in werking, zoekend naar het woord 'void' in zijn eigen sourcefile. Voorafgaand aan onze eigen versie van `Grep` wordt hier ook de 'echte' versie gerund. Die echte versie moet op de commandoregel ook de opties `-H -n` ontvangen om ertoe verleid te worden de filenaam en het regelnummer af te drukken; onze versie doet dat altijd.

Het patroon en de filenamen zijn in de methode `Main` beschikbaar in de vorm van de string-array-parameter: het patroon staat op de nulde positie in deze array, en de filenamen op de posities vanaf 1.

Omdat eenzelfde soort actie op alle files moet worden uitgevoerd, is het het gemakkelijkst om een aparte methode te maken die het werk voor één file uitvoert. Die methode kan dan vanuit `main` voor alle files worden aangeroepen. In listing 55 heet deze methode `bewerk`. De methode kan (en moet zelfs) statisch zijn, omdat `Main`, van waaruit `bewerk` wordt aangeroepen, dat ook is.

blz. 189

Het werk dat resteert in de methode `Main` is controleren of de gebruiker wel voldoende strings heeft ingetikt. Zo niet, dan krijgt de gebruiker een korte uitleg van wat hij geacht wordt in te tikken.

```

jeroen@catrjip /d/imp-gamep/Demo-cs/Grep/bin/Debug
$ Grep.exe -H -n void ../Grep.cs
../Grep.cs:6:      private static void bewerk(String patroon, String filenaam >
../Grep.cs:27:     static void Main(string[] args)

jeroen@catrjip /d/imp-gamep/Demo-cs/Grep/bin/Debug
$ ./Grep.exe void ../Grep.cs
../Grep.cs, line 6:      private static void bewerk(String patroon, String filenaam >
../Grep.cs, line 27:     static void Main(string[] args)

jeroen@catrjip /d/imp-gamep/Demo-cs/Grep/bin/Debug
$

```

Figuur 39: Het programma Grep in werking, zoekend naar het woord ‘void’ in zijn eigen sourcefile. Voor onze eigen versie van Grep wordt hier ook de ‘echte’ versie gerund.

Toekenningen hebben een resultaat

Toekenningen aan variabelen hebben we tot nu toe altijd als opdracht gebruikt. Maar toekenningen zijn ook bruikbaar als expressie. Terwijl deze expressie wordt “uitgerekend” krijgt de variabele zijn nieuwe waarde. Deze waarde is bovendien de resultaatwaarde van de expressie, en kan dus in een grotere expressie worden gebruikt.

Een simpel voorbeeld van zo’n toekenning-expressie is:

```
int x, y;
x = (y=0);
```

De toekenning `y=0` is hier als expressie gebruikt, die zelf aan de rechterkant van de toekenning-opdracht aan `x` staat. Met de toekenning `y=0` krijgt `y` de waarde 0. Dit is bovendien de waarde van het geheel (`y=0`), en dus ook `x` krijgt de waarde 0.

Voor dit geval is het niet zo’n nuttige notatie, want we hadden net zo goed twee aparte toekenningen kunnen doen:

```
x=0; y=0;
```

Toekenning-expressies worden dan ook zelden gebruikt. Er is echter één situatie waarin ze goed van pas komen, en dat is bij het lezen van files. Dit is de naieve manier alle regels van een file te lezen:

```
String regel;
regel = invoer.ReadLine();
while (regel!=null)
{
    doeIetsMet(regel);
    regel = invoer.ReadLine();
}
```

De aanroep van de methode `ReadLine` staat op twee plaatsen in het programma: eenmaal om de eerste regel te lezen; dat moet immers gebeuren alvorens de test `regel!=null` kan worden uitgevoerd. En eenmaal aan het eind van de body van de `while`-opdracht, om de volgende regel te lezen.

Met gebruikmaking van een toekenning-expressie kan dit handiger: we kunnen in dezelfde expressie die test of de regel ongelijk `null` is, meteen ook de regel inlezen. De opdracht wordt dan:

```
while ( (regel=invoer.ReadLine()) != null)
    doeIetsMet(regel);
```

Omdat er nu nog maar één opdracht in de body staat, kunnen zelfs de accolades vervallen. In het voorbeeldprogramma wordt bovendien met een teller bijgehouden wat het regelnummer is. In plaats van een `while`-opdracht kunnen we dus beter een `for`-opdracht gebruiken, in de header waarvan alles gecombineerd wordt:

```
for (nr=1; (regel=invoer.ReadLine())!=null; nr++)
    doeIetsMet(regel);
```

```
using System;
using System.IO;

class Grep
5 {
    private static void bewerk(String patroon, String filenaam )
    {
        TextReader reader;
        String regel;

10        try
        {
            if (filenaam == "")
                reader = Console.In;
15        else
            reader = new StreamReader(filenaam);

            for (int nr=1; (regel=reader.ReadLine()) != null; nr++)
                if (regel.Contains(patroon))
20                Console.WriteLine( filenaam + ", line " + nr + ": " + regel );
        }
        catch (Exception e)
        { Console.WriteLine( filenaam + ": " + e );
        }
25    }

    static void Main(string[] args)
    {
        switch (args.Length)
30        {
            case 0: Console.WriteLine("Usage: Grep pattern [files]");
                    break;
            case 1: Grep.bewerk( args[0], "" );
                    break;
35        default: for (int i=1; i<args.Length; i++)
                    Grep.bewerk( args[0], args[i] );
                    break;
        }
    }
40 }
```

Listing 55: Grep/Grep.cs

Lezen van Console.In

Het is in console-programma's zoals grep gebruikelijk om, als er geen filenamen worden gespecificeerd, in plaats daarvan vanaf `Console.In` te lezen. Daarom staat er in de methode `bewerk` een if-opdracht die beslist welke `TextReader` er gebruikt wordt: `Console.In`, of een `StreamReader` die met een file is geassocieerd.

11.2 Automatische taalherkenning

Beschrijving van de casus

Het programma in deze toepassing heet 'Taalherkenning', omdat het van een ingetikte tekst kan herkennen in welke taal het geschreven is. Om zo'n programma te kunnen schrijven moeten we ons eerst afvragen: hoe doe je dat eigenlijk, talen herkennen? Bekijk eens het onderstaande couplet uit het gedicht 'Jabberwocky' van Lewis Carroll, en vijf vertalingen daarvan:

'Twas brillig, and the slithy toves Did gyre and gimble in the wabe: All mimsy were the borogoves, And the mome raths outgrabe.	Il brigue: les tôves libricilleux Se gyrent en vrillant dans la guave. Enmîmés sont les goubebosqueux Et le mômerade horsgrave.
Es brillig war. Die schlichten Toven Wirrten und wimmelten in Waben; Und aller-mümsigen Burggroven Die mohmen Râth' ausgraben.	't Wier bradig en de slijp'le torfs driltolden op de wijde weep. Misbrozig stonden borogorfs 't verdoolde grasvark schreep.
'Era brilligio, y los rebalosioso mocasos Giraban y Girareon en las ondabolsciabo: Todo debilirana estaban las ramianandos Y los momiasera ratianeras fuerandabando.	Rostiva, e gli ascili titovi Andean nell'eda a triva e a spiva: Mievi stean i borogovi, E i duti ranchi esgrivan.

De woorden zijn in alle vertalingen net zo onzinnig als in het origineel. Toch zie je direct om welke taal het gaat. Dat kan blijkbaar zonder de inhoud te begrijpen, dus puur op grond van de vorm van de tekst.

Een aspect van de vorm is de frequentie van de verschillende letters. Ook als je geen Albanees en Fins kent kun je een Albanese tekst herkennen aan de vele q's en ë's, en een Finse tekst aan de vele u's en andere klinkers. Op deze manier moet het programma ook werken: op grond van de letterfrequentie wordt de beslissing genomen in welke taal de aangeboden tekst waarschijnlijk geschreven is.

Natuurlijk 'kent' het programma de talen niet echt, zoals wij Nederlands of Engels kennen. Maar wat doet dat er toe, als hij ze op deze manier wel feilloos kan onderscheiden? Of zou je toch kunnen zeggen dat een computer zo'n taal dan 'kent'?

Modellering

We kunnen natuurlijk de kennis over diverse talen in het programma inbouwen. Het programma wordt daar behoorlijk lang van; met veel if-opdrachten moeten de kenmerken van de verschillende talen worden onderscheiden. Bovendien moeten we van tevoren weten welke talen de gebruiker zoal wil onderscheiden, en voor al deze talen een letterfrequentie-analyse plegen.

Omdat het programma toch al de letterfrequentie voor de aangeboden tekst moet kunnen bepalen, kunnen we het programma met gebruikmaking van dezelfde methoden ook de letterfrequenties in de diverse talen laten uitzoeken.

Daarvoor is het nodig dat we van de te herkennen talen een representatieve *voorbeeldtekst* aanbieden. Op grond daarvan kan het programma een beeld vormen van de letterfrequenties in de verschillende talen.

In het programma zelf zit dus geen kennis over talen ingebouwd; het programma 'leert' door het bekijken van voorbeelden. Een voordeel hiervan is dat de gebruiker zelf kan beslissen welke talen er herkend moeten kunnen worden (op voorwaarde dat er een voorbeeldtekst beschikbaar is).

In het programma zijn tabellen nodig die voor elke letter uit het alfabet aangeven hoe vaak hij voorkomt. We beperken ons tot letters; leestekens en andere symbolen worden niet in de analyse meegenomen.

Hoofdletters en kleine letters worden niet onderscheiden; dat zou maar afleiden van de hoofdzak: de letterfrequentie. Voor het gemak beperken we ons tot het westerse 26-letter alfabet (het programma kan desgewenst aangepast worden om ook andere letters mee te tellen).

Vershil van frequentietabellen

We moeten een manier bedenken waarmee we kunnen beslissen wanneer een frequentietabel meer op een bepaalde tabel lijkt dan op een andere.

Daarvoor moeten we een maat hebben voor het ‘verschil’ van twee tabellen: hoe groter de waarde daarvan, hoe minder de tabellen op elkaar lijken. Een tabel lijkt dan het meeste op de tabel waarmee het verschil het kleinste is.

Er zijn verschillende mogelijkheden voor de definitie van zo’n maat. We moeten een aantal arbitraire beslissingen nemen. Misschien kan het programma later nog verbeterd worden door deze keuzes aan te passen of te verfijnen.

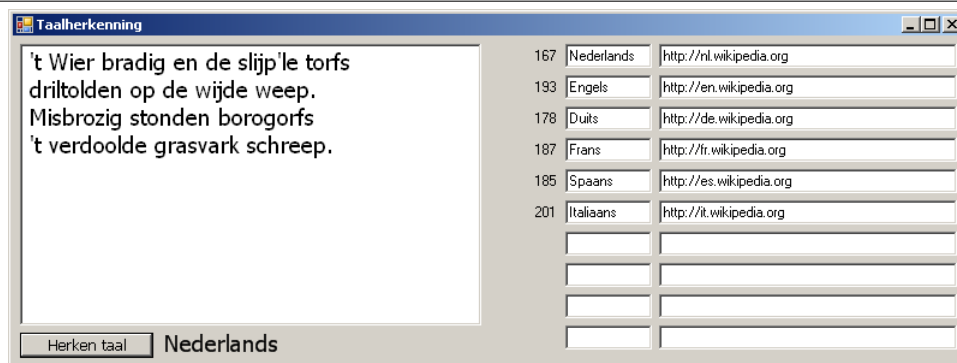
- De frequentie van alle letters draagt bij in het totaal. We bepalen daarom het verschil van alle 26 individuele tellers, en *totaliseren* dat. Eventueel zouden sommige letters zwaarder meegewogen kunnen worden dan andere. Het is echter niet meteen duidelijk welke letters dat zouden moeten zijn, dus daarom wegen we alle letters voorlopig maar even zwaar.
- In een langere tekst komen letters natuurlijk vaker voor dan in kortere teksten. Omdat de lengte van de tekst niet van invloed mag zijn op de keuze van de taal, vergelijken we de *relatieve* letterfrequenties ten opzichte van het totaal.
- Zowel een te kleine als een te grote letterfrequentie moet bijdragen aan een groter verschil. Het mag niet zo zijn dat een tekort aan a’s een teveel aan b’s opheft. Daarom gebruiken we de *absolute waarde* van het (relatieve) verschil van de tellers. Het kwadraat van het verschil had ook gekund. Hier is nog ruimte om te experimenteren.

Als maat voor het verschil van twee tabellen nemen we dus *de som van de absolute waarden van de verschillen tussen de relatieve letterfrequenties*. Om op een mooi getal tussen 0 en 1 uit te komen delen we het geheel nog door 26.

Gebruikersinterface

Op de gebruikersinterface is ruimte om de te determineren tekst in te tikken. Verder moet de gebruiker een aantal voorbeeldtalen kunnen aangeven. Daarvan wordt steeds de naam opgegeven, en een representatieve voorbeeldtekst. Die voorbeeldtekst wordt gegeven in de vorm van een filenaam of een URL op het www. Er zijn genoeg buitenlandse sites waarvandaan we gemakkelijk voorbeeldteksten kunnen ophalen. Verder is er een knop aanwezig, waarmee de gebruiker het herken-proces kan starten. Tenslotte is er een Label waarop het resultaat getoond wordt.

Met een drietal `TableLayoutPanel`-objecten kunnen we een GUI in elkaar zetten die een beetje flexibel meegroeit met resizing van het window. In figuur 40 is het programma in werking te zien.



Figuur 40: Het programma Taalherkenning in werking

Opbouw van de klassen

Een frequentietabel is in dit programma duidelijk een belangrijk object. Nu hadden we in sectie 9.4

al een klasse **TurfTab** gemaakt, waarmee de frequentie van afzonderlijke letters geturfd kan worden. Die klasse komt hier ook weer goed van pas. Er zat in de klasse echter nog geen methode waarmee de relatieve frequentie van de letters bepaald kon worden, en er was ook nog geen methode om de invoer uit files of van het internet te lezen. We gaan daarom een subklasse **RelTurfTab** maken van **TurfTab**, waarin de benodigde extra methoden worden toegevoegd.

In een andere klasse, genaamd **Taal**, bouwen we de userinterface op. In de **Click**-eventhandler van de button wordt het vergelijken van de frequentietabellen gestuurd.

Algoritme ‘zoek de kleinste’

Hart van het programma is die event-handler **klik**. Daarin maken we om te beginnen een frequentietabel van de onbekende tekst, die uit de invoer-textbox wordt geplukt:

```
onbekend = new RelTurfTab();
onbekend.Turf( invoer.Text );
```

Die moet vergeleken worden met alle voorbeeldteksten. In een for-opdracht pakken we daarom één voor één de URL van zo'n voorbeeldtekst, en construeren daarmee een nieuwe frequentietabel:

```
for (int t=0; t<aantal; t++)
{
    naam = url[t].Text;
    voorbeeld = new RelTurfTab();
    voorbeeld.Lees(naam);
}
```

De frequentietabel van de onbekende tekst wordt nu vergeleken met die van de onderhavige voorbeeldtekst. Het resultaat is een verschilwaarde:

```
verschil = onbekend.Verschil(voorbeeld);
```

Is die verschilwaarde kleiner dan wat we tot nu toe dachten dat de kleinste was, dan hebben we een nieuwe waarde voor kleinste gevonden. Bovendien onthouden we in dat geval de naam van de bijbehorende taal:

```
if (verschil<kleinste)
{
    kleinste = verschil;
    antwoord = taal[t].Text;
}
```

Na afloop van de for-opdracht kunnen we het antwoord aan de gebruiker tonen:

```
}
uitvoer.Text = antwoord;
```

Natuurlijk moeten de variabelen kleinste en antwoord worden geïnitieerd. Dat gebeurt helemaal aan het begin met:

```
kleinste = 1.0;
antwoord = "onbekend";
```

De waarde 1.0 is de groter dan het grootst mogelijke verschil tussen frequentietabellen. Dat kan dus alleen maar kleiner worden. De beginwaarde van **antwoord** is een relevante tekst voor het geval alle teksten onverhoopt onleesbaar blijken (bijvoorbeeld als er geen internet-verbinding is).

Vergelijk relatieve frequenties

De methode **Verschil** in de klasse **RelTurfTab** moet het verschil van twee frequentietabellen berekenen, op de in de vorige sectie besproken manier. De methode heeft één van deze twee tabellen onder handen (aanspreekbaar met de naam **this**). De tweede tabel geven we als parameter van de methode mee (met de naam **andere**).

De methode verloopt precies zoals besproken:

```
public double Verschil(RelTurfTab andere)
{
    double totaal = 0.0;
    for (int i=0; i<26; i++)
        totaal += Math.abs( this.relatief(i) - andere.relatief(i) );
    return totaal/26;
}
```

De hierin benodigde methode **relatief** maakt gebruik van de van **TurfTab** geërfde variabelen **tellers** en **totaal**.

Initialisaties

Voor het aantal talen dat de gebruiker als voorbeeld kan invoeren is in het programma gekozen voor 10. Het had echter net zo goed iets meer of minder kunnen zijn. Toch is die waarde 10 steeds nodig: bij het creëren van de array, als bovengrens van de for-teller, enzovoort.

Om het programma later eventueel te veranderen met een groter of kleiner aantal, is consequent de variabele `aantal` gebruikt, in plaats van de constante 10. Om er zeker van te zijn dat deze variabele niet (per ongeluk) veranderd wordt, is bij de declaratie aangegeven dat de waarde `final` is, dat wil zeggen: niet veranderd mag worden. Daarom ook moet de waarde direct bij declaratie al toegekend worden.

Om de gebruiker niet op te zadelen met de vervelende taak elke keer weer de voorbeeldtalen in te tikken, zet het programma alvast een aantal voorbeelden neer. In het programma zijn deze talen met bijbehorende urls in een array neergezet, zodat ze in de for-opdracht waar de tekstvelden worden gecreëerd, meteen van een opschrift kunnen worden voorzien. Deze array wordt bij declaratie meteen van een waarde voorzien:

```
string[] defaultwaarden =
{ "Nederlands", "http://nl.wikipedia.org"
, "Engels",      "http://en.wikipedia.org"
, "Duits",       "http://de.wikipedia.org"
, "Frans",       "http://fr.wikipedia.org"
, "Spaans",      "http://es.wikipedia.org"
, "Italiaans",   "http://it.wikipedia.org"
};
```

Exceptions niet afhandelen

In de methode `Lees` worden allerlei handelingen met files verricht, zonder dat deze opdrachten in een try-catch-opdracht staan. Normaal gesproken is dat gevaarlijk, want het programma zou worden afgebroken als er een exception optreedt. Hier is het echter geen probleem, omdat de *aanroep* van `Lees` in een try-catch opdracht staat. Dat is in dit programma namelijk een betere plaats om de exception zinvol af te handelen.

Invoer van WWW

Lezen van hypertext op het WWW gebeurt op vrijwel dezelfde manier als lezen van lokale files. De kunst is om aan de benodigde `TextReader` te komen. Dat gaat als volgt:

- maak een nieuw `WebClient` object
- roep van dat object de methode `OpenRead` aan, waarbij je het webadres als string meegeeft; dit levert een `Stream`-object op
- geef dat object mee aan de constructor van `StreamReader`

Daarmee kun je de webpagina gewoon lezen alsof het een lokale file is. In de methode `Lees` gebruiken we dit om van het web te lezen als de naam begint met `http://`. Anders wordt een lokale file gelezen.

Herdefinitie van geërfde methoden

In WWW-pagina's komt vaak veel Engelse tekst voor in de vorm van tags en filenamen. Die kan de letterfrequentie van de voorbeeldtalen lelijk verstoren. Daarom bouwen we een voorziening in dat alles tussen `<` en `>` niet wordt meegeteld, zodat tags en hun parameters buiten beschouwing blijven. Hiertoe herdefiniëren (*overriden*) we de methode `turf` uit `TurfTab`, die één character turf. In de herdefinitie wordt gecontroleerd of dit zo'n punt-haakje is. Zo ja, dan wordt een bool variabele veranderd die aangeeft of letters vanaf nu wel of niet meegeteld moeten worden. Als het character geen punthaakje is, wordt de oorspronkelijke methode `turf` (die bereikbaar is via de pseudo-variabele `base`) alleen maar aangeroepen als de bool variabele aangeeft dat we niet met een tag bezig zijn.

De boolean variabele `teltmee` is, doordat hij als member-variabele is gedeclareerd, een onderdeel geworden van de toestand van de turftabel. In de constructormethode krijgt deze zijn initiële waarde.

