Mobiel programmeren

Jeroen Fokker
Departement Informatica
Universiteit Utrecht

16 oktober 2016

Korte inhoudsopgave

1 Mobiel programmeren *1*

2 Een app met een view *13*

3 En. . . aktie! *27*

4 Methoden om te tekenen *37*

5 Objecten en methoden *55*

6 Touch & go *69*

7 Herhaling en keuze *87*

8 Goede bedoelingen *103*

9 Klassen, Strings, en Arrays *117*

10 In lijsten en bewaren *135*

A Syntax *157*

B Werkcollege-opgaven *164*

C Practicum *171*

D Class library Mobiel Programmeren *174*

Inhoudsopgave

1 Mobiel programmeren *1*
1.1 Computers en programma's *1*
1.2 Orde in de chaos *2*
1.3 Programmeerparadigma's *4*
1.4 Programmeertalen *5*
1.5 Vertalen van programma's *7*
1.6 Programmeren *8*
1.7 Mobiel programmeren *10*

2 Een app met een view *13*
2.1 Soorten programma's *13*
2.2 Opbouw van een C#-programma *14*
2.3 Opbouw van een Android-programma *17*
2.4 Syntax-diagrammen *19*
2.5 Methodes *21*
2.6 Een layout met meerdere views *22*

3 En. . . aktie! *27*
3.1 Klikken op buttons *27*
3.2 Een kleurenmixer *31*

4 Methoden om te tekenen *37*
4.1 Een eigen subklasse van View *37*
4.2 Variabelen *41*
4.3 Berekeningen *42*
4.4 Programma-layout *46*
4.5 Declaraties met initialisatie *47*
4.6 Methode-definities *48*
4.7 Op zoek naar parameters *52*

5 Objecten en methoden *55*
5.1 Variabelen *55*
5.2 Objecten *57*
5.3 Object-variabelen *59*
5.4 Typering *64*
5.5 Constanten *66*
5.6 Static properties *68*

6 Touch & go *69*
6.1 Touch *69*
6.2 Lijsten *73*
6.3 Sensors *75*
6.4 Gestures *81*
6.5 Detectors *82*
6.6 Andere sensors *83*

7 Herhaling en keuze *87*
7.1 De while-opdracht *87*
7.2 bool waarden *89*
7.3 De for-opdracht *90*
7.4 Bijzondere herhalingen *92*
7.5 Toepassing: kleurenkubus *94*
7.6 Toepassing: Renteberekening *98*

8 Goede bedoelingen *103*
8.1 Een App met meerdere activiteiten *103*
8.2 Dialogen *103*
8.3 Lanceren van Activities *105*
8.4 Verkorte notaties *112*

9 Klassen, Strings, en Arrays *117*
9.1 Klassen *117*
9.2 Strings *122*
9.3 Arrays *125*
9.4 Een programma voor tekst-analyse *128*

10 In lijsten en bewaren *135*
10.1 ListView: een lijst op het scherm *135*
10.2 Een eigen Adapter *138*
10.3 Interactief toevoegen en verwijderen *146*
10.4 Data bewaren in een database *149*
10.5 Een eigen subklasse *152*

A Syntax *157*

B Werkcollege-opgaven *164*
B.1 Serie 1 *164*
B.2 Serie 2 *166*
B.3 Serie 3 *168*
B.4 Serie 4 *169*

C Practicum *171*
C.1 Practicum 1 *171*

D Class library Mobiel Programmeren *174*

Hoofdstuk 1

Mobiel 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 én 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, GPS, magnetische sensor, beeldscherm, 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