De hergedefinieerde methode `turf` wordt aangeroepen vanuit de klasse `TurfTab`. Zoals altijd bij herdefinities, hebben we daarmee die oorspronkelijke klasse mooi te pakken: in plaats van zijn eigen `turf`-een-character methode, roept die nu de hergedefinieerde versie aan!

```
using System;
using System.Drawing;
using System.Windows.Forms;

5 namespace Taalherkenning
{
    public class Taal : Form
    {
        const int aantal = 10;
10     TextBox invoer;
        Label uitvoer;
        TextBox[] taal;
        Label [] score;
        TextBox[] url;
15     TableLayoutPanel alles;

        void klik(object o, EventArgs ea)
        {
            RelTurfTab onbekend, voorbeeld;
20     double kleinste, verschil;
            String naam, antwoord;

            onbekend = new RelTurfTab();
            onbekend.Turf(invoer.Text);

25     kleinste = 1.0;
            antwoord = "onbekend";
            for (int t = 0; t < aantal; t++)
            {
                naam = url[t].Text;
30     if (naam != "")
                {
                    voorbeeld = new RelTurfTab();
                    try
                    {
                        voorbeeld.Lees(naam);
                        verschil = onbekend.Verschil(voorbeeld);
35     score[t].Text = ((int)(10000 * verschil)).ToString();

                        if (verschil < kleinste)
                        {
                            kleinste = verschil;
                            antwoord = taal[t].Text;
40     }
                        }
                    catch (Exception)
                    {
                        this.score[t].Text = "???";
                    }
45     }
                }
            }
            uitvoer.Text = antwoord;
        }
        void vergroot(object o, EventArgs ea)
50     {
            alles.Size = this.ClientSize;
        }
    }
}
```

Listing 56: Taalherkenning/Taal.cs, deel 1 van 2

```
public Taal(string[] startwaarde)
55 {
    Button knop;
    TableLayoutPanel paneel, talen;

    this.Text = "Taalherkenning";
60 this.Size = new Size(800, 300);
    this.Resize += vergroot;
    invoer = new TextBox(); invoer.Multiline = true;
    invoer.Font = new Font("Tahoma", 14);
    knop = new Button(); knop.Text = "Herken taal";
65 knop.Font = new Font("Tahoma", 10);
    knop.Click += this.klik;
    uitvoer = new Label();
    uitvoer.Text = "Hallo";
    uitvoer.Dock = DockStyle.Fill;
70 uitvoer.Font = new Font("Tahoma", 14);

    paneel = new TableLayoutPanel(); paneel.ColumnCount = 2;
    talen = new TableLayoutPanel(); talen.ColumnCount = 3;
    alles = new TableLayoutPanel(); alles.ColumnCount = 2;

75 taal = new TextBox[aantal];
    url = new TextBox[aantal];
    score = new Label[aantal];
    for (int t = 0; t < aantal; t++)
80 {
        taal[t] = new TextBox(); taal[t].Dock = DockStyle.Fill;
        score[t] = new Label(); score[t].Dock = DockStyle.Fill;
        url[t] = new TextBox(); url[t].Dock = DockStyle.Fill;
        score[t].TextAlign = ContentAlignment.MiddleRight;
85 if (t < startwaarde.Length / 2)
        { taal[t].Text = startwaarde[2 * t];
          url[t].Text = startwaarde[2 * t + 1];
        }
        talen.Controls.Add(score[t]);
90 talen.Controls.Add(taal[t]);
        talen.Controls.Add(url[t]);
    }
    paneel.Controls.Add(invoer); paneel.SetColumnSpan(invoer, 2);
    paneel.Controls.Add(knop);
95 paneel.Controls.Add(uitvoer);
    alles.Controls.Add(paneel);
    alles.Controls.Add(talen);
    this.Controls.Add(alles);
    this.vergroot(null, null);

100 // en nog wat regels om de kolombreedte van de TableLayoutPanel te zetten...
```

Listing 57: Taalherkenning/Taal.cs, deel 2 van 2

```

using System;
using System.IO;
using System.Net;
using LetterTeller;

5
namespace Taalherkenning
{
    class RelTurfTab : TurfTab
    {
10        bool teltmee = true;

        public RelTurfTab()
        {
        }

15        private double relatief(int i)
        { return ((double)this.tellers[i]) / this.totaal;
        }

20        public double Verschil(RelTurfTab andere)
        { if (this.totaal > 0)
          {
              double totaal = 0.0;
              for (int t = 0; t < 26; t++)
25                  totaal += Math.Abs(this.relatief(t) - andere.relatief(t));
              return totaal/26;
          }
          else return 1.0;
        }

30        public void Lees(string naam)
        { TextReader reader;
          String regel;

35          if (naam.StartsWith("http://"))
              reader = new StreamReader(new WebClient().OpenRead(naam));
          else reader = new StreamReader(naam);
          while ((regel = reader.ReadLine()) != null)
              this.Turf(regel);

40        }
        protected override void turf(char c)
        { if (c == '<') teltmee = false;
          else if (c == '>') teltmee = true;
          else if (teltmee) base.turf(c);

45        }
    }
}

```

Listing 58: Taalherkenning/RelTurfTab.cs

```
using System;
using System.Windows.Forms;

namespace Taalherkenning
5 {
    static class Program
    {
        static void Main(string[] args)
        {
10         string[] defaultwaarden =
            { "Nederlands", "http://nl.wikipedia.org"
              , "Engels",    "http://en.wikipedia.org"
              , "Duits",    "http://de.wikipedia.org"
              , "Frans",    "http://fr.wikipedia.org"
15         , "Spaans",    "http://es.wikipedia.org"
              , "Italiaans", "http://it.wikipedia.org"
            };

            if (args.Length == 0)
20         args = defaultwaarden;

            Application.Run(new Taal(args));
        }
    }
25 }
```

Listing 59: Taalherkenning/Program.cs

11.3 Zoeken in een netwerk

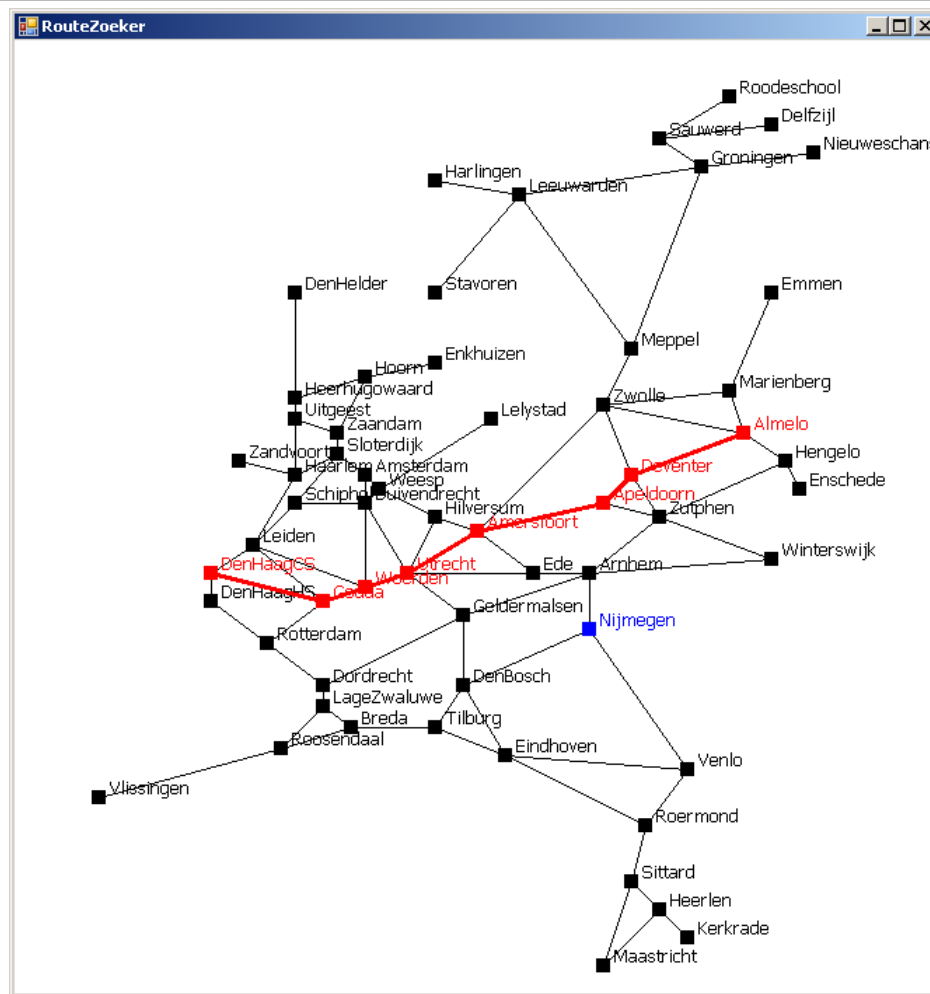
Beschrijving van de casus

Het programma in deze sectie heet 'RouteZoeker'. Het maakt het mogelijk dat de gebruiker op een landkaart met wegen (of spoorlijnen, of telecommunicatieverbindingen) de snelste (of goedkoopste of anderszins beste) route tussen twee plaatsen kan vinden.

De plaatsen van het netwerk en de verbindingen daartussen leest het programma uit een file. Daarna wordt aan de gebruiker een grafische presentatie hiervan aangeboden. De gebruiker mag vervolgens twee plaatsen aanklikken, waarop de beste route tussen die plaatsen in een andere kleur wordt weergegeven. Daarna mag de gebruiker opnieuw twee plaatsen aanklikken.

In figuur 41 is het programma in werking te zien. Als netwerk zijn de belangrijkste spoorwegverbindingen in Nederland ingelezen. Zojuist is de beste route van Den Haag naar Almelo gevonden, en heeft de gebruiker Nijmegen alweer aangeklikt voor een nieuwe zoekopdracht.

De modellering van het probleem is tamelijk ingewikkeld. Zowel het opdelen van het programma in klassen, als het ontwerp van sommige methoden vergen zorgvuldige planning.



Figuur 41: Het programma RouteZoeker in werking

Opdeling in klassen

Omdat dit een interactief programma is, hebben we in ieder geval een subklasse van `Form` nodig. Deze klasse `RouteZoeker` verzorgt de interactie met de gebruiker: hier bevindt zich de event-handler van de muiskliks. Voor het eigenlijke model van het netwerk maken we een aparte klasse `Netwerk`. Op dezelfde manier als de klasse `Calculator` een membervariabele van het type `Processor` had, waarin de toestand van de rekenmachine was vastgelegd, zal `RouteZoeker` een

membervariabele van het type **Netwerk** krijgen. Alle dingen die te maken hebben met forms, muis-activiteit en dergelijke horen thuis in **RouteZoeker**; de opbouw van het netwerk uit steden en wegen, alsmede het zoekproces worden daarentegen gemodelleerd in de klasse **Netwerk**. Belangrijke onderdelen van een netwerk zijn de steden, en de verbindingswegen tussen de steden. Daarom maken we aparte klassen **Stad** en **Weg** die objecten van die types modelleren. Bij het zoeken is verder sprake van een route door het netwerk. Om objecten die zo'n route door het netwerk voorstellen te modelleren, maken we een aparte klasse **Pad**.

Klasse **RouteZoeker**: De grafische interface

De muiskliks van de gebruiker kunnen worden opgevangen door event-handler **klik**, en de grafische output kan getekend worden in de event-handler **teken**. Maar welke membervariabelen zijn er nodig?

Zoals hierboven al werd opgemerkt is er een **Netwerk**-object nodig, waarin het netwerk ligt opgeslagen. Verder zijn er objecten nodig waarmee de inhoud van het window op verzoek getekend kan worden. Daarvoor is nodig:

- Een tekening van het netwerk. Met deze klus kunnen we de **Netwerk** membervariabele opzaden, door daarin een methode te maken die zichzelf tekent op een als parameter meegegeven **Graphics**-object.
- Een tekening van de route in een andere kleur. Er is dus een **Pad**-object nodig, die ook in staat moet zijn om zichzelf te tekenen.

Opmerkelijk aan de methode **klik** is dat de eerste en de tweede klik verschillend behandeld moeten worden. De eerste klik moet alleen maar bewaard worden: 'oh, dit is blijkbaar de stad waar de route moet beginnen'. Pas na de tweede klik kan het zoekproces beginnen.

Er is dus een membervariabele nodig waarin de begin-stad bewaard kan worden. Aan het wel of niet geïnitieerd zijn van deze variabele kan **klik** ook zien of het de tweede of de eerste klik betreft.

Een eerste opzet van de klasse **NetApplet** is al met al als volgt:

```
class RouteZoeker : Form
{
    Netwerk netwerk;           // modellering van het netwerk
    Pad    pad;                // de gevonden route door het netwerk
    Stad    stad1;              // eerste aangeklikte stad
    RouteZoeker() {...}        // initialisatie
    void teken(...) {...}      // laat netwerk en pad zichzelf tekenen
    void klik (...) {...}      // eerste keer: bewaar stad1; tweede keer: zoek pad
}
```

Klasse **Netwerk**: de modellering van een netwerk

Een netwerk bestaat uit een aantal steden en wegen daartussen. Een voor de hand liggende keuze voor de membervariabelen van een netwerk is een collectie van **Stad**-objecten en een collectie van **Weg**-objecten. Als we er echter voor zorgen dat elke stad zijn eigen uitgaande wegen bewaart, hoeven de wegen niet eens apart in het netwerk bewaard te worden, en bestaat een netwerk dus alleen maar uit een collectie van steden.

Een van de methoden van **Netwerk** hebben we al beloofd: een netwerk-object moet zichzelf kunnen tekenen op een mee te geven **Graphics**-object. Ook moet een netwerk zichzelf kunnen opbouwen met de informatie uit een file. Verder komt er natuurlijk een methode voor de centrale vraag: het zoeken van een pad tussen twee steden.

Tijdens het opbouwen van het netwerk is het handig om aparte methoden te maken die een enkele stad, respectievelijk een enkele weg aan het netwerk toevoegen. De gebruiker klikt punten aan, geen steden. Er zal dus een vertalingsmethode moeten komen die aangeeft of er een stad ligt op de aangegeven plaats, en zo ja, welke. Dit is ook een mooie taak voor het netwerk.

Het ontwerp voor de klasse **Netwerk** wordt daarmee als volgt:

```
class Netwerk
{
    ICollection<Stad> steden;    // membervariabele
    Netwerk() {...}            // initialisatie
    void bouwStad(...) {...}
    void bouwWeg(...) {...}
    Stad VindStad(Point) {...}  // ligt er een stad op dit punt?
    Pad ZoekPad(Stad, Stad) {...} // zoek route tussen twee steden
}
```

```

void Lees(String)      {...}    // bouw netwerk volgens data in file
void Teken(Graphics)  {...}    // teken jezelf
}

```

Klassen Stad en Weg: de onderdelen van het netwerk

Een stad heeft een naam en een positie op de kaart. Verder hadden we beloofd bij een stad de uitgaande wegen op te slaan. Het is handig als een stad zichzelf op een Graphics-object kan tekenen, zodat Netwerk dat werk kan uitbesteden. Omdat een stad zijn uitgaande wegen beheert, moet er ook een methode zijn om een nieuwe weg te bouwen.

Het ontwerp van de klasse Stad is dus op z'n minst:

```

class Stad
{
    String Naam;
    Point Plek;
    ICollection<Weg> Wegen;
    Stad(...) {...}
    void BouwWeg(...) {...}
    void Teken(Graphics) {...}
}

```

Als we straks naar de goedkoopste route tussen twee steden willen gaan zoeken, dan moet het wel bekend zijn wat het gebruik van een weg tussen twee steden kost. Verder is het van een weg van belang waar hij naartoe loopt. Een weg wordt dus gemodelleerd door:

```

class Weg
{
    Stad Doel;
    int Kosten;
    Weg(...) {...}
    void Teken(Graphics) {...}
}

```

Klasse Pad: een route door het netwerk

Een pad lijkt op het eerste gezicht te bestaan uit een aantal aan elkaar gekoppelde wegen, bijvoorbeeld opgeslagen in een list. Het zal echter handiger blijken om een pad te representeren door de steden waar het pad langs loopt. Een pad zou dus een list van steden kunnen zijn. Maar een andere leuke representatie is (zie ook figuur 42):

```

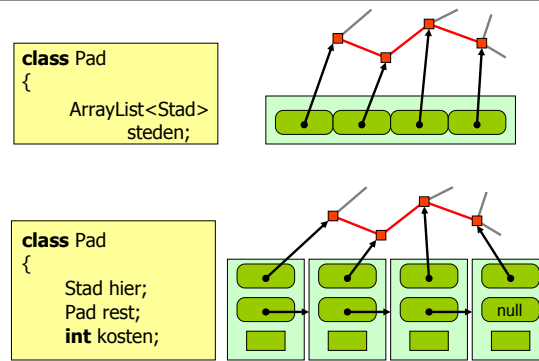
class Pad
{
    // IList<Stad> steden
    // vervangen door een eigen implementatie:
    Stad Hier;
    Pad Rest;
    int Kosten;
    Pad(...) {...}
    bool Bevat(Stad) {...}
    void Teken(Graphics) {...}
}

```

Een pad wordt dus voorgesteld door de stad waar hij begint, en daarbij weer een pad-object waarin de rest van het pad is opgeslagen. Daarin bevindt zich dan de volgende stad, en opnieuw een pad-object waarin het dan nog resterende deel van het pad ligt. In de laatste schakel van het pad laten we eenvoudigweg de membervariabele `rest` ongeïnitieerd. Behalve een constructor- en een teken-methode is er een methode waarmee we kunnen kijken of een bepaalde stad op het pad ligt.

Het zoekalgoritme

Het zoeken van een pad tussen twee steden is een lastige aangelegenheid. De essentie van deze methode is dat we een verzameling van veelbelovende, maar nog niet complete paden gaan aanleggen. Die paden gaan we vervolgens proberen uit te breiden tot paden die het doel nog beter benaderen. Dat alles net zo lang totdat we een pad tegenkomen dat op de gewenste bestemming eindigt. De verzameling paden kan worden geïnitieerd met een zeer kort pad: het pad met lengte 0 dat in de begin-stad begint én eindigt. Dat is weliswaar nog geen geweldig goed pad, maar door



Figuur 42: Twee mogelijke representaties van een Pad

het maar gestaag uit te breiden komen we vanzelf dichterbij het doel. Een eerste opzet van de methode `zoekPad` is:

```

Pad zoekPad(Stad van, Stad naar)
{
    paden.Push(beginpad);    // voeg een pad van 'van' naar 'van' toe aan de paden;
    while (...)              // zolang er nog veelbelovende paden zijn
    {
        pad = ...;           // pak een pad uit de verzameling
        if (pad.Hier==naar)   // eindigt het pad in de bestemming?
            return pad;       // gevonden!
        for (...)             // voor alle 'hier' vertrekkende wegen
            paden.Push(...);  // voeg een pad toe aan de verzameling
    }
    return null;              // helaas, niets gevonden
}

```

Zoeken: niet in een kringetje lopen

Door de paden aldoor maar uit te breiden lopen we het gevaar dat een pad weer uitkomt bij het vertrekpunt, zonder dat de bestemming is bereikt. Op die manier blijf je steeds in kringetjes lopen, zonder ooit op de bestemming te geraken. We moeten daarom in de methode een nieuw pad alleen aan de verzameling toevoegen als er geen lus in zit, dat wil zeggen als de nieuwe stad nog niet op het pad voorkomt. De methode `Bevat` in de klasse `Pad` komt daarbij goed van pas.

Zoeken: niet de eerste, maar de beste

Bovenstaande opzet van de methode stopt direct met zoeken zodra er een pad gevonden is. Dat wil nog niet zeggen dat dit ook het beste pad is dat er bestaat.

Daarom moeten we verder zoeken, in de hoop nog een beter pad te vinden. We kunnen hiertoe het algoritme combineren met het patroon 'zoek de grootste'. We krijgen dan:

```

Pad zoekPad(Stad van, Stad naar)
{
    paden.Push(beginpad);
    beste = null;              // voorlopig nog niets gevonden
    while (...)
    {
        pad = ...;
        if (pad.Hier==naar)
        {
            if (...)           // nieuwe oplossing beter dan de beste?
                beste = pad;    // dan is dit de nieuwe 'beste'
        }
        for (...)
            if (...)           // geen kringetje?
                paden.Push(...);
    }
    return beste;              // retourneer de beste, of 'niets'
}

```

Sneller zoeken: geen hopeloze gevallen

Als we eenmaal een oplossing hebben gevonden, blijven we nu dus verder zoeken in de hoop een betere oplossing te vinden. Maar het heeft natuurlijk geen zin om nog niet-complete oplossingen te blijven exploreren die nu al duurder zijn dan een eerder gevonden oplossingen. Die kunnen dus, net als de circulaire paden, van het zoekproces worden uitgesloten.

Sneller zoeken: veelbelovendste eerst

Door de truc hierboven gaat het zoeken dus aanzienlijk sneller zodra er een eerste oplossing gevonden is. Het is dus zaak zo snel mogelijk alvast een oplossing te vinden. Een manier om dit te bespoedigen is om als eerste te zoeken in een richting die veelbelovend lijkt.

We kennen tenslotte de coördinaten van de steden, en als de ‘road prizing’ een beetje logisch is opgebouwd dan zijn wegen die de verkeerde kant op gaan een goede manier om de reis duur te maken... Wil je van Utrecht naar Groningen reizen, dan kun je het beste eerst Amersfoort of desnoods Hilversum proberen, en niet Geldermalsen!

Dit gaat helaas niet altijd goed. Van Hoorn naar Leeuwarden reis je niet via Enkhuizen, hoewel dat op het eerste gezicht de goede kant op lijkt te gaan. Maar geen nood: kwaad kan het ook niet om de wegen in een bepaalde volgorde te proberen. Tot nu toe hadden we ze tenslotte in een willekeurige volgorde geprobeerd. (Een poging tot versnelling van een algoritme die in veel gevallen, maar niet altijd, goed werkt, maar verder ook geen kwaad kan, heet een ‘heuristiek’).

Klasse Stack: een stapel objecten

In de namespace `System.Collections.Generic` staat nog een bijzondere implementatie van `ICollection`: de klasse `Stack`. Deze klasse modelleert een ‘stapel’ objecten, waarbovenop nieuwe elementen kunnen worden toegevoegd, en waar elementen alleen maar van bovenop de stapel kunnen worden teruggepak (slecht idee om je post op deze manier te behandelen). Deze klasse kent naast de property `Count` een paar extra methoden, waaronder:

- `void Push<Elem>(Elem x)` voegt een element toe bovenop de stapel
- `Elem pop<Elem>()` verwijdert het element bovenop de stapel en levert dat op

Uitwerking van het programma

Het routezoek-programma staat in de hiernavolgende klassen.

- Netwerk, met als taken:
 - het opbouwen van het netwerk
 - het tekenen van het netwerk (zie listing 64);
 - het inlezen van het netwerk
 - het zoekalgoritme (zie listing 65).
- `RouteZoeker`, voor de grafische userinterface (zie listing 60);
- `Stad`, voor de modellering van een stad (zie listing 61);
- `Weg`, voor de modellering van een weg tussen twee aangrenzende steden (zie listing 62);
- `Pad`, voor de modellering van een route tussen twee steden (zie listing 63);
- `Program`, met een functie `Main` die een `RouteZoeker` opstart (listing weggelaten).

blz. 206

blz. 207

blz. 203

blz. 204

blz. 205

blz. 205

```
using System.Drawing;
using System.Windows.Forms;

namespace RouteZoeker
5 {
    public class RouteZoeker : Form
    {
        private Network network;
        private Pad pad;
10     private Stad stad1;

        public RouteZoeker()
        {
            this.ClientSize = new Size(660, 680);
15     this.Text = "RouteZoeker";
            this.BackColor = Color.White;
            this.MouseClick += klik;
            this.Paint += teken;

20     network = new Network();
            network.Lees("../..//Spoor.txt");
        }

        private void klik(object o, MouseEventArgs mea)
25     {
            Stad s = network.VindStad(mea.Location);
            if (s != null)
            {
                if (stad1 == null)
30                 stad1 = s;
                else
                {
                    pad = network.ZoekPad(stad1, s, true);
                    stad1 = null;
35                 }
                this.Invalidate();
            }
        }

40     private void teken(object o, PaintEventArgs pea)
        {
            Graphics gr = pea.Graphics;
            network.Teken(gr);
            if (pad != null) pad.Teken(gr, Brushes.Red, new Pen(Brushes.Red,3));
45     if (stad1 != null) stad1.Teken(gr, Brushes.Blue);
        }
    }
}
```

Listing 60: RouteZoeker/RouteZoeker.cs

```

using System.Collections.Generic;
using System.Drawing;

namespace RouteZoeker
5 {
    class Stad : IComparer<Weg>
    {
        public string Naam;
        public ICollection<Weg> Wegen;
10     public Point Plek;
        private static Font font = new Font("Tahoma", 10);

        public Stad(string naam, Point plek)
        {
15             this.Naam = naam;
            this.Plek = plek;
            this.Wegen = new LinkedList<Weg>();
        }
        public void BouwWeg(Stad doel, int kosten)
20     {
            this.Wegen.Add(new Weg(doel, kosten));
        }
        public void Teken(Graphics gr, Brush brush)
        {
25             gr.FillRectangle(brush, new Rectangle(this.Plek-new Size(5,5), new Size(10,10)));
            gr.DrawString(this.Naam, font, brush, this.Plek+new Size(6,-15));
        }

        // Implementatie van IComparer:
30     // orden twee wegen in *dalende* volgorde
        // wat betreft de hemelsbrede afstand van hun doel tot deze Stad

        public int Compare(Weg a, Weg b)
        {
35             return this.afstand(b.Doel) - this.afstand(a.Doel);
        }
        int afstand(Stad a)
        {
            return Stad.afstand(a.Plek, this.Plek);
40     }
        static int afstand(Point a, Point b)
        {
            return kwadraat(a.X - b.X) + kwadraat(a.Y + b.Y);
            // Pythagoras zegt dat we ook nog de wortel moeten trekken,
45         // maar dat doen we lekker niet, omdat het alleen om de ordening
            // van de afstanden gaat. Die verandert niet door het worteltrekken.
        }
        static int kwadraat(int x)
        {
50             return x * x;
        }
    }
}

```

Listing 61: RouteZoeker/Stad.cs

```
using System.Drawing;

namespace RouteZoeker
{
5   class Weg
    {
        public Stad Doel;
        public int Kosten;

10        public Weg(Stad doel, int kosten)
        {   this.Doel = doel;
            this.Kosten = kosten;
        }
        public void Teken(Graphics gr, Pen pen, Stad stad)
15        {   gr.DrawLine(pen, stad.Plek, this.Doel.Plek);
        }
    }
}
```

Listing 62: RouteZoeker/Weg.cs

```
using System.Drawing;

namespace RouteZoeker
{
5   class Pad
    {
        public Stad Hier;
        public Pad Rest;
        public int Kosten;

10        public Pad(Stad hier, Pad rest, int k)
        {   this.Hier = hier;
            this.Rest = rest;
            this.Kosten = k;
15            if (rest != null)
                this.Kosten += rest.Kosten;
        }
        public bool Bevat(Stad s)
        {   if (this.Hier == s) return true;
20            if (this.Rest == null) return false;
            return Rest.Bevat(s);
        }
        public void Teken(Graphics gr, Brush brush, Pen pen)
        {   this.Hier.Teken(gr, brush);
25            if (this.Rest != null)
            {   gr.DrawLine(pen, Hier.Plek, Rest.Hier.Plek);
                this.Rest.Teken(gr, brush, pen);
            }
        }
30    }
}
```

Listing 63: RouteZoeker/Pad.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
5
namespace RouteZoeker
{
    class Netwerk
    {
10        public ICollection<Stad> Steden;

        public Netwerk()
        {
            Steden = new LinkedList<Stad>();
15        }
        public void Teken(Graphics gr)
        {
            foreach (Stad stad in this.Steden)
            {
20                stad.Teken(gr, Brushes.Black);
                foreach (Weg weg in stad.Wegen)
                    weg.Teken(gr, Pens.Black, stad);
            }
        }
25        private void bouwStad(string naam, Point plek)
        {
            Steden.Add(new Stad(naam, plek));
        }
        private void bouwWeg(string naam1, string naam2, int prijs)
30        {
            Stad stad1 = this.vindStad(naam1);
            Stad stad2 = this.vindStad(naam2);
            stad1.BouwWeg(stad2, prijs);
            stad2.BouwWeg(stad1, prijs);
35        }
        private Stad vindStad(string naam)
        {
            foreach (Stad stad in this.Steden)
            {
40                if (stad.Naam == naam)
                    return stad;
            }
            return null;
        }
45        public Stad VindStad(Point p)
        {
            foreach (Stad stad in this.Steden)
            {
                if (Math.Abs(p.X - stad.Plek.X) < 5 && Math.Abs(p.Y - stad.Plek.Y) < 5)
50                return stad;
            }
            return null;
        }
    }
}
```

```

55     public void Lees(string filenaam)
    {
        int nr = 0;
        StreamReader sr = new StreamReader(filenaam);
        string regel;
        string[] woorden;
60     char[] separators = { ' ' };
        while ((regel = sr.ReadLine()) != null)
        {
            nr++;
            woorden = regel.Split(separators, StringSplitOptions.RemoveEmptyEntries);
65     if (woorden.Length == 4)
            {
                if (woorden[0] == "stad")
                    this.bouwStad( woorden[1]
                                , new Point( int.Parse(woorden[2])
                                , int.Parse(woorden[3])));
70     else if (woorden[0] == "weg")
                    this.bouwWeg(woorden[1], woorden[2], int.Parse(woorden[3]));
            }
        }
75 }
    public Pad ZoekPad(Stad van, Stad naar, bool slim)
    {
        Stack<Pad> paden = new Stack<Pad>();
        Pad beste = null;
80
        paden.Push(new Pad(van, null, 0));

        while (paden.Count > 0)
        {
85     Pad pad = paden.Pop();
            if (pad.Hier == naar)
            {
                if (beste == null || pad.Kosten < beste.Kosten)
                    beste = pad;
90     }
            ICollection<Weg> wegen;
            if (slim)
                wegen = new SortedSet<Weg>(pad.Hier.Wegen, naar);
            else wegen = pad.Hier.Wegen;
95
            foreach (Weg weg in wegen)
            {
                if ( !pad.Bevat(weg.Doel)
                    && (beste==null || pad.Kosten+weg.Kosten<=beste.Kosten)
100    )
                    paden.Push( new Pad(weg.Doel, pad, weg.Kosten) );
            }
        }
        return beste;
105    }
}

```

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. 209 De programmatekst staat in listing 66. Een voorbeeld van hoe deze klasse in een programma
blz. 210 gebruikt zou kunnen worden staat in listing 67.

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
    {
5        public byte Rood, Groen, Blauw;
        public static byte Maximaal = 255;

        public Kleur()
        {
10            Rood = Maximaal; Groen = Maximaal; Blauw = Maximaal;
        }
        public Kleur(byte x)
        {
            Rood = x; Groen = x; Blauw = x;
15        }
        public Kleur(byte r, byte g, byte b)
        {
            Rood = r; Groen = g; Blauw = b;
        }
20        public Kleur(Kleur orig)
        {
            Rood = orig.Rood; Groen = orig.Groen; Blauw = orig.Blauw;
        }
        public Kleur(string s)
25        {
            string[] velden = s.Split(' ');
            Rood = byte.Parse(velden[0]);
            Groen = byte.Parse(velden[1]);
            Blauw = byte.Parse(velden[2]);
30        }
        public override string ToString()
        {
            return $"{Rood} {Groen} {Blauw}";
        }
35        public byte Grijswaarde()
        {
            return (byte)(0.3 * Rood + 0.6 * Groen + 0.1 * Blauw);
        }
        public void MaakDonkerder()
40        {
            Rood = (byte)(Rood * 0.9);
            Groen = (byte)(Groen * 0.9);
            Blauw = (byte)(Blauw * 0.9);
        }
        public Kleur DonkerdereVersie()
45        {
            Kleur res = new Kleur(this);
            res.MaakDonkerder();
            return res;
        }
        public static Kleur Zwart = new Kleur(0, 0, 0);
50        public static Kleur Geel = new Kleur(Maximaal, Maximaal, 0);

        public static Kleur Parse(string s)
        {
            return new Kleur(s);
55        }
    }
}
```

```
using System;

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

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

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

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

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

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

30         }
    }
}
```

Listing 67: 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.Blaauw = 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; Blaauw = 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 zoveel 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; Blaauw = 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; Blaauw = 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 `Blaauw` 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 * Blaauw);
}
```

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**.

Bijlage A

Gebruik van Visual Studio

Deze appendix beschrijft het gebruik van de C#-compiler van Microsoft, met gebruikmaking van de geïntegreerde ontwikkelomgeving (IDE) *Visual Studio*.

A.1 Installatie van de software

Als je de software op een eigen computer wilt gebruiken, moet je deze eerst installeren. Op de practicumcomputers is dit niet nodig; daar is de software al geïnstalleerd. Het installeren op een eigen computer gaat als volgt:

Download Visual Studio

Ga naar <https://www.visualstudio.com/downloads>. Download de versie *Visual Studio Community*. De recente versie is 2017, maar de voorbeelden uit dit diktaat werken ook met versie 2015 en 2013.

Installeer Visual Studio

Start het installatieprogramma dat je hebt gedownload. Tijdens de installatie moet je aankruisen dat je *.NET desktop development* nodig hebt. Er is 15 GB diskruimte nodig, en installatie kan een kwartier duren.

A.2 De eerste keer Visual Studio starten

Start Visual Studio. Op de practicumcomputers kun je het vinden onder menukeuze **Start**→**All Programs**→**Course software**→**Informatica**→**Microsoft Visual Studio**→**Microsoft Visual Studio 2017**. Als je Visual Studio voor het eerst opstart, dan zal het programma vragen wat je standaardontwikkeltaal is. Kies hier voor ‘Visual C# Development Settings’. Deze optie zal de ontwikkelomgeving optimaal instellen voor het gebruik van de taal C#. Zodra dit gedaan is, zie je het opstartscherm van Visual Studio (zie figuur 43).

A.3 De werking van de Visual Studio IDE

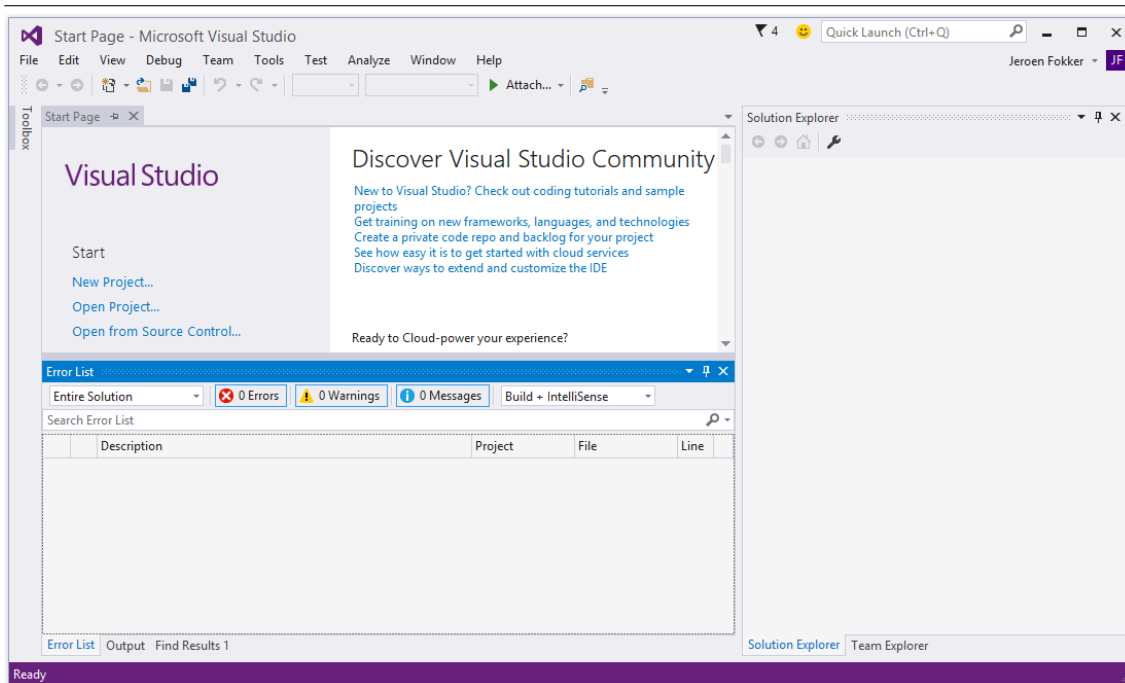
Projecten en Solutions

Visual Studio werkt met zogeheten *projects* en *solutions*. Alle bronbestanden (met extensie **.cs**) staan over het algemeen in een gemeenschappelijke directory. Daarin staat ook een bestand met de extensie **.csproj** waarin gedocumenteerd staat welke bronbestanden tot het project behoren, plus enkele gekozen opties. Het **.csproj**-bestand bevat dus alleen de administratie van het project; de eigenlijke code staat in de **.cs**-bestanden.

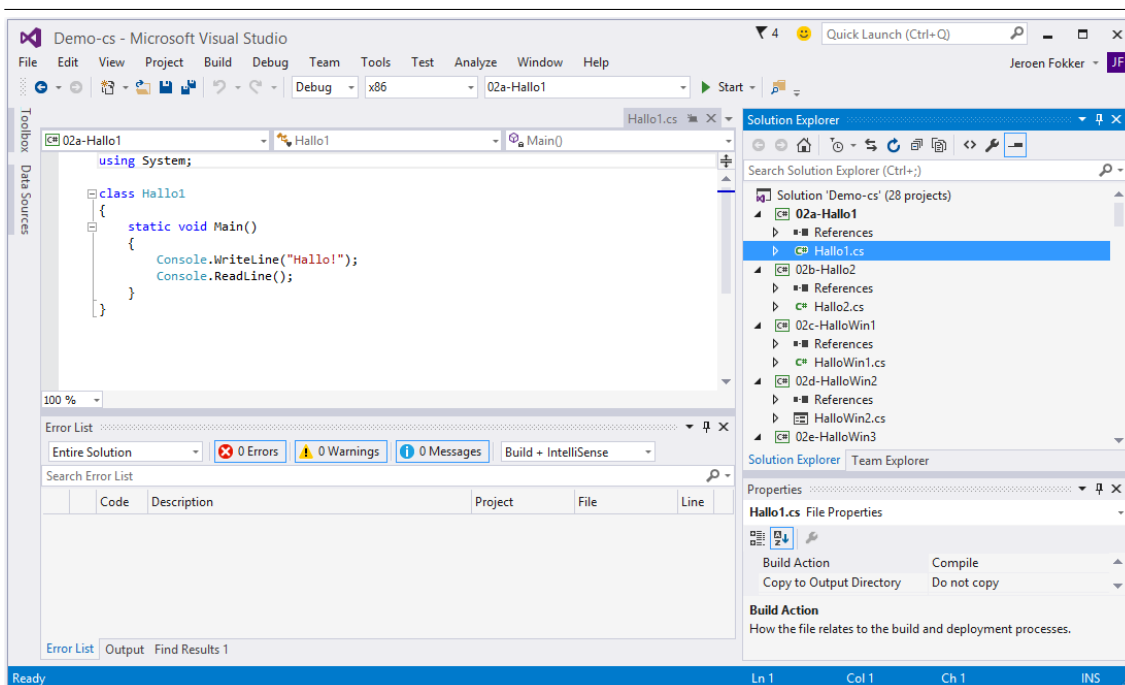
Een *solution* is een bundeling van projecten. Den bestand met de extensie **.sln** bevat de administratie van welke projecten er zijn gebundeld.

De voorbeeldprogramma's uit dit diktaat kun je downloaden. Elk programma is een apart project, en staat in een eigen directory. Alle project-directories samen staan in de directory ‘Demo-cs’, en hierin staat ook een solution die een index vormt op alle projecten. Download de voorbeelden als **.zip**-file. Vergeet niet om de zip-file uit te pakken; je kunt met Visual Studio niet de bestanden direct vanuit de zip-file manipuleren.

In figuur 44 zie je de ontwikkelomgeving na het openen van de solution **Demo-cs.sln**. Een overzicht van de projecten in de solution kun je vinden aan de rechterkant van het scherm in de *solution explorer*.



Figuur 43: Het startscherm na het opstarten van Visual Studio.



Figuur 44: Visual Studio met Solution explorer (rechtsboven), Error list (linksonder) en edit-window (linksboven).

Je ziet ook dat één van de projectnamen vetgedrukt staat. Dit is het huidige ‘actieve’ project. Als je in Visual Studio menukeuze **Debug→Start Debugging** maakt, op de F5-toets drukt, dan wordt het actieve project uitgevoerd, in dit geval het programma *Hallo1*. Door rechts te klikken op een project en dan ‘Set as StartUp Project’ te kiezen kun je een ander project actief maken.

Opbouw van een project

Als je een applicatie ontwikkelt, dan wordt die applicatie beheerd in een project. In een project staan een aantal verschillende bestanden. Je vindt onder andere de C# broncode-bestanden die gebruikt worden, referenties naar de bibliotheken die gebruikt worden (bijvoorbeeld **System** en **System.Windows.Forms**), en eventuele andere hulpbestanden. Klik zo nodig op het pijltje voor de projectnaam om de inhoud ervan uit te klappen.

Door te dubbelklikken op een bestand kun je het bestand bekijken en/of aanpassen. In figuur 44 is dit gebeurd met het bestand *Hallo1.cs*. Je kunt nu de programmacode aanpassen. Ook zie je dat Visual Studio automatisch verschillende kleuren geeft aan woorden van de tekst, zodat je makkelijk keywords, klassen, methoden, en andere basiselementen van de taal C# kunt herkennen. Je kunt meerdere (programma-)tekstbestanden tegelijk openen; de windows voor de verschillende bestanden komen als ‘tabbladen’ over elkaar heen te liggen.

Als je een broncode-bestand opent waarin een subklasse van **Form** wordt gedefinieerd, dan zal het bestand in eerste instantie in ‘Visual Design’ mode worden geopend. Dit is bijvoorbeeld het geval bij *Mixer.cs*. Je krijgt dan een impressie van de userinterface en kunt die eventueel interactief veranderen. Met de F7-toets kun je alsnog de broncode zelf te zien krijgen, en met Shift-F7 switch je weer terug naar Visual Design.

Uitvoeren van een programma

Om een programma uit te voeren, moet het programma eerst gecompileerd worden. Dit kun je doen door op het groene pijltje ‘Start’ te klikken in de knoppenbalk boven de editor, via menukeuze **Debug→Start Debugging**, of door de F5-toets in te drukken. Daarmee wordt het huidige actieve project eerst gecompileerd, en als het foutvrij is meteen ook uitgevoerd.

De compiler ontdekt tijdens het compileren of er fouten in de code staan (syntactische fouten of fouten in de typering, maar géén semantische fouten!) en geeft dit aan in de *error list*, die meestal linkonder op het scherm staat.

Voordat het programma uitgevoerd kan worden, moeten eerst alle fouten in het programma verbeterd zijn. Als er wel fouten zijn biedt de IDE aan om de vorige versie van het programma uit te voeren, maar dat is weinig zinvol.

Soms geeft de compiler ook ‘waarschuwingen’. Dit zijn ‘fouten die wat minder erg zijn’, maar die je eigenlijk wel zou moeten oplossen om een stabiel programma op te leveren. Je kunt bijvoorbeeld een waarschuwing krijgen als je ergens een variabele declareert, maar die variabele verder nergens in je programma gebruikt. Dat is verdacht en kan de indicatie zijn van een tikfout, maar op zich is het niet verboden, dus het programma kan wel uitgevoerd worden.

Een nieuw project toevoegen

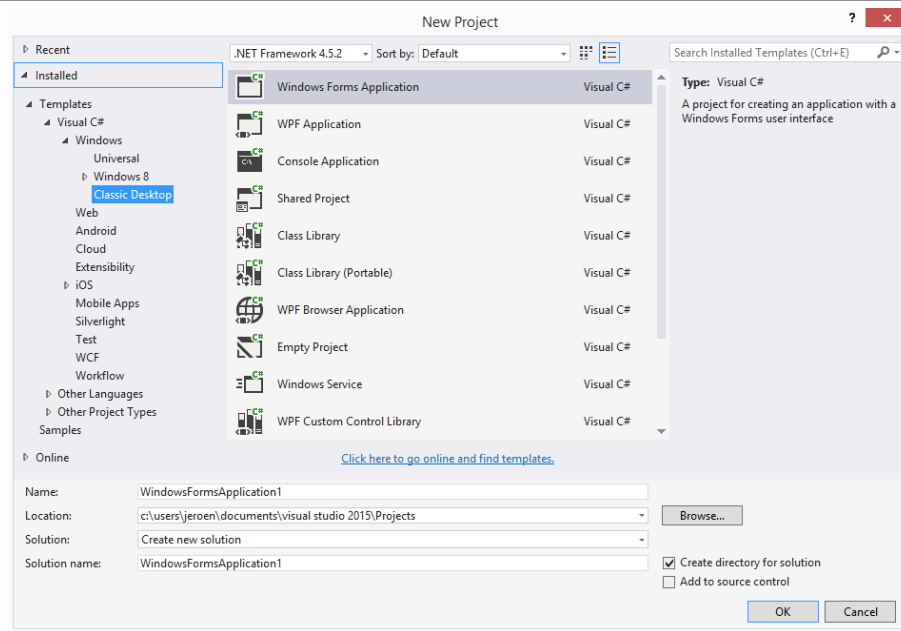
Als je een nieuwe applicatie/game schrijft in plaats van een bestaand programma uit te breiden, dan moet je een nieuw project aanmaken. Dit kun je doen via menukeuze **File→New→Project**. Je ziet dan een keuzeschermbild zoals afgebeeld in figuur 45.

Visual Studio kent een aantal *templates*, oftewel: vooraf ingevulde projecten voor bepaalde typen applicaties. De templates zijn naar categorie gegroepeerd; kies in de lijst categorieën in de linkerhelft van de dialoog voor menukeuze **Visual C#→Windows→Classic Desktop**. In de lijst rechts in de dialoog vind je nu onder andere een Console Application, een Windows Forms Application en nog een paar meer. Wil je alles zelf doen, dan kies je voor een ‘Empty Project’.

Onderin de dialoog kun je de naam van de nieuwe applicatie kiezen. Bij het aanmaken van een nieuw project kun je kiezen of je ook een nieuwe solution wil (dit is de default keuze) of dat het project aan de huidige geladen solution toegevoegd moet worden.

Het toevoegen van nieuwe klassen

Je kunt aan een project altijd nieuwe klassen of andere items toevoegen. Dit doe je door rechts op het project in de solution explorer te klikken en dan menukeuze **Add→New Item**. Je kunt hier weer kiezen uit een aantal templates, net als bij de projecten. De template die je het meest zal gebruiken is de *C# Class*. Je voegt dan een nieuwe, lege klasse aan je project toe. Kies een geschikte naam



Figuur 45: Templates voor een nieuw project.

voor je klasse, en dubbelklik daarna op de toegevoegde klasse. Je ziet dat Visual Studio alvast wat `using`-directives en wat code voor je in het `.cs` bestand heeft gezet.

Documentatie en hulp

Via menukeuze **Help**→**View Help** kom je in de uitgebreide documentatie van Visual Studio. Je kunt rechtsboven op het vergrootglas klikken om een zoekterm in te vullen (bijvoorbeeld: **Graphics**), en dan zoekt Visual Studio naar relevante artikelen. Je vindt hier onder andere syntax, voorbeelden, en online discussies over het onderwerp waar je naar zoekt.

Als je de muiscursor iets langer laat zweven boven de naam van een klasse of methode, dan krijg je een eenregelige uitleg daarover te zien. Als de tekstcursor op de naam van een klasse of methode staat, kun je op de F1-toets drukken om uitgebreide documentatie in het Help-window te bekijken.

Bijlage B

Werkcollege-opgaven

2.1 *Programmeerparadigma's*

Waar of niet waar (en waarom?)

- Alle imperatieve talen zijn object-georiënteerd.
- Er zijn object-georiënteerde talen die niet procedureel zijn.
- Procedurele talen moeten worden gecompileerd.
- Declaratieve talen kunnen niet op dezelfde processor runnen als imperatieve, omdat die processor een toekenningsoopdracht kan uitvoeren, die in declaratieve talen niet bestaat.

2.2 *Namen veranderen*

- Bekijk de klasse `HalloWin3` in het voorbeeldprogramma `HalloWin3` in hoofdstuk 2. Wat moet er allemaal veranderen als we de naam van deze klasse willen veranderen in `Hoi`? Wat hoeft er strikt genomen niet te veranderen, maar zou logisch zijn om ook te veranderen?
- In het programma `HalloWin3` gebruiken we twee klassen: `HalloWin3` en `HalloForm`. Zouden we dit programma ook met één klasse kunnen schrijven? Wat moeten we dan veranderen?

2.3 Wat wordt er in een C#-methode aangeduid met `this`? In welke situatie is het niet toegestaan om `this` te gebruiken?

2.4 *Syntax en semantiek*

Wat wordt er verstaan onder de *syntax* van een (programmeer)taal-constructie? En wat is de *semantiek* van een taal-constructie?

3.1 *Commentaar*

Wat zijn de twee manieren om in C# commentaar bij een programma te schrijven?

3.2 *Declaratie, opdracht, expressie*

Wat is het verschil tussen een *declaratie*, een *opdracht* en een *expressie*?

3.3 *Vermenigvuldigen en delen*

Is er verschil tussen de opdrachten:

```
topy = y - 3*br / 2;
topy = y - 3/2 * br;
topy = y - br/2 * 3;
```

3.4 *Methode schrijven en gebruiken*

Maak een schetsje van de uitvoer van onderstaand programma.

Herschrijf nu de methode `TekenScherm`, zo dat er een extra methode wordt gebruikt om de regelmaat explicieter te maken.