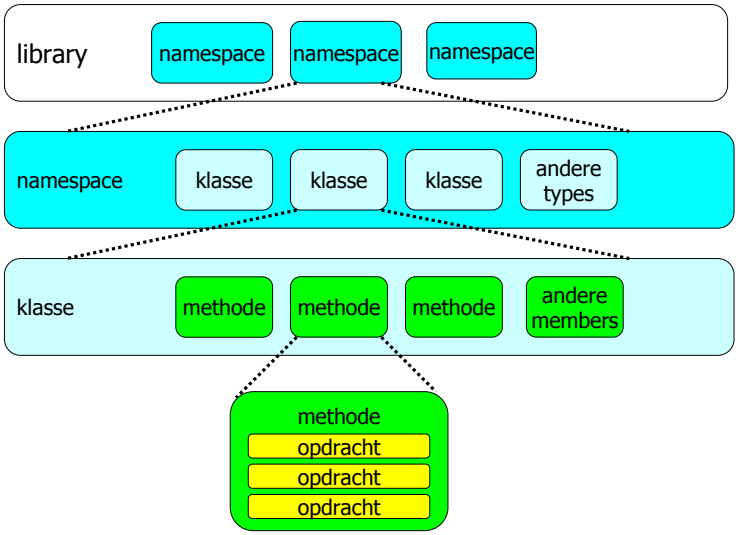
Omvang van het geheugen
Weinig zaken hebben zo'n spectaculaire groei doorgemaakt als de omvang van het geheugen van computers. In 1948 werd een voorstel van Alan Turing om een (één) computer te bouwen met een geheugencapaciteit van 6 kilobyte nog afgekeurd (te ambitieus, te duur!). Tegenwoordig zit dat 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 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 dat bij elkaar hoort 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. In 1995 pasten operating system, programmeertaal en tekstverwerker samen in een ROM van 256 kilobyte; in 2005 werd een tekstverwerker geleverd op meerdere DVD's à 4 gigabyte, en tegenwoordig haalt je telefoon nieuwe versies op van al je apps zodra hij wifi ruikt.

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 er ook met een team 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 ontstond 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 mogelijke *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).

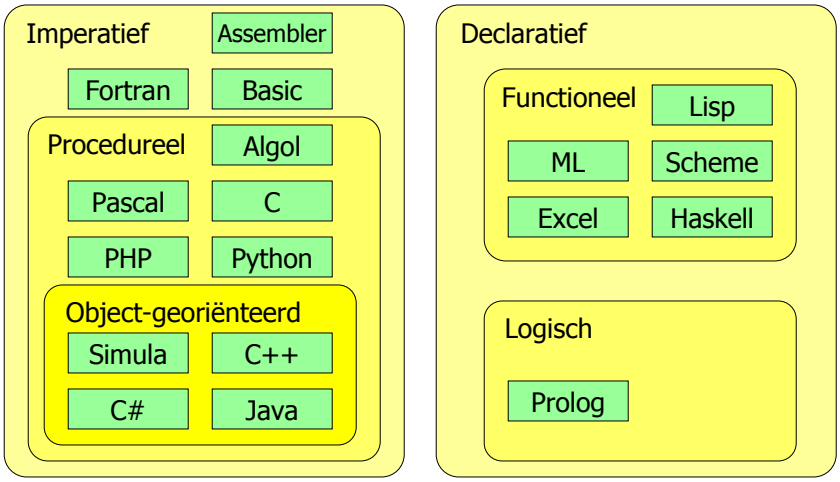
1.3 Programmeerparadigma's

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.

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 propositielogica zijn gebaseerd: de *logische programmeertalen*. Samen staan deze bekend als het *declaratieve paradigma*.

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
Object-georiënteerde talen vormen weer een uitbreiding van procedurele 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 (Backus-Naur-form, of 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 voor een loket, 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. Eén 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 konden 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 aan 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 draaiden. 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
Java SE	9	juli 2017	C#	6.0	juli 2015

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 ontstond 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++.

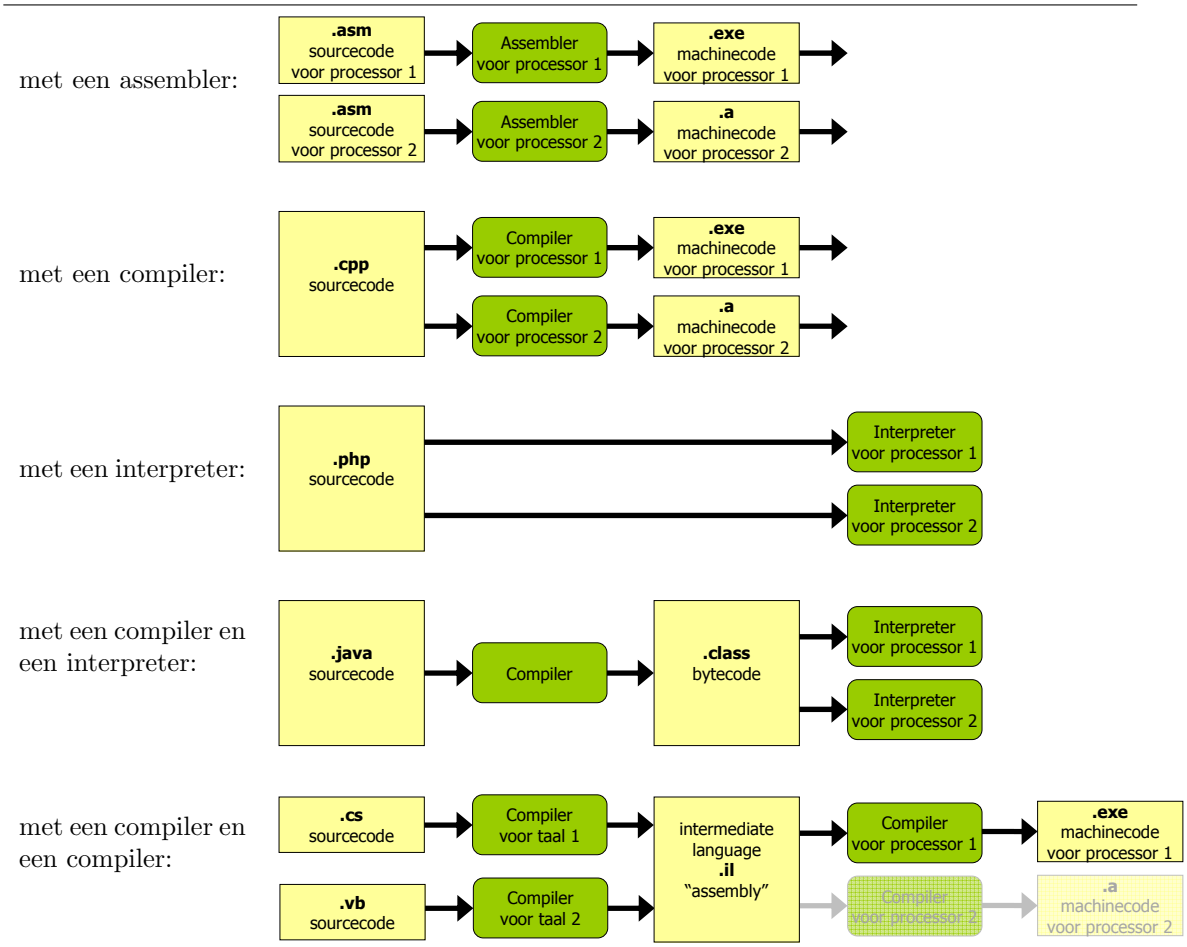
Interpreter
Een directere manier om programma's te vertalen is met behulp van een *interpreter*. Dat is een programma dat de broncode leest, en de opdrachten daarin direct uitvoert, dus zonder deze eerst te vertalen naar machinecode. De interpreter is specifiek voor de machine, maar de broncode is machine-onafhankelijk.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