```
class Iets : Form
{
    Iets()
    {
        this.Paint += this.tekenScherm;
    }
    public void tekenScherm(object o, PaintEventArgs pea)
    {
        Graphics g = pea.Graphics;
        g.DrawLine(Pens.Black, 10,10,20,20);
        g.DrawLine(Pens.Black, 20,10,10,20);
        g.DrawLine(Pens.Red, 30,10,50,30);
    }
}
```

```

        g.DrawLine(Pens.Red,    50,10,30,30);
        g.DrawLine(Pens.Blue,   60,10,90,40);
        g.DrawLine(Pens.Blue,   90,10,60,40);
    }
}

```

4.1 Technische begrippen

Geef korte definities van de begrippen *opdracht*, *variabele*, *methode*, en *object*.

Welke twee relaties heeft het begrip *klasse* met de genoemde begrippen?

4.2 Toekenningen aan variabelen

Bekijk een situatie waarin vier variabelen zijn gedeclareerd, en twee toekenningen zijn gebeurd, zoals na:

```

int x, y;
string s, t;
x = 40;
y = 12;

```

Beantwoord voor elke van de 9 groepjes opdrachten hieronder apart de vraag: wat zijn de waarden van *x* en *y* na het uitvoeren van het groepje opdrachten in bovengenoemde situatie?

<i>y</i> = <i>x</i> +1;	<i>x</i> = <i>y</i> ;	<i>x</i> = <i>y</i> +1;	<i>x</i> = <i>x</i> + <i>y</i> ;	<i>y</i> = <i>x</i> /3;
<i>x</i> = <i>y</i> +1;	<i>y</i> = <i>x</i> ;	<i>y</i> = <i>x</i> -1;	<i>y</i> = <i>x</i> - <i>y</i> ;	<i>x</i> = <i>y</i> *3;
			<i>x</i> = <i>x</i> - <i>y</i> ;	
<i>y</i> = 2/3* <i>x</i> ;	<i>y</i> = <i>x</i> %6;	<i>s</i> = "hallo";		<i>s</i> = "hallo";
<i>x</i> = 2* <i>x</i> /3;	<i>x</i> = <i>x</i> /6;	<i>t</i> = "\\//";		<i>t</i> = <i>s</i> ;
		<i>x</i> = <i>s</i> .Length;		<i>s</i> = <i>s</i> + "!";
		<i>y</i> = <i>t</i> .Length;		<i>x</i> = <i>s</i> .Length;
				<i>y</i> = <i>t</i> .Length;

Bij een van bovenstaande gevallen worden de waarden van *x* en *y* omgewisseld. Werkt dat voor alle mogelijke waarden van *x* en *y*? Zo ja, hoe is dat te verklaren? Zo nee, voor welke waarden niet?

4.3 Conversies

Stel dat is gedeclareerd:

```

int x;    string s;    double d;

```

Vul de volgende toekenningen aan met de benodigde conversies (waarbij we er van uitgaan dat de string inderdaad een getal voorstelt).

```

x = d;    x = s;
s = x;    s = d;
d = x;    d = s;

```

4.4 Syntactische categorieën

Hieronder staan 15 fragmenten uit een programma (in een blok van 3 bij 5). Schrijf bij elk fragment een letter passend bij het overeenkomstige fragment:

- T als het programmafragment een type is
- E als het programmafragment een expressie is
- O als het programmafragment een opdracht is
- D als het programmafragment een declaratie is
- H als het programmafragment een methode-header is
- X als het programmafragment geen van bovenstaande dingen is

double	double x;	(double)x*x
void x()	x==y+1	y=x!=x;
a%=x;	2xa0	0x2a
Button b=ok;	Button	new Button("")
this.add(b);	String ok(Button b)	class OK : Form

4.5 nog eentje: Syntactische categorieën

Hieronder staat 16 fragmenten uit een programma. Schrijf

op je antwoordblad een blok van 4 bij 4 vakjes en zet in elk vakje een letter passend bij het overeenkomstige fragment:

- **T** als het programmafragment een **type** is
- **E** als het programmafragment een **expressie** is
- **O** als het programmafragment een **opdracht** is
- **D** als het programmafragment een **declaratie** is
- **H** als het programmafragment een **methode-header** is
- **X** als het programmafragment geen van bovenstaande dingen is

Button	(bool>true	int t()	class A : Size
Button b;	const bool true;	int t=1;	this.Size=this.ClientSize;
b.Text = "ok";	bool	t==t+1	Size s(Size t)
new Button b;	while(true) t=1;	t=t+1;	new Size(x,y)

5.1 Keywords

Wat betekent het Engelse woord **void**, en in welke situatie is dit keyword nodig?

Wat betekent de Engelse afkorting **int**, en in welke situatie is dit keyword nodig?

Wat betekent in een C#-opdracht het woord **return**, en in welke situatie is dit keyword nodig?

Wat wordt in een C#-methode aangeduid door **this**, en in welke situatie is dit keyword nodig?

In welk soort methodes kan het *niet* gebruikt worden?

5.2 Methodes met een resultaat

- Schrijf een methode **restBijDeling** met twee parameters **x** en **y**, die de waarde van **x/y** teruggeeft, *zonder* daarbij de operator **%** te gebruiken.
- Schrijf een methode **omtrek**, die als resultaat oplevert wat de omtrek is van een rechthoek waarvan de lengte en breedte als parameters worden meegegeven.
- Schrijf een methode **diagonaal**, die als resultaat oplevert hoe lang de diagonaal is van een rechthoek waarvan de lengte en breedte als parameters worden meegegeven.
- Schrijf een methode **driewerf**, die drie aan elkaar geplakte kopieën oplevert van de string die als parameter wordt meegegeven, dus **driewerf("hoera!")** moet de string **"hoera!hoera!hoera!"** opleveren.
- Schrijf een methode **keer64**, die 64 aan elkaar geplakte kopieën oplevert van de string die als parameter wordt meegegeven. Probeer daarbij het schrijfwerk in de methode te beperken.

5.3 Uren, minuten, seconden

Stel dat de variabele **tijd** een (mogelijk groot) aantal seconden bevat. Schrijf een aantal opdrachten, waardoor de variabelen **uren**, **minuten** en **seconden** een daarmee overeenkomende waarde krijgen, waarbij de waarden van **minuten** en **seconden** kleiner dan 60 moeten zijn.

Schrijf bovendien een methode die, het omgekeerde probleem, uit drie parameters **uren**, **minuten** en **seconden** de totale hoeveelheid seconden berekent.

6.1 Spijkerschrift

a. Schrijf een methode **streepjes** met een getal als parameter. Je mag zonder controle aannemen dat de parameter 0 of groter is. De methode moet als resultaat een string opleveren met daarin zoveel verticale streepjes als de parameter aangeeft. Bijvoorbeeld: de aanroep **this.streepjes(5)** levert **"|||||"** op.

b. Schrijf een methode **spijker** met een getal als parameter. Je mag zonder controle aannemen dat de parameter 1 of groter is. De methode moet als resultaat een string opleveren met daarin het getal in spijkerschrift-notatie. Elk cijfer wordt daarin weergegeven met verticale streepjes, en de cijfers worden gescheiden door een liggend streepje. Er staan ook liggende streepjes aan het begin en het eind. Hier zijn een paar voorbeelden:

```
this.spijker(25) geeft "-||-|||||-"
this.spijker(12345) geeft "-|-||-|||-|||l-|||l-"
this.spijker(7) geeft "-|||||l-"
this.spijker(203) geeft "-||--||l-"
```

Hint: verwerk eerst het laatste cijfer, en herhaal dan voor de rest van de cijfers.

6.2 Ringen

Neem het volgende raamwerk voor een programma:

```

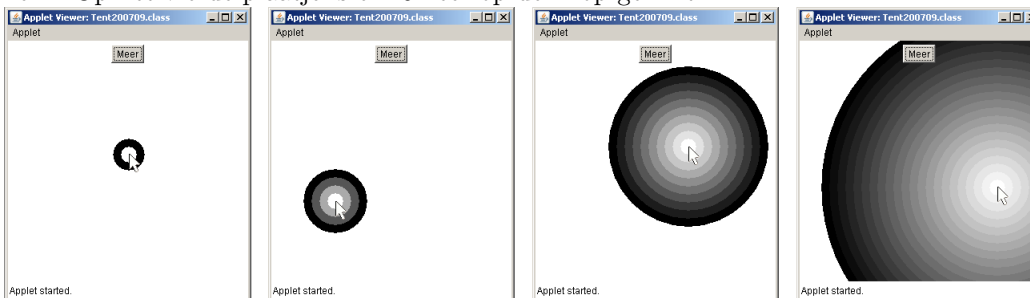
class Ring : Form
{
    Ring()
    {
        Button knop = new Button();
        knop.Text = "Meer";
        knop.Location = new Point(50,10);
        this.Controls.Add(knop);
        this.Paint += this.teken;
        this.MouseMove += this.beweeg;
        knop.Click += this.klik;
    }
    static void Main()
    {
        Application.Run(new Ring());
    }
}

```

Schrijf de ontbrekende declaraties en methoden in de klasse, zo dat het programma de volgende werking krijgt: een zwarte ring met een doorsnede van 40 beeldpunten beweegt mee met de bewegingen van de muis. De witte kern in het midden van de ring heeft een doorsnede van 20 beeldpunten. (Je kunt ook zeggen dat de dikte van het zwarte gedeelte van de ring 10 beeldpunten is). Zie het eerste plaatje hieronder.

In het window is een knop zichtbaar met het opschrift “Meer”. Elke keer dat de gebruiker daarop drukt, wordt de ring groter: er komt een band bij met een dikte van 10 beeldpunten; de totale diameter wordt dus 20 beeldpunten groter. De buitenste band is altijd zwart, de kern blijft altijd wit, en de banden daartussen krijgen een vloeiend verloop van zwart naar wit. Zie het tweede plaatje: er is tweemaal op de knop geklikt, en er zijn dus een zwarte buitenring, een donkergrijze band, een lichtgrijze band, en een witte kern.

Op het derde plaatje is er 8 keer op de knop ‘Meer’ geklikt, en zijn er dus 9 banden en een witte kern. Op het vierde plaatje is er 20 keer op de knop geklikt.



6.3 Totaal van een rij getallen

Schrijf een statische methode **totaal** met een getal n als parameter, die het totaal van de getallen van 0 tot en met n als resultaat oplevert.

Als n kleiner of gelijk is aan 0, moet het antwoord 0 zijn.

Hint: gebruik een variabele om het resultaat in op te bouwen.

6.4 Product van een rij getallen

Schrijf een statische methode **faculteit** met een getal n als parameter, die het product van de getallen 1 tot en met n als resultaat oplevert, dus alle getallen van één tot en met n met elkaar vermenigvuldigd.

Als n kleiner of gelijk is aan 1, moet het antwoord 1 zijn.

6.5 Machtsverheffen

Schrijf een statische methode **macht** met een grondtal x en een exponent n als parameter. Het resultaat moet x^n zijn, dus x wordt n keer met zichzelf vermenigvuldigd.

Je mag aannemen dat n een natuurlijk getal is (dus niet negatief is). De methode moet ook goed werken als n gelijk is aan 0, en als x niet een geheel getal is.

Hint: gebruik een variabele om het resultaat in op te bouwen. Vergeet niet om die resultaat-variabele ook een beginwaarde te geven!

(Nota bene: in veel programmeertalen kan dit ook worden gedaan met de operator \wedge . In C# kan dat niet, wel is er een methode `Math.Pow`, maar die mag je hier niet gebruiken want anders wordt het te makkelijk :-).

6.6 Reeksen

De ‘faculteit’ van een natuurlijk getal is de uitkomst van alle getallen vanaf 1 tot en met dat getal met elkaar vermenigvuldigd. Bijvoorbeeld: de faculteit van 3 is $1 \times 2 \times 3 = 6$. Schrijf een statische methode `faculteit` die de faculteit van zijn parameter uitrekent. Je mag er zonder controle van uitgaan dat de parameter ≥ 1 is.

Een benadering van cosinus hyperbolicus van een reel getal x kun je berekenen door:

$$1 + x^2/2! + x^4/4! + x^6/6! + x^8/8! + x^{10}/10! + \dots$$

De notatie $6!$ betekent hierin de faculteit van 6. Schrijf een statische methode `coshyp` die deze benadering berekent door 20 van deze termen te sommeren, en dat als resultaat oplevert. Je mag (maar hoeft niet) zelf extra hulp-methoden definiëren.

6.7 String verveelvoudigen

Eerder schreven we een methode `driewerf`, die drie kopieën van een string aan elkaar plakte.

Schrijf nu een statische methode `veelwerf`, die behalve een string ook een getal als parameter heeft, die aangeeft hoe vaak de string herhaald moet worden. Bijvoorbeeld, de aanroep `veelwerf("ha", 4)` levert "hahahaha". Als het getal 0 is of negatief, moet deze methode een lege string opleveren.

Hint: gebruik een variabele om het resultaat in op te bouwen. Vergeet niet om die resultaat-variabele ook een beginwaarde te geven!

6.8 Priemgetallen

- Schrijf een statische methode `even` die als resultaat oplevert of zijn parameter een even getal is. Bedenk goed wat een handig type is voor de parameter en het resultaat van de methode.
- Schrijf een statische methode `deelbaar` met twee parameters x en y , die als resultaat oplevert of x een deelbaar is door y , dat wil zeggen de deling x/y precies opgaat.
- Schrijf een statische methode `kleinsteDeler`, die het kleinste getal ≥ 2 bepaalt waar de parameter door deelbaar is.
Hint: probeer één voor één de mogelijke delers, en stop als je er eentje gevonden hebt.
- Schrijf een statische methode die bepaalt of een getal een priemgetal is, dat wil zeggen alleen maar deelbaar is door 1 en zichzelf.

6.9 Stralen

Gegeven is de volgende klasse:

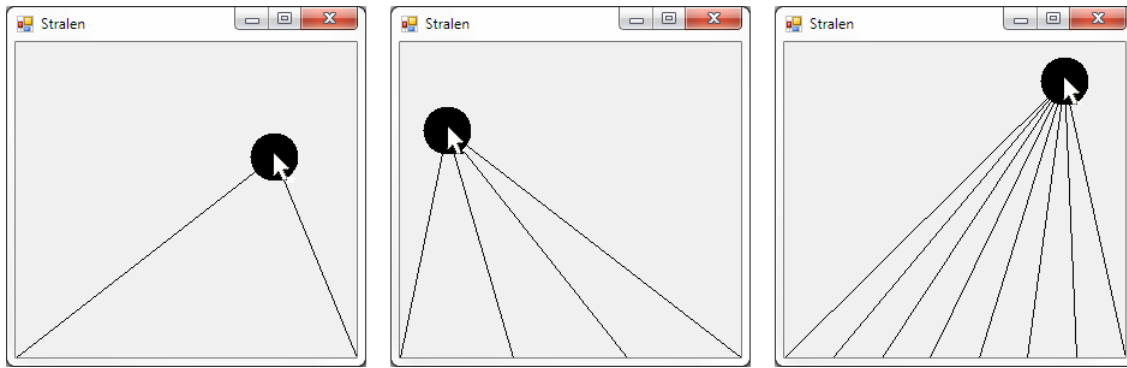
```
class Program
{
    public static void Main()
    {
        Stralen s = new Stralen();
        s.Text = "Stralen";
        Application.Run(s);
    }
}
```

Schrijf de klasse `Stralen`, zo dat het programma zich als volgt gaat gedragen.

Er is een zwart opgevulde cirkel met een diameter van 40 pixels in beeld. Het middelpunt van de cirkel bevindt zich op de positie van de muis; de cirkel beweegt dus mee met de muis.

Twee lijnen verbinden het midden van de cirkel met de twee onderhoeken van het window. Elke keer als de gebruiker met de muis klikt komt er een lijn bij. De lijnen monden op gelijke afstanden uit op de onderrand van het window.

Zie onderstaande figuur, met daarin: de beginpositie, de situatie na 2 keer klikken, en de situatie na nog 4 keer klikken. (De pijl geeft de muiscursor aan, deze hoeft je niet te tekenen).



8.1 Try/Catch

Wat is de semantiek van een opdracht van de volgende vorm: `try A catch (E) B`?

8.2 Lichtkrant

Schrijf een programma'tje genaamd "Lichtkrant". Deze laat een tekst langzaam door het beeld bewegen, van rechts naar links. Als de tekst helemaal uit beeld is verdwenen, begint hij weer opnieuw.

De tekst die gebruikt wordt staat in een constante-declaratie. Ga ervan uit dat elke letter gemiddeld 10 beeldpunten breed is. Gebruik een Thread-object voor de animatie!

8.3 Stabiel beeld bij animaties

Als het Control opnieuw getekend moet worden (bijvoorbeeld omdat zijn Invalidate-methode is aangeroepen), dan worden alle "abonnees" van het Paint-event aangeroepen. Maar daarvoor nog wordt eerst de `virtual` methode `OnPaintBackground` aangeroepen, die er verantwoordelijk voor is om de achtergrond neutraal te kleuren.

Bij snelle animaties kan dat een hinderlijk knipper-effect geven: je ziet steeds de achtergrondkleur even oplichten, waarna het plaatje weer opnieuw getekend wordt. Verzin een manier om dat te voorkomen, om een stabiel beeld te krijgen.

8.4 Geheugen tekenen

Gegeven zijn de volgende klasse-definities:

```
class Een
{
    int x;
    public Een()
    {
        x = 0;
    }
    public void setX(int a)
    {
        x = a;
    }
}

class Twee
{
    int x; Een e;
    public Twee(Een b, int c)
    {
        e = b;
        x = c+1;
    }
    public Een getE()
    {
        return e;
    }
}

class Drie : Een
{
    Twee p, q;
    public Drie()
    {
        p = new Twee( new Een(), 1 );
        p.getE().setX(7);
        q = new Twee( p.getE() , 2 );
        q.getE().setX(8);
    }
}
```

```

        p = new Twee( this , 3 );
        p.getE().setX(9);
    }
}

```

Teken, in dezelfde stijl als figuur 22 en 24 van het diktaat, de situatie die in het geheugen ontstaat na uitvoering van

```
Drie d = new Drie();
```

Maak, net als in het voorbeeld, duidelijk onderscheid tussen de naam en de waarde van de variabelen: de naam staat naast de hokjes, de waarde er in. Object-verwijzingen moeten, net als in het voorbeeld, met een duidelijke stip beginnen in het hokje van de verwijzings-variabele, en wijzen naar de rand van het object.

8.5 *Type van Add*

Iemand schrijft:

```

class Hallo : Form
{
    Hallo()
    {
        b = new Button();
        t = new TextBox();
        this.Controls.Add(b);
        this.Controls.Add(t);
    }
}

```

- Hoe kan het correct zijn dat `Add` zowel een `Button` als een `TextBox` als parameter accepteert? Wat is het type van de parameter van `Add`?
- Mag je in plaats van de laatste opdracht ook schrijven:

```
b.Controls.Add(t);
```

zo nee, waarom niet? zo ja, wat gebeurt er dan?

8.6 *Klassen en overerving*

Gegeven de volgende klassen:

```

class A
{
    public float var1;
    protected int var2;
    private bool var3;
    public float Var1
    {
        get { return var1; }
    }
    public int Var2
    {
        get { return var2; }
        set { if (value > 0) var2 = value; }
    }
    public void methode_in_A()
    {
        ...
    }
}
class B : A
{
    public int var4;
    private int var5;
    public void methode_in_B()
    {
        ...
    }
}

```

- Geef aan of de volgende expressies mogelijk zijn in `methode_in_A`:

```
this.var1    this.var2    this.var3
```


`this.var4` `this.Var2` `base.var1`

b. Geef aan of de volgende expressies mogelijk zijn in `methode_in_B`:

`this.var1` `this.var2` `this.var3`
`this.var5` `this.Var2` `base.var1`
`base.var2` `base.var3` `base.Var2`

8.7 *Type-controle*

Worden de types van expressies over het algemeen gecontroleerd tijdens het compileren of tijdens het runnen van het programma?

Er is een uitzondering op deze regel. In welk geval is dat? (En waarom is die uitzondering nodig?)

9.1 Arrays

- Schrijf een methode `aantalNullen` die als parameter een array van integers meekrijgt. Het resultaat van de methode is het aantal nullen dat in de array staat.
- Schrijf een methode `optellen` die twee integer arrays van gelijke lengte als parameters meekrijgt. De methode telt de waarden in de twee arrays bij elkaar op en geeft als resultaat een nieuwe array. Bijvoorbeeld, gegeven een array `array1 = { 0, 3, 8, -4 }` en een array `array2 = { 10, 2, -8, 8 }`. De aanroep van de methode `optellen(array1, array2)` levert als resultaat `{ 10, 5, 0, 4 }` op.
- Schrijf een methode `eerstePositie` die als parameters een `string` en een `char` meekrijgt en die de eerste positie van het karakter in de string als resultaat teruggeeft. Als het karakter niet in de string voorkomt, dan levert de methode -1 op. Bijvoorbeeld:

```
int resultaat = this.eerstePositie("arjan", 'j'); // levert 2 op
resultaat = this.eerstePositie("jeroen", 'j'); // levert 0 op
resultaat = this.eerstePositie("hans", 'j'); // levert -1 op
```

9.2 String-methoden

- In de klasse `String` zit de methode `ToUpper`, die een hoofdletterversie van de string oplevert. Die kun je bijvoorbeeld aanroepen met

```
h = s.ToUpper( );
```

Als je zelf de klasse `String` zou moeten schrijven, hoe zou je deze methode dan kunnen definiëren (gebruikmakend van andere methoden in de klasse `String`)?

- In de klasse `String` zit een methode `Replace`. Deze methode levert een nieuwe string op, waarin elk voorkomen van het character dat als eerste parameter wordt meegegeven, is vervangen door het character dat als tweede parameter wordt meegegeven. Bijvoorbeeld:

```
"Utrecht".replace('t','x') // geeft "Uxrechx"
"A+2+#?".replace('+','9') // geeft "A929#?"
```

Stel dat je de auteur van de klasse `String` bent. Veel andere methoden van die klasse zijn al geschreven (die mag je dus gebruiken). Schrijf de methode `Replace`.

- Ook is er een methode `EndsWith`, die oplevert of een string eindigt met de string die als parameter wordt meegegeven. Bijvoorbeeld:

```
"Utrecht".endsWith("recht") // geeft true
```

Stel dat je de auteur van de klasse `String` bent. Veel andere methoden van die klasse zijn al geschreven (die mag je dus gebruiken), maar nog niet de `Substring` en `IndexOf` methoden (die mag je dus niet gebruiken). Schrijf de methode `endsWith`.

9.3 Tweedimensionale arrays

Er zijn twee manieren om een twee-dimensionale array te declareren:

```
int [,] eerste;
int [][] tweede;
```

Wat is het verschil? In welke situatie is het handig om de `tweede` te gebruiken?

9.4 String-methoden

In de klasse `String` zitten onder andere de volgende methoden:

```
static bool IsNullOrWhiteSpace(string)
static int Compare(string, string)
```

- De naam van `IsNullOrWhiteSpace` spreekt voor zichzelf. Onder 'whitespace' verstaan we spaties, tab-tekens en newline-tekens. Schrijf deze methode, *zonder* gebruik te maken van de bestaande `IsNullOrEmpty` en `IsNullOrWhiteSpace` methoden.
- De methode `Compare` levert 0 op als de twee parameters precies gelijk zijn. Hij levert een negatief getal op (bijvoorbeeld -1, maar iets anders mag ook) als de eerste parameter kleiner is dan de tweede, en een positief getal (bijvoorbeeld 1) als die groter is. Met kleiner en groter wordt hier de woordenboek-ordening bedoeld: de eerste letter waar de strings verschillen bepaalt de ordening (volgens de Unicodes van die letters). Is de ene string een beginstuk van

de andere, dan is de kortste de kleinste. Spaties en leestekens tellen gewoon mee, die hoeven dus niet speciaal behandeld te worden.

Voorbeelden:

<code>String.Compare("aap", "noot")</code>	geeft een negatief getal, want 'a' < 'n'
<code>String.Compare("noot", "nieten")</code>	geeft een positief getal, want 'o' > 'i'
<code>String.Compare("niet", "nietmachine")</code>	geeft een negatief getal vanwege de lengte
<code>String.Compare("noot", "noot")</code>	geeft 0, want precies gelijk
<code>String.Compare("noot", "NOOT")</code>	geeft een positief getal, want 'n' > 'N'

De methode neemt aan dat de parameters niet `null` zijn (en controleert dat ook niet).

Schrijf deze methode, *zonder* gebruik te maken van de bestaande `Compare` en `CompareTo` methoden.

9.5 Zoeken en sorteren

- Schrijf een statische methode `grootste` met als parameter een array van doubles, met als resultaat de *hoogste waarde* die in de array voorkomt.
- Schrijf een andere statische methode `plaatsGrootste`, met als resultaat het *volgnummer* (de index in de array) van de grootste waarde.
- Schrijf een methode met als parameter een array van doubles. De methode moet opleveren hoe vaak de kleinste waarde van de array voorkomt.
Bijvoorbeeld: als de array de waarden 9,12,9,7,12,7,8,25,7 bevat, dan is het resultaat 3, omdat de kleinste waarde (7) drie maal voorkomt.
- Maak een variant van de methode `plaatsGrootste`, waarbij niet de hele array doorzocht wordt, maar alleen de eerste `n` elementen, waarbij `n` een aparte parameter is.
- Gebruik deze methode in de volgende methode: schrijf een statische methode `Sorteer`, met als parameter een array van doubles. Het resultaatstype is `void`. Na afloop van het uitvoeren van de methode moeten de elementen in de array in opklimmende volgorde van grootte staan. Hint hierbij: zoek eerst de plaats van de grootste waarde in de hele array. Verwissel deze waarde met de waarde op de laatste plaats. Dan staat de grootste waarde alvast achteraan, waar hij hoort. Zoek nu de grootste waarde van het resterende deel van de array. Die waarde wordt verwisseld met de op één na laatste waarde in de array. Dit gaat zo verder: zoek weer het grootste element van de array minus de laatste twee elementen, verwissel die met op het twee na laatste element, enzovoorts.
- Schrijf een statische methode met als parameter een array van integers en een losse integer, die oplevert op welke plaats de losse integer als eerste in de array voorkomt. Als het getal nergens voorkomt, moet de methode `-1` opleveren.
- (moeilijker) Als je weet dat de array op volgorde gesorteerd is, kun je op een slimmere manier zoeken: kijk in het midden van de array of je daar de gezochte waarde aantreft. Zo ja, mooi. Zo nee, dan weet je of je in de helft links ervan of in de helft rechts ervan moet verder zoeken. Op deze manier wordt het te doorzoeken stuk van de array bij elke stap twee keer zo klein. Gebruik twee integers die de grenzen van het te doorzoeken stuk array aanduiden.

9.6 Strings

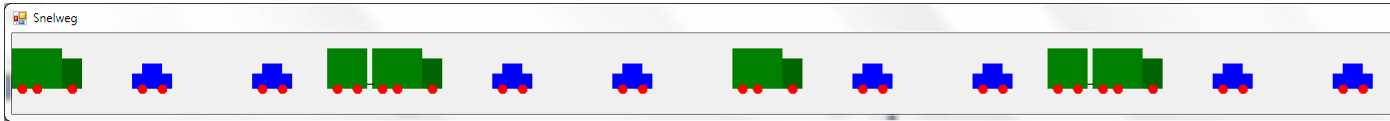
In de klasse `String` zitten onder andere methoden met de volgende headers:

```
public string Substring(int startIndex);
public string Substring(int startIndex, int length);
public int IndexOf(char value)
```

Stel dat je de auteur van de klasse `String` bent, en deze methoden zijn nog niet aanwezig. Schrijf deze drie methoden. Je mag daarbij niet gebruik maken van methoden `Substring` en `IndexOf` (niet de genoemde drie, maar ook niet van andere varianten met deze namen, want we nemen aan dat die er nog niet zijn), maar wel van de andere methoden die in de klasse `String` al aanwezig zijn, en van je eigen methoden.

9.7 subklassen en arrays

Bekijk het onderstaande programma. Het moet een file auto's op de snelweg tekenen, zoals in de screendump hieronder. Elke derde auto is een vrachtwagen, en elke tweede vrachtwagen is een combinatie met aanhanger.



```

public class Snelweg : Form
{
    public Snelweg()
    {
        this.Text = "Snelweg";
        this.ClientSize = new Size(1800, 80);
        this.Paint += this.tekenSnelweg;
        // TODO: ontbrekend deel van de constructor
    }
    public void tekenSnelweg(object o, PaintEventArgs pea)
    {
        for (int t = 0; t < rijbaan.Length; t++)
            rijbaan[t].Teken(pea.Graphics, t*120, 60);
    }
    static void Main()
    {
        Application.Run(new Snelweg());
    }
}

class MotorVoertuig
{
    public void Teken(Graphics gr, int x, int y)
    {
    }
}

class PersonenAuto : MotorVoertuig
{
    public void Teken(Graphics gr, int x, int y)
    {
        gr.FillRectangle(Brushes.Blue, x, y - 20, 40, 15);
        gr.FillRectangle(Brushes.Blue, x+10, y - 30, 20, 10);
        gr.FillEllipse(Brushes.Red, x + 5, y - 10, 10, 10);
        gr.FillEllipse(Brushes.Red, x + 25, y - 10, 10, 10);
    }
}

class Vrachtwagen : MotorVoertuig
{
    public void Teken(Graphics gr, int x, int y)
    {
        gr.FillRectangle(Brushes.Green, x, y - 45, 50, 40);
        gr.FillRectangle(Brushes.DarkGreen, x+50, y - 35, 20, 30);
        gr.FillEllipse(Brushes.Red, x + 5, y - 10, 10, 10);
        gr.FillEllipse(Brushes.Red, x + 20, y - 10, 10, 10);
        gr.FillEllipse(Brushes.Red, x + 55, y - 10, 10, 10);
    }
}

class Combinatie : Vrachtwagen
{
    public void Teken(Graphics gr, int x, int y)
    {
        // de vrachtwagen
        gr.FillRectangle(Brushes.Green, x, y - 45, 50, 40);
        gr.FillRectangle(Brushes.DarkGreen, x + 50, y - 35, 20, 30);
        gr.FillEllipse(Brushes.Red, x + 5, y - 10, 10, 10);
        gr.FillEllipse(Brushes.Red, x + 20, y - 10, 10, 10);
        gr.FillEllipse(Brushes.Red, x + 55, y - 10, 10, 10);
        // de aanhanger
        gr.DrawLine(Pens.Black, x - 5, y - 10, x, y - 10);
        gr.FillRectangle(Brushes.Green, x-45, y - 45, 40, 40);
        gr.FillEllipse(Brushes.Red, x -40, y - 10, 10, 10);
        gr.FillEllipse(Brushes.Red, x -20, y - 10, 10, 10);
    }
}

```

- Er ontbreekt nog een declaratie. Schrijf deze declaratie, en geef aan waar die moet staan.
- Schrijf het ontbrekende deel van de constructor van `Snelweg`.
- In het gegeven programma zit nog een fout, waardoor er helemaal niets zichtbaar wordt. Hoe

- komt dat, en hoe kan de fout worden verbeterd?
- De programmeur heeft een flink stuk code met copy&paste gedupliceerd. Waarom is dat geen goed idee?
 - Hoe had het dupliceren van de code het beste vermeden kunnen worden?
 - We willen de object-georiënteerde opzet van het programma nog verder doorvoeren, zo dat ook de concepten ‘wiel’ en ‘aanhanger’ met klassen worden gemodelleerd. Hoe kan dat netjes worden aangepakt? Zorg ervoor dat code-duplicatie zo veel mogelijk vermeden kan worden, en dat er nooit door een programmeurfout een losse aanhanger op de weg terecht kan komen. Je hoeft dit niet helemaal uit te programmeren; geef alleen aan welke klassen er komen, hoe hun subklasse-relatie is, en welke declaraties er in (bestaande en/of nieuwe) klassen komen te staan.

9.8 Lijnen

Gegeven is de volgende klasse, met daarin de methode `Main` en twee andere handige methoden.

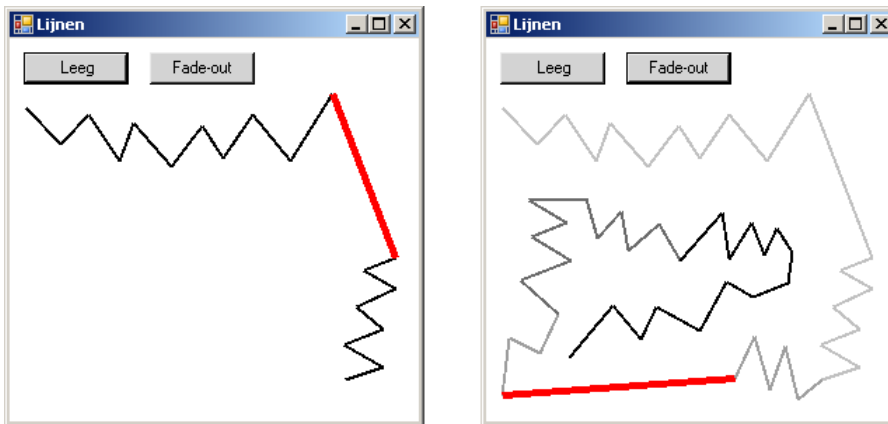
```
public class Help
{
    public static void Main()
    {
        Lijnen lijnen = new Lijnen();
        lijnen.Text = "Lijnen";
        lijnen.BackColor = Color.White;
        Application.Run(lijnen);
    }

    public static Button MaakKnop(string tekst, int x, int y, EventHandler eh)
    {
        Button b = new Button();
        b.Text = tekst;
        b.BackColor = Color.LightGray;
        b.Location = new Point(x, y);
        b.Click += eh;
        return b;
    }

    public static int Kwadraat(int x)
    {
        return x * x;
    }
}
```

Schrijf nu de klasse `Lijnen`, zo dat het programma zich als volgt gaat gedragen:

- De gebruiker ziet twee knoppen met het opschrift ‘Leeg’ en ‘Fade out’.
- De gebruiker kan verder overal in het window klikken. De aangeklikte punten worden verbonden door lijnen. (Na de eerste klik ziet de gebruiker nog niets, bij de tweede klik verschijnt er een lijn, bij de derde klik en tweede lijn, enz).
- Er zijn maximaal 100 lijnen zichtbaar. Als de gebruiker daarna toch meer punten aanklikt, gebeurt er niets (ook geen foutmelding!).
- Alle lijnen zijn zwart met dikte 2, behalve de langste lijn: die is rood met dikte 5. (Als er meerdere lijnen de langste zijn, mag je er daar een van kiezen, of ze allemaal rood maken).
- Na het indrukken van de knop ‘Leeg’ verdwijnen alle lijnen. De gebruiker kan dan weer met 100 nieuwe lijnen beginnen.
- Na het indrukken van de knop ‘Fade out’ beginnen de lijnen langzaam te vervagen: elke seconde worden ze 10% grijzer. Theoretisch gesproken worden ze dus nooit helemaal wit, maar in de praktijk zijn ze na een halve minuut onzichtbaar geworden op de witte achtergrond. Nieuw aangeklikte lijnen beginnen wel weer zwart. De langste lijn blijft rood.
- Na nogmaals indrukken van de knop ‘Fade out’ stopt het vervagen, na een derde keer indrukken gaat het weer verder waar het gebleven was, enz.



In het voorbeeld links heeft de gebruiker 19 punten aangeklikt. De langste lijn is dikker en rood. In het voorbeeld rechts is de 'Fade out' knop gebruikt. Later gemaakte lijnen zijn nog donkerder. Inmiddels is ook een andere lijn de langste.

10.1 *TextReader*

In de klasse `TextReader` zitten onder andere de volgende methoden:

```
char Read();           // geeft de eerstvolgende letter, of -1 als die er niet is
string ReadLine();     // geeft alles tot (maar zonder) de eerstvolgende newline
string ReadToEnd();    // geeft de rest van de file
```

Welk van deze methoden moet(en) **abstract** gedefinieerd worden? Implementeer de overige methode(n). Is daarbij de modifier **virtual** en/of **override** nodig?

10.2 *Tekstfiles en collections*

Schrijf een programma (met commandline interface) met de volgende specificatie. Het programma wordt door de gebruiker opgestart vanaf een commandoregel. De gebruiker specificeert daarbij twee filenamen. Het programma leest een tekstfile met de eerste filenaam als naam. Daarna schrijft het een tekstfile met de tweede filenaam als naam. De uitvoer moet elk woord uit de invoer op aparte regels vermelden. De woorden staan daarbij op (alfabetische, of eigenlijk Unicode) volgorde, waarbij dubbele woorden slechts éénmaal worden vermeld.

Elk groepje karakters zonder spatie erin beschouwen we als woord. Je mag ervan uitgaan dat er tussen de woorden precies 1 spatie staat.

Als de gebruiker te weinig of te veel filenamen opgeeft, of als er een fout optreedt bij het lezen of schrijven, krijgt de gebruiker daarvan een korte melding.

Voorbeeld: als de invoer de volgende twee regels bevat:

```
dit IS een *%#$ voorbeeld
van een tekst!
```

dan moet de uitvoer de volgende zeven regels bevatten:

```
*%#$
IS
dit
een
tekst!
van
voorbeeld
```

Als dit niet helemaal lukt, kun je je de volgende versimpelingen permitteren: gebruik vaste filenamen in plaats van door de gebruiker gespecificeerde; laat het sorteren en ontdubbelen achterwege; verwerk niet de woorden, maar de regels van de tekst; laat de foutmeldingen achterwege.

10.3 *Decorator streams*

De klasse `Stream` heeft onder andere de volgende methoden:

```
int ReadByte();        // geeft het eerstvolgende byte, of -1 als dat er niet meer is
int Read(byte[] doel, int n); // leest maximaal n bytes, en zet die neer in doel.
                          // geeft het aantal gelezen bytes terug.
```

Het is vaak efficiënter om, met methode `Read`, een heel blok tegelijk te lezen. Maar het is gemakkelijker om een losse byte te kunnen lezen als je hem nodig hebt.

Ziedaar het nut van de klasse `BufferedStream`. Bij de constructor van deze klasse geef je hem een reeds bestaande `Stream` in beheer. Als je daarna `ReadByte` aanroept, spreekt hij het object dat hij in beheer heeft aan om meteen maar 1000 bytes te lezen. De eerste daarvan geeft hij terug, de overige bewaart hij zolang in een array. Roep je daarna nog eens `ReadByte` aan, dan kan die 'uit voorraad' geleverd worden, en hoeft de onderliggende file dus niet aangesproken te worden. Pas als de voorraad op is, wordt er weer een nieuw blok ingelezen.

Opgave: schrijf de klasse `BufferedStream`, met een constructormethode en de methode `ReadByte`.

10.4 Lists

In de klasse `List<string>` zitten onder andere methoden met de volgende headers:

```
public void Reverse()
public int LastIndexOf(string item)
public bool Contains(string item)
```

Stel dat je de auteur van de klasse `List` bent, en deze methoden zijn nog niet aanwezig. Schrijf deze drie methoden. Je mag daarbij niet gebruik maken van de bestaande methoden met dezelfde naam en ook niet van andere varianten met deze namen, want we nemen aan dat die er nog niet zijn, maar wel van de andere methoden die in de klasse `List` al aanwezig zijn, en van je eigen methoden.

10.5 Collections

Schrijf een methode `verwijderDubbeleGetallen` die als parameter een `List` van ints meekrijgt. De methode verwijdert alle dubbele getallen in de lijst die meegegeven is als parameter. Bijvoorbeeld, de lijst bevattende 0, 1, 3, 2, 1, 5, 2 wordt 0, 1, 3, 2, 5. Let op: het return-type van deze methode is `void`!

10.6 Klassen en interfaces

Eén van de volgende drie declaraties-met-toekenningen is correct. Welke is dat, en waarom zijn de andere twee niet correct?

```
List a = new IList(); // versie 1
IList b = new List(); // versie 2
IList c = new IList(); // versie 3
```

Beschrijf een situatie waarin de correcte versie van bovenstaande regels een voordeel heeft, vergeleken met het in ieder geval ook correcte:

```
List d = new List(); // versie 4
```

Waarin onderscheiden abstracte klassen zich van interfaces? Geef een voorbeeld van een situatie waar je een abstracte klasse zou gebruiken. Geef ook een voorbeeld van een situatie waar je een interface zou gebruiken.

10.7 Abstracte klassen

Gegeven de volgende klassen:

```
abstract class A
{
    public abstract void methode1();
    public void methode2()
    {
        return;
    }
}
class B : A
{
    public override void methode1()
    {
        return;
    }
    public void methode3(A a)
    {
        a.methode1();
    }
}
```

```
    }
}
```

Geef voor elk van de volgende opdrachten aan of ze mogen:

```
A obj;
obj = new A();
obj = new B();
obj.methode1();
obj.methode2();
obj.methode3(object);
B anderObject = (B)(new A());
A nogEenObject = (A)obj;
obj.methode3(nogEenObject);
A[] lijst;
lijst = new A[10];
lijst[0] = new A();
lijst[1] = new B();
List<A> nogEenLijst = new List<A>();
```

10.8 List en foreach

Gegeven de volgende klasse:

```
class Tellertje
{
    int waarde = 0;
    public Tellertje(int w)
    {
        waarde = w;
    }
    public void increment()
    {
        waarde++;
    }
}
```

en de volgende opdrachten in een **Applicatie**-klasse:

```
class Applicatie
{
    static void Main()
    {
        List<Tellertje> lijst = new List<Tellertje>();
        for (int i=0; i<25; i++)
            lijst.Add(new Tellertje(i));
    }
}
```

- Schrijf een methode **ophogen** in de klasse **Applicatie** die als parameter een **ICollection<Tellertje>** meekrijgt en die alle tellertjes ophooft (via de **increment**-methode). Maak een versie die de **foreach**-opdracht gebruikt, en een versie die een **for**- of een **while**-opdracht gebruikt.
- Stel dat we in plaats van een **List<Tellertje>** nu een **EigenLijst<Tellertje>** zouden willen gebruiken, en de **EigenLijst**-klasse is onze eigen versie van een lijst, die ook de **ICollection** interface implementeert, wat moeten we dan veranderen aan de **ophogen**-methode zodat we hem ook met onze eigen lijstklasse kunnen gebruiken?

10.9 Klassen en overerving

Beschouw de volgende twee klassen:

```
class A {
    public void func1() { Console.WriteLine("A::func1"); }
    public virtual void func2() { Console.WriteLine("A::func2"); }
}
class B : A {
    public void func1() { Console.WriteLine("B::func1"); }
```



```
    public override void func2() { Console.WriteLine("B::func2"); }
}
```

Wat is de uitvoer van de volgende serie opdrachten?