1.7 Mobiel programmeren

Vaste telefonie
De telefoon is ouder dan de computer. De ontwikkeling en ingebruikname ervan is ontzettend snel gegaan: de snelheid van innovatie en het 'uitrollen van een landelijk dekkend netwerk' was in de negentiende eeuw echt niet anders dan nu. Kijk maar naar de jaartallen: deed telefoon-uitvinder Bell in 1876 nog de eerste experimenten, in 1878 was er al een commercieel netwerk (met 21 abonnees). In 1880 ontstonden de eerste netwerken in Nederlandse steden. In 1888 werden de stad-netwerken verbonden en kon er voor het eerst interlokaal worden gebeld. In 1895 werd er voor het eerst internationaal gebeld (met België).
Alle verbindingen waren overigens bovengronds. Pas in 1904 werd de eerste ondergrondse kabel gelegd (Amsterdam-Haarlem). In 1915 waren er in Nederland 75.000 abonnees.
Verbindingen verliepen via een operator, die op een plugbord een directe fysieke verbinding moest leggen. Maar vanaf 1925 deden de eerste automatische centrales de intrede, en kregen telefoons een kiesschijf. De schakeltechniek die hiervoor gebruikt werd was direct van invloed op de ontwikkeling van proto-computers. Zo werd de mechanische 'Bomba' decodeermachine, waarmee Turing in de Tweede Wereldoorlog werkte, helemaal gefabriceerd met telefoon-schakeltechnologie. In Nederland was het automatische netwerk in 1962 landelijk uitgerold en waren operators voortaan overbodig.

Mobiele telefonie
De eerste prototypes van mobiele verbindingen ontstonden in 1956. Zweden had de primeur in 1956 met het eerste mobiele netwerk. Toestellen wogen 40kg, en waren dus typisch bedoeld voor gebruik in een auto. Motorola werd een bekende fabrikant van auto-telefoons.
In 1973 lanceerde Motorola voor het eerst een 'handheld' (van ruim 1kg). De eerste standaard voor mobiele communicatie (het '1G-netwerk'), dat analoge spraakverbinding gebruikte, ontstond in 1979. De eerste digitale standaard (het '2G-netwerk' van het 'Global System for Mobile Communication (GSM)) is uit 1991. Vanaf die tijd breken de mobieltjes door, voor spraak en een 'Short Message Service' (SMS) via een kanaal dat eigenlijk bedoeld was voor intern beheer van het netwerk. Gelijktijdig ontstonden 'Personal Digital Assistents' (PDA), die agenda- en notitiefaciliteiten toevoegde.

Smartphones
Echte smartphones, waarbij internet beschikbaar is op de telefoon, kwamen er pas met het 3G-netwerk. De eerste prototypes zijn uit 2001, de doorbraak kwam in 2004. Vroege operating systems (Symbian) werden weggevaagd door de introductie van de Apple iPhone in 2007, en de HTC Dream (met Android) in 2008.
Android is een bedrijf uit 2003, dat zich oorspronkelijk richtte op het 'smart' maken van digitale camera's. In 2005 is het overgenomen door Google, en vanaf 2007 was Android beschikbaar als operating system voor mobiele telefonie. Microsoft dreigde de boot te missen, en kwam in 2010 met een derde standaard voor smartphones: Windows Phone.

Mobiel programmeren
En zo komen we bij de titel van deze cursus: mobiel programmeren. In feite is het natuurlijk niet het programmeren dat mobiel gebeurt (hoewel dat ook zou kunnen), maar het gebruik van het programma.
De drie concurrerende bedrijven (Apple, Google, en Microsoft) gebruikten natuurlijk niet dezelfde programmeertaal. Apple ontwikkelde een eigen programmeertaal: het heet 'Objective C' maar lijkt qua constructies meer op Smalltalk dan op C. Google baseerde zich met Android op het Linux operating system, en richt zich op programma's in Java. Microsoft gebruikt zijn eigen .NET-technologie, en gebruikt C#.
Natuurlijk ontstonden er ook allerlei mogelijkheden om een universele broncode te vertalen naar de drie systemen. Maar omdat die systemen zo uiteen lopen bieden die niet het gebruikscomfort dat de met 'native' programma's verwende gebruikers verwachten.
Op zich hoeft het programmeren van een smartphone zich niet te beperken tot de oorspronkelijke programmeertaal. Apple is zich inmiddels steeds meer aan het richten op een functionele programmeertaal: Swift.
Xamarin biedt een library waarmee je Android-programma's èn iOS en Windows Phone programma's in C# kunt schrijven. Dat lijkt wel een veelbelovende ontwikkeling. De universele library is nog iets te experimenteel, en daarom gebruken we in deze cursus de native Xamarin.Android library, waarin je architectuur van Android zoals die ook vanuit Java wordt gebruikt nog helemaal terugziet.