```
A x = new A();
A y = new B();
B z = new B();
x.func1();
x.func2();
y.func1();
y.func2();
z.func1();
z.func2();
```

10.10 Nog meer tekstfiles

Schrijf een programma met de volgende specificaties. Het programma wordt door de gebruiker vanaf een commandoregel opgestart, en toont zijn uitvoer ook via de console. Het programma heeft dus geen eigen window.

Bij het starten van de programma geeft de gebruiker ook één of meer namen van tekstfiles op. Van elke file wordt gerapporteerd:

- welke regel het langste is
- welke regel het meeste woorden bevat

Als er een probleem is bij het lezen van de file wordt dat gemeld. Aan het eind wordt bovendien gemeld welk van de files het meeste regels heeft.

In het programma moet er een aparte methode zijn voor het verwerken van n file. Bij het rapporteren van regels tekst, moet de geciteerde regel tussen aanhalingstekens worden getoond (zoals in het voorbeeld). Bij het rapporteren van de langste tekst moet ook de naam van de file genoemd worden (zoals in het voorbeeld).

Een woord is een aaneengesloten reeks van één of meer letters, cijfers en leestekens. Woorden worden dus gescheiden door spaties. Let op: er kunnen meerdere spaties tussen twee woorden staan, maar dat telt niet als extra woorden.

Voorbeeld: De inhoud van de file `pipo.txt` is:

```
Kort he!
Eenentwintig vijfendertig zevenenveertig.
Een twee drie vier vijf zes zeven.
```

De inhoud van de file `test.txt` is:

```
Dit is de eerste regel.
Dit de tweede.
Vanwege de waarschuwingsknipperlichtinstallatie is dit een hele lange regel geworden.
Op deze regel staan dus wel erg veel maar niet zo erg lange woorden.
Dit is de vijfde regel.
Op deze regel staan veel extra spaties.
Deze zevende regel is behoorlijk lang, maar overtreft toch niet tot de langste.
```

De file `fuot.txt` bestaat niet.

De gebruiker start het programma bijvoorbeeld met:

```
TekstAnalyse pipo.txt fuot.txt test.txt
```

De output is dan:

```
Langste regel van pipo.txt is: "Eenentwintig vijfendertig zevenenveertig."
Meeste woorden in: "Een twee drie vier vijf zes zeven."
Probleem bij het lezen van fuot.txt
Langste regel van test.txt is: "Vanwege de waarschuwingsknipperlichtinstallatie is dit een hele lange reg
Meeste woorden in: "Op deze regel staan dus wel erg veel maar niet zo erg lange woorden."
De meeste regels zitten in file test.txt
```

Bijlage C

Practicumopdrachten

C.0 Oefenpracticum

In deze eerste serie practicumopgaven gaan we de mogelijkheden van C# en Microsoft Visual Studio verkennen. Deze opgaven moeten wel gemaakt worden, maar hoeven niet ingeleverd te worden zoals de andere practicumopgaven. Gebruik de Visual Studio tutorial voor uitleg over de bediening van de *Integrated Development Environment* (IDE).

1. Filestysteem

- a. Maak een directory `Imp` aan in “My Documents” voor de files die je bij dit vak nodig gaat hebben.

2. Voorbeeldprogramma's

- a. Download en unzip de Solution met alle voorbeeldprogramma's.
- b. Open de solution in Visual Studio.
- c. Maak het programma "HalloWin1" het active project.
- d. Run het programma.
- e. Bekijk de broncode van het programma.
- f. Bekijk de broncode van HalloWin3.

3. Console Application

- a. Sluit de Solution met voorbeeldprogramma's.
- b. Maak een nieuw project met menukeuze `File→New→Project`.
- c. Kies in de linker-kolom 'Visual C#' en in de rechter-kolom 'Console App (.NET Framework)', vul als naam in: 'Leeftijd', en zorg ervoor dat die in een nieuwe Solution 'Practicum' terecht komt.
- d. Tik de body van de methode 'Main' in zoals in het voorbeeldprogramma 'Hallo2' uit het collegediktaat. Je merkt waarschijnlijk tijdens het typen dat Visual Studio je suggesties geeft voor methode-namen. Je kunt hier handig gebruik van maken zodat je niet bijvoorbeeld `WriteLine` helemaal hoeft uit te schrijven.
- e. De gebruiker van het programma moet nu op de Enter-toets drukken vanwege de `ReadLine` aan het eind van het programma. Verander dit zo dat de gebruiker op een willekeurige toets kan drukken.
Hint: je hebt hiervoor een andere methode van de klasse `Console` nodig. Zet de cursor op het woord `Console` en druk op F1. Je krijgt dan een hulp-pagina over de klasse `Console`. Scroll een flink stuk omlaag en zoek de benodigde methode.
- f. Gebruik de hulp-pagina van de klasse `DateTime` om uit te zoeken hoe je de huidige datum kunt vinden. Breid het programma uit zodat de datum op het scherm wordt geprint. (Welke properties zijn er nodig, en welke daarvan zijn er static? Hoe wordt het static-zijn in de Help aangegeven?)
- g. Breid het programma uit zodat het de gebruiker om een geboortjaar vraagt. Print daarna een schatting van de leeftijd van de gebruiker. Het hierbij benodigde converteren van een string naar een getal heet `Parse`. Zoek een geschikte methode daarvoor.

4. Windows Application

- a. Maak een nieuw project, ditmaal van het type 'Windows Forms App (.NET Framework)'. Druk op F7 om de sourcecode te bekijken, en tik het programma 'HalloWin2' in (of kopieer het uit de voorbeelddirectory).

- b. Welke properties hebben een **Form** en een **Label** nog meer? Pas er een paar aan in het programma.
- c. Welke controls zijn er nog meer behalve **Label**? Zet er een paar op het scherm.

5. Fouten

- a. Introduceer een paar fouten in het programma (bijvoorbeeld: spelfout in een methode-naam, spelfout in een variabele-naam, puntkomma vergeten, aanhalingstekens vergeten, haakje vergeten, accolade vergeten). Welke soorten fouten herkent Visual Studio allemaal, en wat gebeurt er als je dubbelklikt op een fout?
- b. Bij een aanroep van een statische methode of property staat er een klassenaam voor de punt, bij een niet-statische methode een object. Hoe klaagt de compiler als je dit verwisselt?
- c. Maak een nieuw project aan met het template 'Windows Forms Application'. Verken de structuur van het programma.

6. Teken

- a. Maak een nieuw project aan, en zet er de tekst van het programma 'HalloWin3' in.
- b. Pas het programma aan zo dat, in plaats van de tekst, er een *Smiley* getekend wordt (en gele cirkel met een zwart randje, twee ogen en een lachende mond). Precieze vorm en kleur naar eigen smaak. Zoek in Help met welke methode je de mond kunt tekenen. Gebruik variabelen zo dat je nog wat kunt experimenteren met de positie en afmeting van het plaatje.
- c. Verander de waarde van de variabele die de positie van het plaatje vastlegt. Zitten de ogen nog op de goede plek? Zo nee, pas het programma aan zo dat dat wel vanzelf goed gaat.

7. Methodes

- a. Maak een nieuw project met een kopie van je Smiley-programma.
- b. Verander de code zo dat er een methode komt om een smiley te tekenen. Geef de methode parameters voor relevante eigenschappen, zoals de positie en grootte van de smiley. Maar geef het geen onnodige parameters!
- c. Roep je methode een paar keer aan met verschillende waarden voor de parameters.
- d. Maak ook parameters voor de kleur van het gezicht en de kleur van de ogen, en voor de inkleurings-stijl van het gezicht. Gebruik diverse waardes bij het aanroepen van de methode.
- e. Maak ook een parameter voor de 'vrolijkheid' van de smiley: hoe groter het getal, hoe vrolijker. Met een negatief getal moet het een *frowney* worden.

8. Checksum

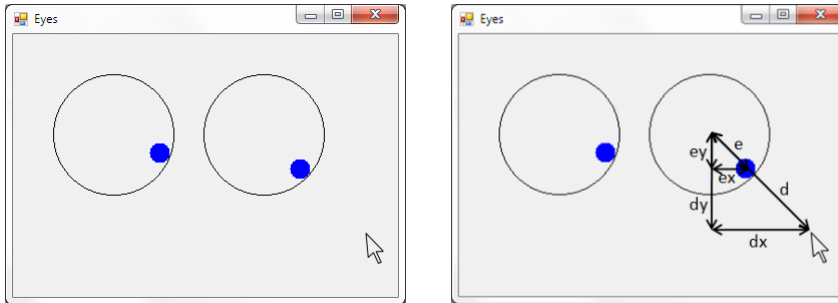
- a. Bekijk je eigen 7-cijferige studentnummer. Vermenigvuldig het eerste cijfer met 7, het tweede cijfer met 6, enzovoort tot het laatste cijfer met 1, en tel alle resultaten op. Het resultaat is, als het goed is, deelbaar door 11. Zo niet, dan is het geen geldig studentnummer, of heb je een rekenfout gemaakt. (Dit werkt niet voor F-nummers).
- b. Schrijf een (statische) methode met een studentnummer als parameter, waarmee gecontroleerd kan worden of een studentnummer klopt.
- c. Maak een compleet programma dat de gebruiker om een studentnummer vraagt, en dan terugmeldt of dit nummer klopt. Je kunt er een console-application van maken, of kijk in hoofdstuk 5 en maak er een variant van het 'CirkelCalc'-programma van.

9. Muis volgen

Schrijf een programma dat een smiley tekent op de plek waar de muis staat. Het plaatje 'volgt' dus de muis. Hint: geef een invulling aan het **MouseMove** event van het window. Zet in die methode op het eind een aanroep van **Invalidate**, wat er voor zorgt dat het window opnieuw getekend wordt.

10. Eyes

Schrijf een programma dat twee grote cartoon-achtige ogen tekent (desgewenst aangevuld tot een complete smiley, maar dan wel met grote ogen, want je moet de pupil duidelijk lost kunnen zien van de oogbol). De pupillen van de ogen moeten worden getekend aan de rand van de oogbol, zo dat de ogen ‘kijken’ in de richting van de muis.



Hint 1: maak een methode die verantwoordelijk is voor het tekenen van één oog, en roep die twee keer aan.

Hint 2: kijk naar de twee driehoeken in het rechter plaatje hierboven. Het midden van de oogbol en de positie van de muis zijn bekend. Daarmee kun je dan ook dx en dy bepalen, en vervolgens met Pythagoras ook d . De afstand e is een vaste lengte, die alleen afhangt van de stralen van oogbol en pupil. Uit de verhouding van e en d , kun je nu de lengte ex bepalen als diezelfde fractie van dx , en evenzo ey . Daarmee heb je dus de afstand van het midden van de pupil ten opzichte van het midden van de oogbol.

C.1 Mandelbrot

Mandelgetallen

Voor elk punt (x, y) van het platte vlak, waarbij x en y reële getallen zijn, kan een bijbehorend getal worden bepaald – laten we dit het ‘mandelgetal’ noemen. Om het mandelgetal te kunnen uitrekenen, bekijken we eerst de volgende functie, die punten (a, b) van het vlak transformeert naar andere punten:

$$f(a, b) = (a * a - b * b + x, 2 * a * b + y)$$

Let op: deze functie transformeert het punt (a, b) , maar in de berekening speelt ook de waarde van x en y , dat is het punt waarvan we het mandelgetal willen bepalen, een rol.

Deze functie f nu, passen we toe op het punt $(a, b) = (0, 0)$. Op het punt dat daar uitkomt, passen we nog eens de functie f toe. Op het punt dat daar weer het resultaat van is, passen we opnieuw f toe, enzovoorts enzovoorts. We stoppen pas met toepassen van f als het resultaat-punt een afstand van meer dan 2 tot het punt $(0, 0)$ heeft. Het mandelgetal is nu gelijk aan het aantal keren dat f is toegepast.

Voor sommige punten (x, y) is dat meteen al zo, en is het mandelgetal dus gelijk aan 1. Voor andere punten duurt het langer: die hebben een groter mandelgetal. Er zijn ook punten waarbij je f kan blijven toepassen, zonder dat de afstand tot het punt $(0, 0)$ ooit meer dan 2 wordt: die punten hebben mandelgetal oneindig.

In de praktijk kunnen we niet oneindig vaak een functie toepassen; daarom stoppen we met toepassen van f bij een bepaald maximum aantal keren, zeg 100 keer; is dan de afstand nog niet groter dan 2, dan nemen we maar aan dat het mandelgetal oneindig is.

Een voorbeeld

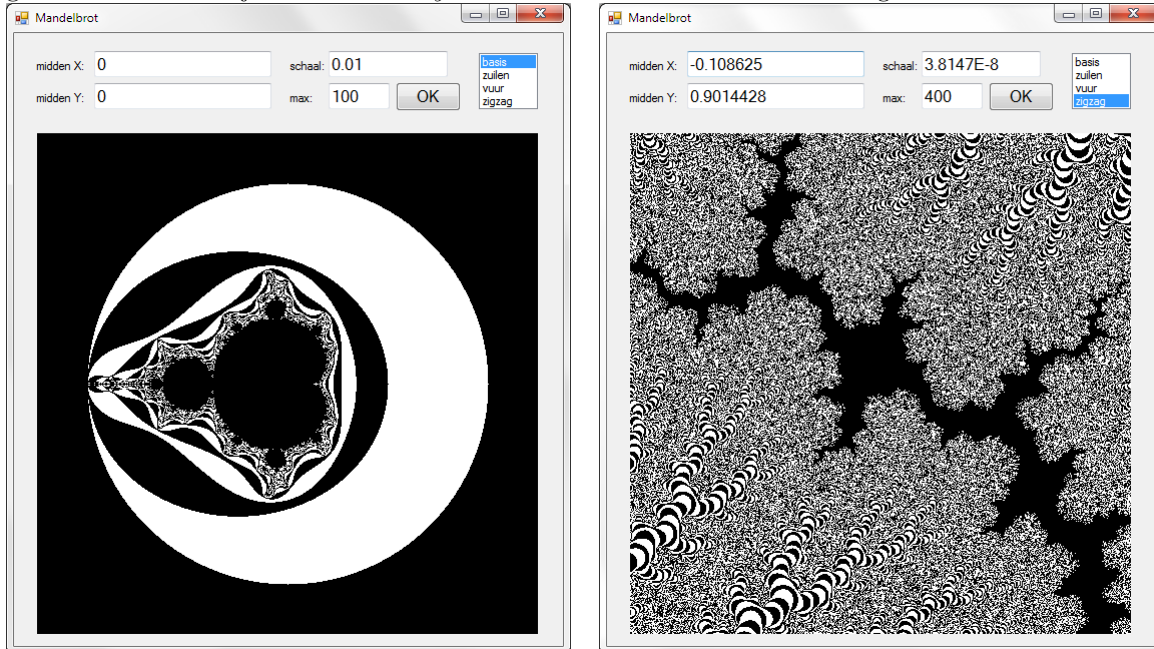
Als voorbeeld berekenen we het Mandelgetal van het punt $(0.5, 0.8)$. We beginnen met $(a, b) = (0, 0)$ en zetten de teller op 0. Bij het toepassen van de formule zijn a en b nog 0, en komen we dus uit op het punt $(0.5, 0.8)$. De teller staat nu op 1. De afstand van $(0.5, 0.8)$ tot $(0, 0)$ is volgens Pythagoras $\sqrt{0.25 + 0.64} = \sqrt{0.89} \approx 0.94$, en dat is niet groter dan 2. We mogen dus doorgaan.

We berekenen met de formule de nieuwe a : $0.5 * 0.5 - 0.8 * 0.8 + 0.5 = 0.11$ en de nieuwe b : $2 * 0.5 * 0.8 + 0.8 = 1.6$, en de teller komt op 2. De afstand van $(0.11, 1.6)$ tot $(0, 0)$ is volgens Pythagoras ongeveer 1.6038, dus we mogen nog steeds doorgaan.

We berekenen met de formule weer een nieuwe a : $0.11 * 0.11 - 1.6 * 1.6 + 0.5 = -2.0479$ en een nieuwe b : $2 * 0.11 * 1.6 + 0.8 = 1.152$. De afstand van $(-2.0479, 1.152)$ tot $(0, 0)$ is royaal meer dan 2, dus de teller blijft staan op 3. Het mandelgetal van $(0.5, 0.8)$ is dus 3.

De Mandelbrot-figuur

Aan de hand van mandelgetallen kunnen we gemakkelijk een kleur van elk punt bepalen. Een mogelijkheid is deze: punten met een even mandelgetal worden wit, punten met een oneven mandelgetal worden zwart. Ook punten met mandelgetal oneiding worden zwart. Geef je elk punt van het scherm de aldus bepaalde kleur, dan zie je een afbeelding van de Mandelbrot-figuur. Het interessante gedeelte van de figuur zit in het gedeelte voor x en y tussen -2 en 2 (daarbuiten is het mandelgetal 2 en het plaatje dus zwart). We moeten dus een beetje inzoomen om de figuur goed te kunnen bekijken. Voor x en y tussen -2.5 en 2.5 ziet die er als volgt uit:



De figuur is opmerkelijk grillig, zeker als je bedenkt hoe relatief eenvoudig het algoritme eigenlijk is. Het wordt nog mooier als je “inzoomt” op onderdelen van de figuur, zoals in het tweede plaatje.

De userinterface

We gaan een programma maken waarmee de gebruiker kan inzoomen op de Mandelbrotfiguur. In het window staan behalve de figuur vier tekstboxes. De gebruiker kan daarin intikken:

- De x -coördinaat van het middelpunt van het scherm
- De y -coördinaat van het middelpunt van het scherm
- De schaalfactor. Bij een schaalfactor van bijvoorbeeld 0.01 kan in een window van 400 bij 400 beeldpunten een gebied voor x en y tussen bijvoorbeeld -2 en 2 worden weergegeven.
- Het maximum aantal herhalingen van f

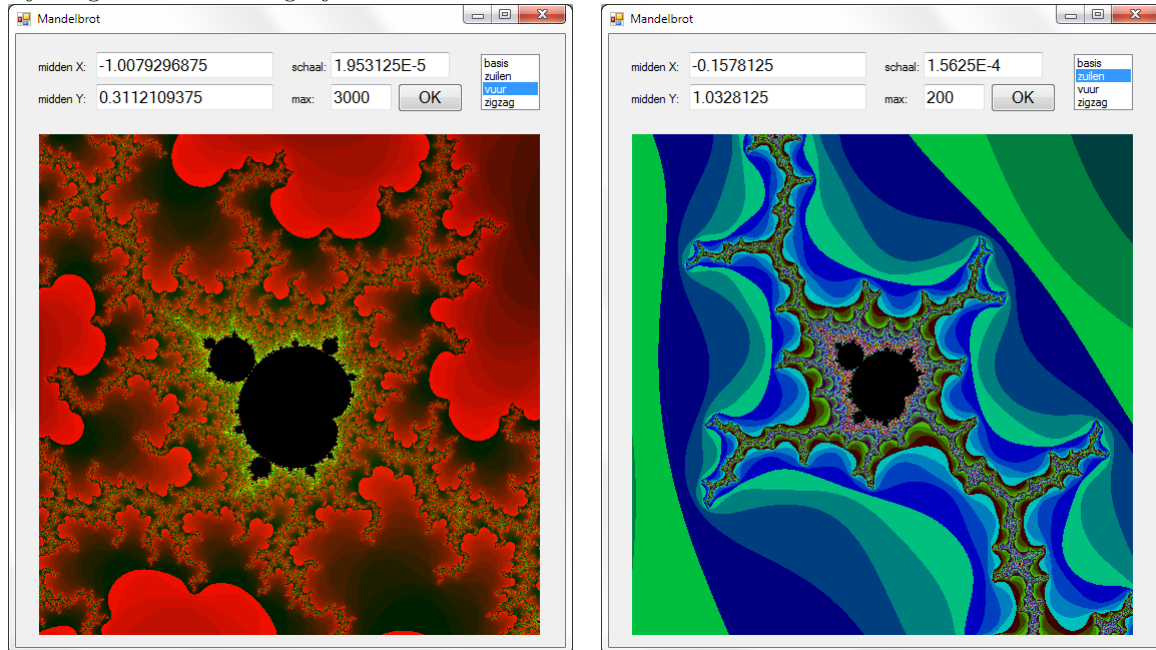
Drukt de gebruiker daarna op de OK-button, dan moet het daarin aangegeven deel van de Mandelbrotfiguur worden getoond. De userinterface hoeft er niet precies zo uit te zien als in het voorbeeld. In plaats van een OK-button mag je ook andere manieren gebruiken om de invoer te bevestigen. Je mag kiezen of je de interface maakt met hulp van Visual Studio's visuele designer, of dat je hem helemaal zelf uitprogrammeert.

Verdere interactie

Ook kan de gebruiker ook met de muis op de figuur klikken. Het aangeklikte punt wordt het nieuwe middelpunt, en de schaal wordt twee keer zo klein; de gebruiker zoomt daarmee in op het aangeklikte punt. De nieuwe waarden van midden en schaal worden ook in de tekstvelden weergegeven. Verder is er nog een control waarmee de gebruiker kan kiezen uit een aantal voorbeeldplaatjes. De tekstvelden worden dan automatisch ingevuld en het bijbehorende plaatje getoond. Kies zelf een aantal mooie details, waaronder het basisplaatje. Het basisplaatje moet ook zichtbaar zijn als het programma begint.

De kleuren

Hierboven is een zeer eenvoudige kleuring van de figuur beschreven: punten met even mandelgetal worden wit, andere punten zwart. Daar kun je het programma in eerste instantie mee testen. Maar in het definitieve programma moeten de punten gekleurd worden, waarbij de rood-, groen- en blauw-component alledrie van het mandelgetal afhangen. Hier zijn twee voorbeelden, maar er zijn nog vele andere mogelijkheden. Wees eens creatief!



Zorg er voor dat de figuur goed zichtbaar blijft, ook bij andere keuzes van het maximum aantal herhalingen.

Optionele extra (voor als je nog tijd over hebt): je kunt de kleurbepaling wellicht nog laten afhangen van extra, door de gebruiker te bepalen parameter(s). Daarvoor kun je eventueel nog extra interactiecomponenten toevoegen.

Beoordelingspunten

Behalve op de werking wordt het programma ook beoordeeld op grond van stijlkenmerken in het programma, zoals:

- Het zinvol gebruiken van methoden en parameters
- Het gebruik van duidelijke namen voor methoden en variabelen
- Een overzichtelijke en consequente layout
- Het opnemen van commentaar bij belangrijke passages
- Het vermijden van onnodig ingewikkelde of tijdrovende algoritmes
- Het gebruiksgemak van de userinterface

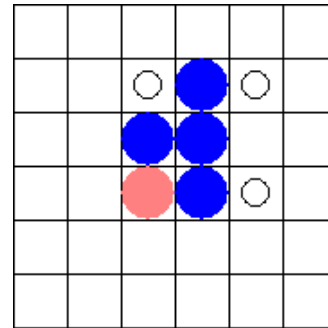
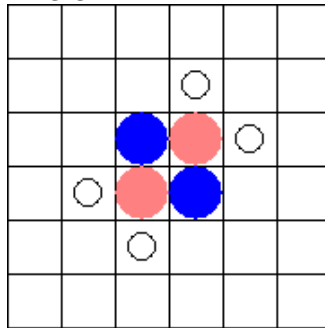
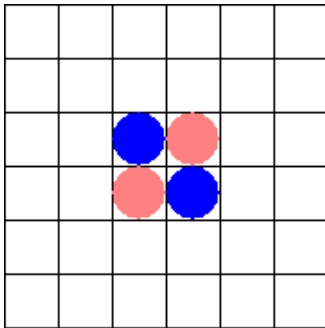
C.2 Reversi-spel

Het spel

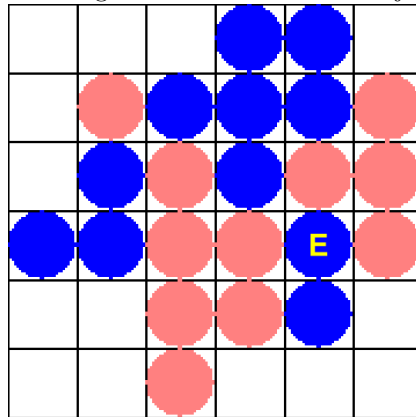
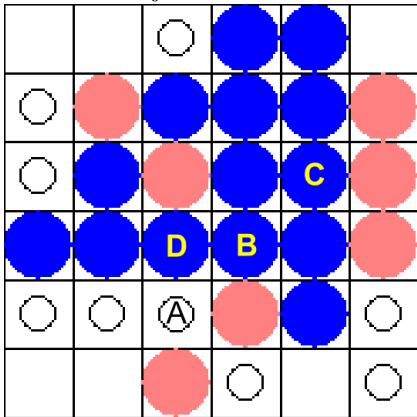
Het spel wordt gespeeld op een bord met vierkante velden. De twee spelers kunnen de beurt een blauwe, respectievelijk een rode steen neerzetten op een vrij veld. Aan het begin van het spel liggen er al twee blauwe (zwarte) en twee rode (grijze) stenen op het bord, volgens het volgende patroon (hier getoond op een bord met 6×6 velden, maar het spel kan ook gespeeld worden op een bord met andere afmetingen).

Spelers zijn niet helemaal vrij in hun keuze waar ze een steen kunnen zetten. Blauw mag een steen alleen maar neerzetten, als daarmee een of meer rode stenen worden ingesloten tussen de nieuwe steen en een van de al op het bord liggende blauwe stenen. Het insluiten kan horizontaal, vertikaal of diagonaal gebeuren. Bijvoorbeeld, in de beginsituatie heeft blauw (zwart), die als eerste aan zet is, de keus uit de velden zoals hiernaast met een rondje aangegeven.

Door het doen van de zet veranderen alle stenen van de tegenspeler die met die zet worden ingesloten in de eigen kleur. Bijvoorbeeld, als blauw in de beginsituatie voor de “bovenste” van de vier mogelijkheden kiest, dan wordt de ingesloten, direct daaronder liggende rode steen blauw. Daarna is rood (grijs) aan zet, die drie mogelijkheden heeft om een blauwe steen in te sluiten.



Na nog een tijdje spelen kan de situatie ontstaan zoals hier links onder getekend is. Rood is aan zet, en kiest bijvoorbeeld voor het veld aangegeven met de letter A. De twee blauwe stenen er diagonaal rechtsboven (aangegeven met de letters B en C) worden ingesloten, maar ook de blauwe steen er recht boven (aangegeven met de letter D). Die worden dus allemaal rood, en de eindsituatie is zoals hier rechts onder getekend is. Merk op dat als gevolg van het veranderen van kleur weer andere blauwe stenen ingesloten raken (zoals die aangegeven met de letter E), maar die veranderen niet van kleur: bij het insluiten moet de nieuw neergezette steen betrokken zijn.



In zeldzame gevallen kan het gebeuren dat een speler geen zet kan doen, omdat er geen enkele mogelijkheid is om stenen van de andere partij in te sluiten. De speler moet dan passen, en de andere speler is weer aan de beurt. Kan ook de andere speler geen zet doen, dan is het spel afgelopen.

Winnaar is aan het eind degene met de meeste stenen op het bord (bij gelijk aantal is de uitslag “remise”).

Het programma

Het programma moet een userinterface hebben dat de volgende dingen toont:

- Het bord met de huidige situatie
- De huidige stand: een indicatie van het aantal stenen van rood en blauw
- De status: ‘blauw aan zet’, ‘rood aan zet’, ‘blauw is winnaar’, ‘rood is winnaar’, of ‘remise’
- Twee knoppen: ‘nieuw spel’ en ‘help’

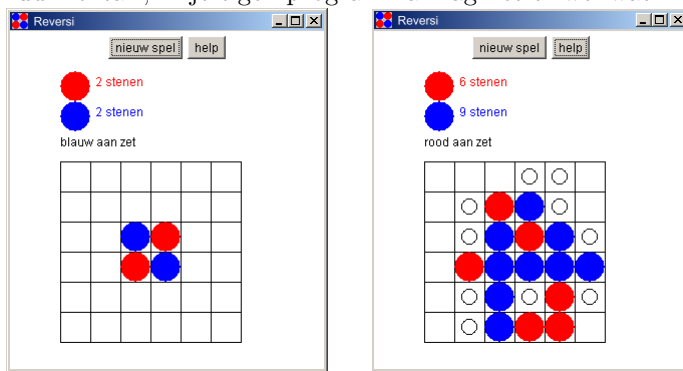
Standaard moet het bord 6×6 velden bevatten, maar aantal rijen en het aantal kolommen van het bord moet door het aanpassen van twee constante-declaraties in het programma gewijzigd kunnen worden (minimaal 3×3 , anders is het spel meteen afgelopen!). Het bord kan ook langwerping worden.

De beginstand staat altijd in het midden van een bord, of (bij een oneven aantal rijen of kolommen) vlakbij het midden.

Met de knop ‘nieuw spel’ kan de gebruiker op elk moment een nieuw spel starten. Door het indrukken van de ‘help’-knop geeft het programma aan op welke velden een zet gedaan mag worden. Nogmaals indrukken van de ‘help’-knop schakelt deze functie weer uit.

Door met de muis op een veld van het bord te klikken kan de gebruiker een zet doen voor de partij die aan de beurt is. Het moet natuurlijk wel een toegestane zet zijn, anders wordt de zet niet geaccepteerd. (De twee spelers zullen dus om de beurt de muis moeten bedienen. Je hoeft niet ‘tegen de computer’ te kunnen spelen).

De afbeeldingen hieronder tonen een voorbeeld hoe het scherm er uit kan zien aan het begin, en na een tijdje met aanschakelen van de help-functie. De layout in deze voorbeelden is tamelijk rudimentair; in je eigen programma mag het er wel wat mooier uitzien.



Hints

- Geef namen aan de constanten die je arbitrair kiest, zoals de diameter van de stenen.
- Beperk het aantal object-variabelen tot een minimum: naast de constanten en de buttons zijn alleen variabelen nodig die de ‘momentele’ toestand van het spel beschrijven.
- Maak methoden (met parameters!) voor spel-specifieke vragen en handelingen, zoals ‘is een zet op een bepaald veld voor een bepaalde kleur legaal?’ en ‘verander de door een bepaalde zet in een bepaalde richting ingesloten stenen van kleur’.
- Bedenk dat er in acht richtingen ingesloten kan worden: alle combinaties van -1 , 0 en $+1$ in de x - en y -richting behalve $(0,0)$.
- Vermijd het om de acht richtingen allemaal apart te behandelen: als er dan een foutje in zit, moet je die acht keer herstellen, en bovendien maakt het het programma onoverzichtelijk. Probeer de regelmaat liever in een methode met handig gekozen parameters te vangen.
- Als je niet weet hoe te beginnen: teken eerst een (flexibel ingevuld) bord, voeg dan de interactie toe maar nog zonder controle van legaliteit en veranderen van kleur van ingesloten stenen, maak dan de help-functie (daarvoor heb je een methode voor controle op legaliteit nodig, die je daarna ook mooi kunt gebruiken voor het controleren van de zetten), enz.

C.3 SchetsPlus

Anders dan de vorige opdrachten gaat het dit keer om de uitbreiding van een bestaand programma. Je hoeft dus niet *from scratch* te beginnen, maar kunt het programma ‘Schets’ uit hoofdstuk 10.5 als uitgangspunt gebruiken. Dat bestaat uit 6 C#-files, waarin 11 klassen, 2 abstracte klassen en een interface worden gedefinieerd.

Sommige van die klassen kun je onveranderd laten. In sommige klassen zul je methoden moeten toevoegen, of bestaande methoden veranderen (extra opdrachten en/of extra parameters). Ook zul je waarschijnlijk nog enkele klassen moeten toevoegen. Om het voor anderen mogelijk te maken snel een overzicht te krijgen van de uitbreidingen die je hebt gedaan, moet je bij het programma ook een overzicht van de gedane wijzigingen maken. Dat is een tekstfile, waarin je opsomt in welke methoden van welke klassen er iets is veranderd, waar zinvol met een korte motivatie waarom deze verandering noodzakelijk was. De toelichting mag een Word- (.doc of .docx) of PDF-bestand zijn. De file die je inlevert moet een ZIP- of RAR-file zijn, met daarin alle source-bestanden (de ongewijzigde en de gewijzigde en de toegevoegde), resource-bestanden (de ongewijzigde en de gewijzigde en de toegevoegde), en de project-file (VS2008 of VS2010, maar niet allebei). Gegeneerde binaries en executables (de bin- en obj-directories) hoeven niet te worden meeverpakt.

Beoordelingsnormen

Bij de beoordeling van het programma wordt natuurlijk gelet op correcte werking. Maar daarnaast wordt ook gelet op de stijl van het programma. In het bijzonder gaat het ditmaal om het aanhouden van het *once and only once* principe, dus het vermijden van het kopiëren en plakken van stukken code. Het gebruik van methoden met geschikte parameters, en het gebruik van overerving in subklassen (inheritance) kunnen daarbij helpen. Ook het zinvol inzetten van klassen uit de C#-bibliotheek (de Collection-hierarchie, File-I/O enz.) kan leiden tot het vermijden van dubbel werk en draagt bij aan de duidelijkheid van het programma; dit wordt dus positief gewaardeerd. Bovendien geldt het als fraai als variabelen waar mogelijk lokaal worden gedeclareerd en/of als parameter doorgegeven aan methoden, zodat de declaraties bovenin de klasse beperkt blijven tot variabelen die inderdaad essentieel zijn voor de toestand van het object. Tenslotte wordt ook de duidelijkheid van de toelichtings-tekst wordt ook de beoordeling betrokken.

Als de onderdelen 1 t/m 4 hieronder goed werken, de code is netjes opgebouwd en een duidelijke beschrijving is toegevoegd, dan kun je het cijfer 8 halen. Voor een hoger cijfer kun je één of meer van de extra's onder punt 5 inbouwen.

1. Cirkels

Voeg twee tools toe voor het tekenen van een open en een gevulde ovaal. De nieuwe tools zijn natuurlijk zowel via de toolbox, als via het Tool-menu beschikbaar. Hint: laat je inspireren door de manier waarop de rechthoeken in het bestaande programma worden aangepakt.

2. Opslaan en teruglezen

Maak nieuwe menu-items om het plaatje op te slaan en weer in te lezen. De file moet worden opgeslagen in een bitmap-formaat, waarbij de gebruiker uit een aantal gangbare filetype typen moet kunnen kiezen (bijvoorbeeld PNG, JPG en BMP).

Zorg er bovendien voor dat de gebruiker een waarschuwing krijgt als hij een schets-window afsluit terwijl er wijzigingen zijn gedaan sinds de laatste keer dat het plaatje werd opgeslagen of ingelezen.

3. Het nieuwe gummen

De gum-tool werkt in het basisprogramma door eigenlijk door het tekenen van een dikke witte lijn over de bestaande tekening heen, waardoor het lijkt alsof die verdwijnt. We gaan het gedrag van de gum nu veranderen, zo dat de gebruiker een getekend element (bijvoorbeeld een rechthoek) in n keer kan uitwissen, door het met de ‘nieuwe gum’ aan te klikken. Niet alleen verdwijnt zo’n element dan, maar ook komen andere elementen die geheel of gedeeltelijk door het weggehaalde element werden bedekt weer tevoorschijn. (Met de ‘oude gum’ was dat niet mogelijk: als je daarmee een rechthoek wegveegde, bleef er slechts een witte vlek over).

Het is niet nodig om een nieuwe icon te maken; je kunt simpelweg de functionaliteit van de gum veranderen. De oude manier van gummen komt gewoon te vervallen.

Aanpak: Vanwege de mogelijkheid dat bedekte elementen weer tevoorschijn kunnen komen, is het noodzakelijk om ook ‘onzichtbare’ (want bedekte) elementen nog te onthouden. Met een bitmap, zoals in het basisprogramma, is dat niet mogelijk: weg is weg. In theorie is het mogelijk om bij

elk getekend element een kopie van de oude bitmap te maken, althans van het bedekte gedeelte. Dat vreet echter wel erg veel geheugen, en het is dan ook niet de aanpak die hier wordt gevraagd. Een betere aanpak is om de hele tekening, behalve als bitmap, ook nog te bewaren in de vorm van een lijst van getekende elementen. De objecten in die lijst vormen een compacte beschrijving van elk getekend element (soort, beginpunt, eindpunt, kleur, eventuele tekst). Bij elke teken-actie wordt de lijst uitgebreid. De bitmap uit het basisprogramma mag ook blijven, en voldoet prima voor de meeste teken-akties. De ‘nieuwe gum’-actie werkt echter door het te verwijderen element uit de lijst te halen, en aan de hand van die lijst de hele bitmap te reconstrueren, door alle overgebleven elementen in de lijst opnieuw te tekenen.

Vrijheid: Wat je precies als elementen beschouwt, die met één klik kunnen worden uitgewist, mag je zelf bedenken. Het ligt voor de hand dat een rechthoek en een ovaal in zijn geheel als één element te beschouwen. Maar of je een met de pen-tool gemaakte kromme lijn als één element, of als allemaal losse lijntjes beschouwt, mag je zelf kiezen. Hetzelfde geldt voor een tekst: is dat in zijn geheel één element, of is elke aparte letter een apart element? (Zet de gemaakte keuze in de toelichtings-tekst!)

Hints:

- Het model van de tekening zal moeten worden uitgebreid: naast de al bestaande bitmap komt de elementen-lijst erbij, met een methode die de bitmap uit de elementenlijst kan reconstrueren.
- Het is ook niet raar als de methode die een figuur tekent ingrijpend aangepast moet worden aan het aldus gewijzigde model.
- Wanneer heeft een gebruiker ‘raak’ geklikt op een element van de tekening? Bij een gevulde rechthoek is dat duidelijk. Voor de overige elementen kun je in eerste instantie het programma uittesten door alles binnen de *bounding box* van zo’n figuur als ‘raak’ te beschouwen. Maar dat kan natuurlijk ook subtieler. Bij de gevulde cirkel kun je met een Pythagoras-achtige formule zorgen dat daadwerkelijk het gekleurde deel van de cirkel aangeklikt moet worden. En bij een open rechthoek? Het hele binnengebied als ‘raak’ beschouwen is wel erg royaal. Maar om precies de rand te raken moet de gebruiker wel erg goed mikken. Je zou ervoor kunnen zorgen dat een zone van -zeg- vijf pixels rond de rand als ‘raak’ telt. Licht hoe dan ook de gemaakte keuze toe in de toelichtings-tekst.
- Wanneer is er raak geklikt op een schuine lijn? Nou, bijvoorbeeld als de afstand van het geklikte punt tot de lijn niet te groot is. Hoe bepaal je de afstand van een punt tot een lijn? Tik ‘distance point line’ in op Google, en je bent al snel op een pagina die de formule daarvoor geeft. Laat je niet afschrikken door de wiskunde eromheen: de benodigde formule staat er gewoon tussen. Vermeld de gebruikte bron in de toelichting en in commentaar in het programma!
- Het zou handig zijn als er een methode is die bepaalt of een punt ‘raak’ is voor een bepaald type element. Die kun je in eerste instantie simpel houden, om snel te kunnen testen of de rest van het programma werkt; daarna kun je de ‘raak’-methode naar believen subtieler maken.

4. Het nieuwe Opslaan en teruglezen

Als de gebruiker een ingewikkelde tekening heeft gemaakt, wil hij daar misschien later weer mee verder werken, waarbij dan nog steeds de elementen als objecten te verwijderen zijn.

Daartoe is het nodig dat bij het opslaan/inlezen ook gekozen kan worden uit een speciaal ‘schets’-formaat. Het plaatje wordt nu dus *niet* als bitmap opgeslagen, maar op een manier (bijvoorbeeld als tekstfile) waaruit de opbouw van het plaatje gereconstrueerd kan worden: ‘lijntje hier, rechthoek daar, tekstje in het midden’.

Hint: De file mag een gewone tekst-file zijn, bijvoorbeeld met voor elk element een regel. De vorm van de file mag je helemaal zelf bepalen: enige vereiste is dat je eigen programma een geschreven file weer kan inlezen. Je hoeft je dus niet te conformeren aan bestaande standaarden. (Een voorbeeld kun je vinden in de manier waarop ‘Steden’ en ‘Wegen’ in een file worden opgeslagen in het Zoeknetwerk-programma in hoofdstuk 11.3).

Als de onderdelen 1 t/m 4 hierboven goed werken, de code is netjes opgebouwd en een duidelijke beschrijving is toegevoegd, dan kun je het cijfer 8 halen. Voor een hoger cijfer moet je één of meer van onderstaande extra’s inbouwen. (Beschrijf de gemaakte keuzes ook in de toelichting).

5. Extra's




















































1. Voeg een 'Undo'-knop toe aan het control-panel en/of een menu. Daarmee kan de gebruiker het laatst getekende element weer weghalen, en door herhaaldelijk gebruik de hele tekening weer langzaam afbreken. Misschien kun je zelfs ook nog een 'Redo'-knop maken, die de weggehaalde elementen toch weer toevoegt. Hint: In de bitmap-representatie is dit vrijwel onmogelijk, maar met behulp van de elementen-lijst, die voor onderdeel 2 sowieso nodig is, is het tamelijk eenvoudig.
2. Maak naast de kleur-control ook een control waarmee de gebruiker de lijndikte kan instellen. (En als je dan ook nog kleur 'Wit' aan de kleur-control toevoegt, hebben we de mogelijkheid van de oude gum terug...)
3. Zorg dat de gebruiker uit veel meer kleuren dan de 7 standaardkleuren kan kiezen.
4. Maak een extra tool waarmee je een aangeklikt element 'bovenop', of juist 'onderop' de stapel van getekende elementen legt. Een half-bedeekte rechthoek kun je daarmee dus weer helemaal zichtbaar maken.
5. Maak een extra tool waarmee je een element kunt aanklikken, waarna je het door te 'drijven' naar een andere plaats kunt slepen. Uiteraard kunnen daarbij achterliggende elementen verschijnen of juist bedekt worden.
6. Als je het window groter of kleiner maakt, reageert het basisprogramma door de bitmap van de tekening groter of kleiner te maken. De getekende elementen blijven echter even groot. Je zou het echter ook zo kunnen maken, dat de getekende elementen meegroeien, respectievelijk krimpen, met de bitmap.
7. Verzin zelf een zinvolle uitbreiding van het programma.

Bijlage D

Class library Imperatief programmeren

D.1 System

class **String** // alias voor string

	int	Length	
	char	this	[int]
	 operator	+	(string, string)
	 operator	==	(string, string)
	 operator	!=	(string, string)
	 static	Empty	
	 static	Compare	(string, string)
	 static	Compare	(string, string, StringComparison)
	 static	Concat	(string, string)
	 static	Copy	(string)
	 static	Equals	(string, string)
	 static	Equals	(string, string, StringComparison)
	 static	IsNullOrEmpty	(string)
	 static	IsNullOrWhiteSpace	(string)
	int	CompareTo	(string)
	bool	Contains	(string)
	bool	EndsWith	(string)
	bool	EndsWith	(string, StringComparison)
	bool	Equals	(string)
	bool	Equals	(string, StringComparison)
	int	IndexOf	(char)
	int	IndexOf	(string)
	int	IndexOf	(string, StringComparison)
	string	Insert	(int, string)
	int	LastIndexOf	(char)
	int	LastIndexOf	(string)
	int	LastIndexOf	(string, StringComparison)
	string	Replace	(char, char)
	string	Replace	(string, string)
	string[]	Split	()
	string[]	Split	(char)
	string[]	Split	(char[])
	bool	StartsWith	(string)
	bool	StartsWith	(string, StringComparison)
	string	Substring	(int)
	string	Substring	(int, int)
	char[]	ToCharArray	()
	string	ToLower	()
	string	ToUpper	()
<i>enum</i> StringComparison		Ordinal, OrdinalIgnoreCase, CurrentCulture, CurrentCultureIgnoreCase	











```

class Object // alias voor object
    virtual bool Equals (object)
    static bool Equals (object, object)
    virtual string ToString ()
struct Int32 // alias voor int
    static int Parse (string)
struct Double // alias voor double
    static double Parse (string)
abstract class Array // methodes werken op arrays
    int Length
    object Clone ()
    int GetUpperBound (int)
struct TimeSpan
    int Milliseconds
    int Seconds
    int Minutes
    int Hours
    int Days
static class Console
    static TextReader In
    static TextWriter Out
    static void Beep ()
    static int Read ()
    static ConsoleKeyInfo ReadKey ()
    static string ReadLine ()
    static void Write (int)
    static void Write (string)
    static void WriteLine ()
    static void WriteLine (int)
    static void WriteLine (string)
struct ConsoleKeyInfo
    char KeyChar
static class Math
    static double E
    static double PI
    static double Abs (int)
    static double Abs (double)
    static double Sin, Cos, Tan (double)
    static double Exp, Log, Log10 (double)
    static double Pow (double, double)
    static double Sqrt (double)
    static double Floor, Ceiling (double)
    static double Truncate, Round (double)
    static double Min, Max (double, double)
    static double Min, Max (int, int)
delegate void Action ()
delegate void Action<A> (A)
delegate void Action<A,B> (A, B)
delegate R Func<R> ()
delegate R Func<A,R> (A)
delegate R Func<A,B,R> (A, B)















```

D.2 System.Drawing


















struct Point

	©	Point	(int, int)
	S	operator bool	== (Point, Point)
	S	operator bool	!= (Point, Point)
	S	operator Point	+ (Point, Point)
		bool	IsEmpty
		int	X
		int	Y
		Point	Offset (Point)
		Point	Offset (int, int)
	S	Point	Empty







struct Color

	S	static Color	FromArgb (int, int, int)
	S	static Color	FromArgb (int, int, int, int)
	S	static Color	FromName (string)
		byte	A
		byte	R
		byte	G
		byte	B
	S	static Color	White, Gray, Black
	S	static Color	Red, Green, Blue
	S	static Color	Yellow, Magenta, Cyan
	S	static Color	Tomato, Fuchsia, Salmon, ...
		float	GetHue ()
		float	GetBrightness ()
		float	GetSaturation ()



struct Rectangle

	©	Rectangle	(int, int, int, int)
	©	Rectangle	(Point, Size)
		int	X, Y
		int	Width, Height, Bottom, Right
		bool	IsEmpty
		Size	Size
		Point	Location
		bool	Contains (Point)
		bool	Contains (Rectangle)
		bool	Intersects (Rectangle)
	S	static Rectangle	Intersect (Rectangle, Rectangle)
		void	Intersect (Rectangle)
		void	Inflate (Size)
		void	Inflate (int, int)
		void	Offset (int, int)
		void	Offset (Point)
	S	static Rectangle	Empty

struct Size

	©	Size	(int, int)
	S	operator Size	+ (Size, Size)
	S	operator bool	==, != (Size, Size)
		bool	IsEmpty
		int	Width, Height
	S	static Size	Empty












class Font


	©	Font	(string, float)
	©	Font	(string, float, FontStyle)

enum FontStyle








Regular, Bold, Italic, Underline, Strikeout

class Graphics

  static Graphics
 void
 void
 void
 void
 void
 void
 void
 void
 void

 SmoothingMode


enum SmoothingMode**class Pen**

 ©
 ©
 ©
 ©
 float
 LineJoin
 DashStyle

enum LineJoin**enum DashStyle****abstract class Brush****class SolidBrush : Brush**

 ©

class TextureBrush : Brush

 ©

class LinearGradientBrush : Brush



 ©



class HatchBrush : Brush

 ©



 ©



static class Pens

  static Pen



  static Pen



static class Brushes

  static Brush


  static Brush

class Image

  static Image

  static Image

 int

 Size

 void

 void

 object

class Bitmap : Image

 ©

 ©

 ©

 void

class ImageFormat

  static ImageFormat

enum RotateFlipType

FromImage (Image)
 DrawString (string, Font, Brush, int, int)
 DrawLine (Pen, int, int, int, int)
 DrawLine (Pen, Point, Point)
 DrawArc (Pen, int, int, int, int, int, int)
 DrawRectangle (Pen, int, int, int, int)
 DrawRectangle (Pen, Point, Size)
 DrawRectangle (Pen, Rectangle)
 DrawEllipse (Pen, int, int, int, int)
 FillRectangle (Brush, int, int, int, int)
 FillEllipse (Brush, int, int, int, int)
 SmoothingMode
 None, AntiAlias

Pen (Brush)
 Pen (Color)
 Pen (Brush, float)
 Pen (Color, float)
 Width
 LineJoin
 DashStyle
 Miter, Bevel, Round, MiterClipped
 Solid, Dash, Dot, DashDot, DashDotDot, Custom

SolidBrush (Color)
 TextureBrush (Image)
 LinearGradientBrush (Point, Point, Color, Color)
 HatchBrush (HatchStyle, Color)
 HatchBrush (HatchStyle, Color, Color)

White, Gray, Black, Red, Green, Blue
 Yellow, Magenta, Cyan, Tomato, Fuchsia, Salmon, ...

White, Gray, Black, Red, Green, Blue
 Yellow, Magenta, Cyan, Tomato, Fuchsia, Salmon, ...

FromFile (String)
 FromStream (Stream)
 Width, Height
 Size
 RotateFlip (RotateFlipType)
 Save (Stream, ImageFormat)
 Clone ()

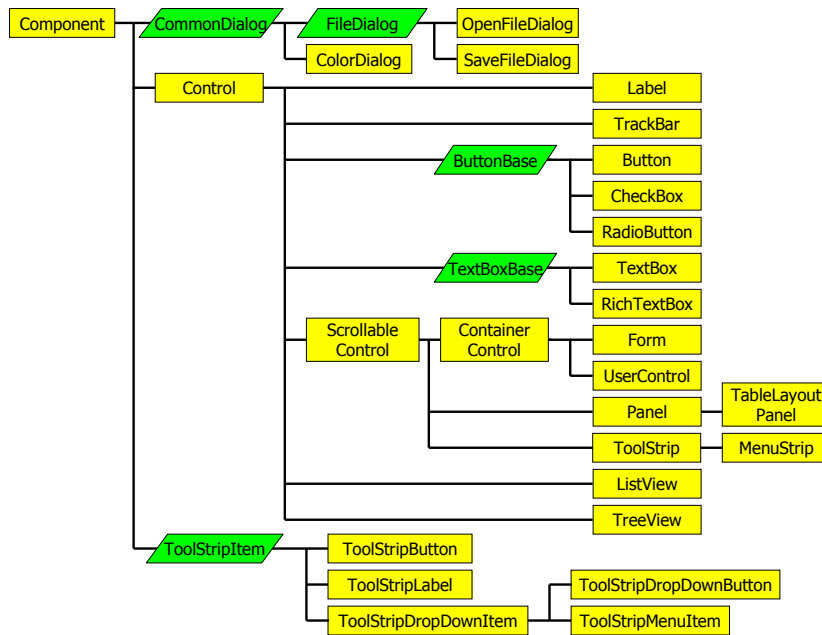
Bitmap (String)
 Bitmap (Image)
 Bitmap (int, int)
 SetPixel (int, int, Color)

Bmp, Gif, Jpeg, Png, Exif, Tiff
 Rotate[None/90/180/270]Flip[None/X/Y/XY]

D.3 System.Windows.Forms

static class **Application**

static void Run (Form)
static void Exit ()



class **Control** : Component

	Color	BackColor, ForeColor	
	Image	BackgroundImage	
	Point	Location	
	Size	Size	
	Rectangle	Bounds	
	int	Top, Bottom, Left, Right	
	string	Text	
	object	Tag	
	Font	Font	
	Control	Parent	
	bool	Visible	
	DockStyle	Dock	
	ICollection<Control>	Controls	
	bool	Focus	()
	void	Invalidate	()
	void	Refresh	()
	Graphics	CreateGraphics	()
	event EventHandler	Click	
	event EventHandler	DoubleClick	
	event EventHandler	Resize	
	event KeyPressEventHandler	KeyPress	
	event PaintEventHandler	Paint	
	event MouseEventHandler	MouseClick	
	event MouseEventHandler	MouseMove	

class **ScrollableControl** : Control

	bool	VScroll, HScroll
	event ScrollEventHandler	Scroll

class **ContainerControl** : ScrollableControl

	bool	Validate	()
--	------	----------	----


```

class Form : ContainerControl
    © Form ()
    Button AcceptButton
    Button CancelButton
    Size ClientSize
    Icon Icon
    bool IsMdiChild
    bool IsMdiContainer
    Form[] MdiChildren
    MenuStrip MainMenuStrip
    bool Modal
    void Close ()
    void ShowDialog ()
class Label : Control
    © Label ()
    Image Image
    ContentAlignment TextAlign
abstract class ButtonBase : Control
class Button : ButtonBase
    © Button ()
class CheckBox : ButtonBase
    © CheckBox ()
    bool Checked
    CheckState CheckState
class RadioButton : ButtonBase
    © RadioButton ()
    bool Checked
abstract class TextBoxBase : Control
    bool AcceptsTab
    string SelectedText
    int SelectionStart
    int SelectionLength
    void Clear ()
    void AppendText (string)
    void Select (int, int)
class TextBox : TextBoxBase
    © TextBox ()
    bool AcceptsReturn
    bool MultiLine
    char PasswordChar
class RichTextBox : TextBoxBase
    © RichTextBox ()
    bool DetectUrls
    Font SelectionFont
    Color SelectionColor
    int Find (string)
    void LoadFile, SaveFile (string)
    void LoadFile, SaveFile (Stream, RichTextBoxStreamType)
    ⚡ event LinkClickedEventHandler LinkClicked
enum RichTextBoxStreamType PlainText, UnicodePlainText, RichText
class TrackBar : Control
    int Minimum
    int Maximum
    int Value
    int TickFrequency
    TickStyle TickStyle
    ⚡ event EventHandler ValueChanged

```

```

class Panel : ScrollableControl
    (C) Panel ()
class TableLayoutPanel : Panel
    (C) TableLayoutPanel ()
    int RowCount, ColumnCount
    ICollection<TableLayoutRowStyle> RowStyles
    ICollection<TableLayoutColumnStyle> ColumnStyles
abstract class TableLayoutStyle
    SizeType SizeType
class TableLayoutRowStyle : TableLayoutStyle
    (C) TableLayoutRowStyle (SizeType)
    (C) TableLayoutRowStyle (SizeType, float)
class TableLayoutColumnStyle : TableLayoutStyle
    (C) TableLayoutColumnStyle (SizeType)
    (C) TableLayoutColumnStyle (SizeType, float)
enum SizeType
    AutoSize, Absolute, Percent

delegate void EventHandler (object, EventArgs)
delegate void PaintEventHandler (object, PaintEventArgs)
delegate void MouseEventHandler (object, MouseEventArgs)
delegate void KeyPressEventHandler (object, KeyPressEventArgs)
delegate void ScrollEventHandler (object, ScrollEventArgs)
delegate void LinkClickedEventHandler (object, LinkClickedEventArgs)
class EventArgs
class PaintEventArgs : EventArgs
    Graphics Graphics
    Rectangle ClipRectangle
class MouseEventArgs : EventArgs
    MouseButton Button
    int Clicks
    Point Location
    int X
    int Y
class KeyPressEventArgs : EventArgs
    char KeyChar
class ScrollEventArgs : EventArgs
    ScrollOrientation ScrollOrientation
    ScrollEventType Type
    int OldValue
    int NewValue
class LinkClickedEventArgs : EventArgs
    string LinkText
enum CheckState
    Unchecked, Checked, Indeterminate
enum TickStyle
    None, TopLeft, BottomRight, Both
enum MouseButton
    None, Left, Middle, Right
enum ScrollOrientation
    HorizontalScroll, VerticalScroll
enum ScrollEventType
    First, Last, ThumbPosition, ThumbTrack, [Small/Large][Inc/Dec]rement
enum DockStyle
    None, Top, Bottom, Left, Right, Fill
enum ContentAlignment
    [Top/Middle/Bottom][Left/Center/Right]
enum DialogResult
    None, OK, Cancel, Yes, No, Abort, Retry, Ignore

```

```

class ToolStrip : ScrollableControl
    (C) ToolStrip ()
    (C) ToolStrip (ToolStripItem[])
    ICollection<ToolStripItem> Items
class ToolStripMenuStrip : ToolStrip
    (C) ToolStripMenuStrip ()
    bool AllowMerge
abstract class ToolStripItem : Component
class ToolStripButton : ToolStripItem
class ToolStripLabel : ToolStripItem
class ToolStripDropDownItem : ToolStripItem
    ToolStripItemCollection DropDownItems
class ToolStripDropDownButton : ToolStripDropDownItem
class ToolStripMenuItem : ToolStripDropDownItem
    (C) ToolStripMenuItem (String)
    (C) ToolStripMenuItem (String, Image)
    (C) ToolStripMenuItem (String, Image, ToolStripItem[])
    (C) ToolStripMenuItem (String, Image, EventHandler)
class ToolStripItemCollection
    void Add (String)
    void Add (String, Image)
    void Add (String, Image, EventHandler)

abstract class CommonDialog : Component
    DialogResult ShowDialog ()
abstract class FileDialog : CommonDialog
    string Filename
    string[] Filenames
    string Title
    string Filter
    int FilterIndex
    bool CheckfileExists, CheckPathExists
class OpenFileDialog : FileDialog
    (C) OpenFileDialog ()
class SaveFileDialog : FileDialog
    (C) SaveFileDialog ()
class ColorDialog : CommonDialog
    (C) ColorDialog ()
    Color Color
    int[] CustomColors

```

D.4 System.Threading









```

class Thread
    (C) Thread (ThreadStart)
    void Start ()
    static void Sleep (int)
    static void Sleep (TimeSpan)
    delegate void ThreadStart ()

```

D.5 System.IO

abstract class **Stream**

	long	Length, Position
	bool	CanRead, CanWrite, CanSeek
	void	Close
	int	ReadByte
	int	Read
	long	Seek
	void	WriteByte
	void	Write

class **FileStream** : Stream

	Ⓒ	FileStream	(String, FileMode)
---	---	------------	--------------------



class **MemoryStream** : Stream

	Ⓒ	MemoryStream	()
	Ⓒ	MemoryStream	(byte[])

class **NetworkStream** : Stream

	Ⓒ	NetworkStream	(Socket)
---	---	---------------	----------

class **BufferedStream** : Stream

	Ⓒ	BufferedStream	(Stream)
	Ⓒ	BufferedStream	(Stream, int)

class **CryptoStream** : Stream

	Ⓒ	CryptoStream	(Stream, ICryptoTransform, CryptoStreamMode)
---	---	--------------	--

class **GZipStream** : Stream

	Ⓒ	GZipStream	(Stream, CompressionMode)
---	---	------------	---------------------------

class **WebClient**

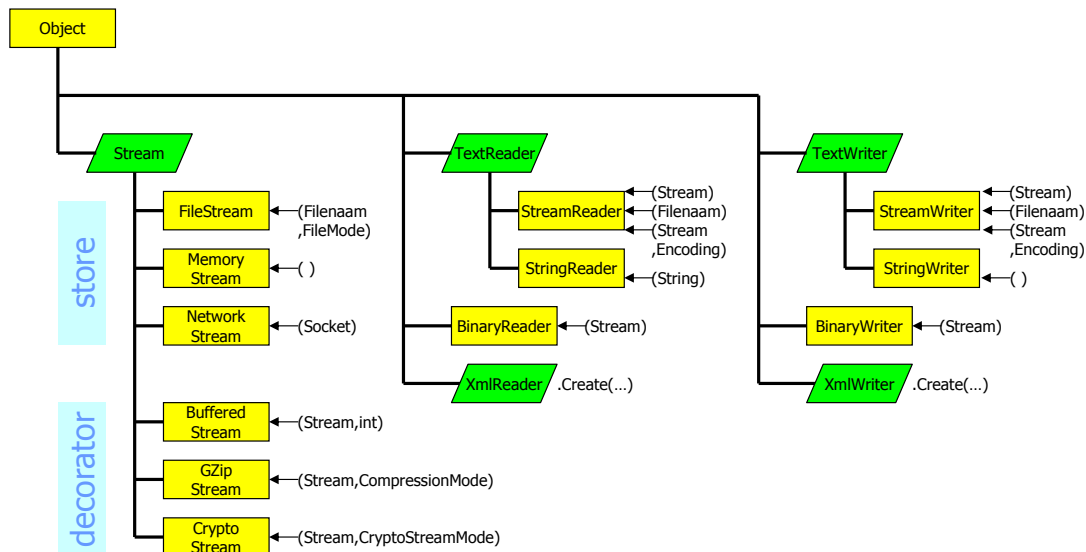
	Ⓒ	WebClient	()
	Stream	OpenRead	(string)

enum **SeekOrigin** Begin, Current, End








enum **FileMode** CreateNew, Create, Open, OpenOrCreate, Truncate, Append

enum **CryptoStreamMode** Read, Write

enum **CompressionMode** Compress, Decompress



abstract class **TextReader**

		TextReader	Null
	int	Peek	
	void	Close	()
	int	Read	()
	string	ReadLine	()
	string	ReadToEnd	()

```

class StreamReader : TextReader
    (C) StreamReader (string)
    (C) StreamReader (Stream)
    (C) StreamReader (Stream, Encoding)
class StringReader : TextReader
    (C) StringReader (string)
class BinaryReader
    (C) BinaryReader (Stream)
    (C) BinaryReader (Stream, Encoding)
    void Close ()
    bool ReadBoolean ()
    byte ReadByte ()
    char ReadChar ()
    short ReadInt16 ()
    int ReadInt32 ()
    long ReadInt64 ()
    float ReadSingle ()
    double ReadDouble ()
abstract class TextWriter
    S static TextWriter Null
    void Close ()
    void Write, WriteLine (char)
    void Write, WriteLine (string)
    void Write, WriteLine (int)
    void Write, WriteLine (double)
    void WriteLine ()
class StreamWriter : TextWriter
    string Newline
    (C) StreamWriter (string)
    (C) StreamWriter (Stream)
    (C) StreamWriter (Stream, Encoding)
class StringWriter : TextWriter
    (C) StringWriter ()
class BinaryWriter
    (C) BinaryWriter (Stream)
    (C) BinaryWriter (Stream, Encoding)
    void Close ()
    void Write (bool)
    void Write (byte)
    void Write (char)
    void Write (string)
    void Write (short)
    void Write (int)
    void Write (long)
    void Write (float)
    void Write (double)
class XmlWriter
    S static XmlWriter Create (TextWriter, XmlWriterSettings)
class Encoding
    S static Encoding ASCII, Unicode, BigEndianUnicode, UTF32, UTF8
    S static Encoding GetEncoding (string)
    S static Encoding[] GetEncodings ()

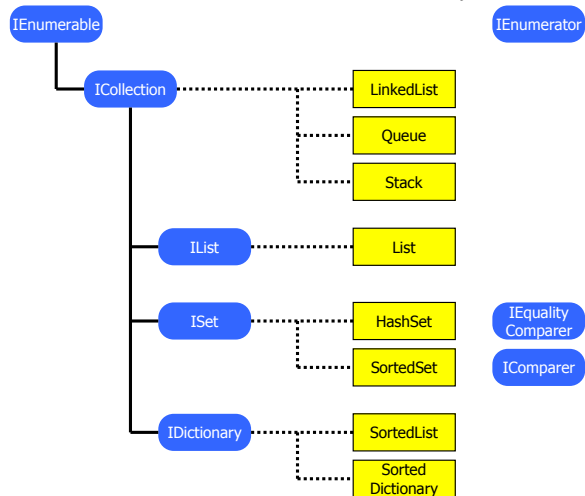
```

D.6 System.Collections.Generic

```

interface IEnumerable<T>
    IEnumerator GetEnumerator      ()
interface ICollection<T> : IEnumerable<T>
    int Count
    bool IsReadOnly
    void Clear                      ()
    void Add                       (T)
    bool Remove                    (T)
    bool Contains                  (T)
    void CopyTo                    (T[], int)
interface IList<T> : ICollection<T>
    T this [int]
    int IndexOf                   (T)
    void Insert                   (int, T)
    void RemoveAt                 (int)
interface ISet<T> : ICollection<T>
    void IntersectWith            (IEnumerable<T>)
    void UnionWith                 (IEnumerable<T>)
    void ExceptWith               (IEnumerable<T>)
    bool IsSubsetOf               (IEnumerable<T>)
    bool IsSupersetOf             (IEnumerable<T>)
    bool Overlaps, SetEquals      (IEnumerable<T>)
interface IDictionary<K,V> : ICollection<KeyValuePair<K,V>>
    ICollection<K> Keys
    ICollection<V> Values
    void Add                      (K, V)
    bool ContainsKey              (K)
    bool Remove                   (K)
    bool TryGetValue              (K, out V)

```



```

struct KeyValuePair<K,V>

```

```

    K Key
    V Value
interface IEnumerator
    object Current
    void Reset      ()
    bool MoveNext   ()
interface IEqualityComparer<T>
    bool Equals      (T, T)
interface IComparer<T>
    int Compare      (T, T)

```

```

class LinkedList<T> : ICollection<T>
    (C)      LinkedList<T>      ()
    (C)      LinkedList<T>      (IEnumerable<T>)
    void     AddFirst, AddLast  (T)
    LinkedListNode<T> Find      (T)
    void     AddBefore, AddAfter (LinkedListNode<T>, T)

class LinkedListNode<T>
    LinkedList<T>      List
    LinkedListNode<T> Next, Previous
    T                  Value

class Queue<T> : ICollection<T>
    (C)      Queue<T>      ()
    (C)      Queue<T>      (IEnumerable<T>)
    void     Enqueue        (T)
    T        Dequeue        ()
    T        Peek           ()

class Stack<T> : ICollection<T>
    (C)      Stack<T>      ()
    (C)      Stack<T>      (IEnumerable<T>)
    void     Push          (T)
    T        Pop           ()
    T        Peek          ()

class List<T> : IList<T>
    (C)      List<T>      ()
    (C)      List<T>      (IEnumerable<T>)
    int      Capacity

class HashSet<T> : ISet<T>
    (C)      HashSet<T>      ()
    (C)      HashSet<T>      (IEnumerable<T>)
    IEqualityComparer<T> Comparer
    IEqualityComparer<HashSet<T>> CreateSetComparer ()

class SortedSet<T> : ISet<T>
    (C)      SortedSet<T>      ()
    (C)      SortedSet<T>      (IEnumerable<T>)
    (C)      SortedSet<T>      (IComparer<T>)
    (C)      SortedSet<T>      (IEnumerable<T>, IComparer<T>)
    IComparer<T> Comparer

class SortedList<K,V> : IDictionary<K,V>
    (C)      SortedSet<K,V>      ()
    (C)      SortedSet<K,V>      (IComparer<K>)

class SortedDictionary<K,V> : IDictionary<K,V>
    (C)      SortedDictionary<K,V>()
    (C)      SortedDictionary<K,V>(IComparer<K>)

```

Bijlage E

Syntax