Hoofdstuk 2

Een app met een view

2.1 Soorten programma's

C# is opgezet als universele programmeertaal. De taalconstructies zijn hetzelfde, of het nu om een fotobewerkingsprogramma, een agenda-app, 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 apps, en omgekeerd. Voor verschillende toepassingen gebruik je verschillende raamwerken, elk met hun eigen idioom.
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 opzet van het programma. Enkele veelgebruikte vormen zijn:

- *Console-applicatie*: er is alleen een simpel tekstscherm 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 een andere externe partij) 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, gamecontroller, nunchuck enz., of zelfs met z'n vingers op een aanraakscherm. 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*-computer, 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 daarentegen doordat met GPS de locatie van het apparaat beschikbaar is, en er sensoren zijn voor de ruimtelijke oriëntatie. De naam is gepromoot door Apple voor programma's op de iPhone, maar werd al snel ook gebruikt voor Android programma's. Verwarrend genoeg gebruikt Microsoft de term tegenwoordig ook voor programma's op een computer, in een kennelijke poging om het onderscheid tussen een telefoon en een computer kleiner te maken.

In dit hoofdstuk bespreken we de simpelst denkbare Android app. Daarbij bespreken we het raamwerk voor de opbouw die specifiek voor apps geldt, maar het is ook een eerste kennismaking met de taalconstructies van C#. Met die taalconstructies kun je ook uit de voeten in toepassingen uit een andere categorie (bijvoorbeeld windows-applicaties of games), ook al komen die in deze cursus niet uitgebreid aan de orde.

2.2 Opbouw van een C#-programma

blz. 15

In listing 1 staat een van de kortst mogelijke apps in C#. Het is een app die de tekst `Hallo!` op het scherm van de telefoon toont, zoals afgebeeld in figuur 5. We bespreken aan de hand van dit programma eerst de opbouw van een C#-programma. Daarna bespreken we het nog eens, maar dan met de nadruk op de Android-specifieke keuzes die er zijn gemaakt.

Figuur 5: De app Hallo in werking

Opdrachten: bouwstenen van een imperatief programma

In een imperatief programma doen de *opdrachten* het eigenlijke werk: de opdrachten worden één voor één uitgevoerd. In dit programma staan er een handjevol, onder andere eentje om een `TextView`-object aan te maken:

```
scherm = new TextView(this);
```

eentje om daarop de tekst 'Hallo!' neer te zetten:

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

en eentje om deze `TextView` als gebruikersinterface van de app aan te wijzen:

```
this.SetContentView(scherm);
```

```
using Android.OS;       // vanwege Bundle
using Android.App;      // vanwege Activity
using Android.Widget;   // vanwege TextView
using Android.Graphics; // vanwege Color

[ActivityAttribute(Label = "Hello", MainLauncher = true)]
public class HalloApp : Activity
{
    protected override void OnCreate(Bundle b)
    {
        base.OnCreate(b);

        TextView scherm;
        scherm = new TextView(this);
        scherm.Text     = "Hallo!";
        scherm.TextSize = 80;
        scherm.SetBackgroundColor(Color.Yellow);
        scherm.SetTextColor      (Color.DarkBlue);

        this.SetContentView(scherm);
    }
}
```

Listing 1: Hallo/HalloApp.cs

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 zeven 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. Behalve opdrachten kunnen er in een blok ook *declaraties* staan, waarmee nieuwe namen van variabelen worden geïntroduceerd. In dit geval staat er één declaratie, waarmee de naam `scherm` wordt gekozen voor onze `TextView`:

```
TextView scherm;
```

Boven het blok staat de *header* van de methode, in dit geval:

```
protected override void OnCreate(Bundle b)
```

Hierin staat onder andere de naam van de methode, in dit geval `OnCreate`. De programmeur mag de naam van de methode in principe vrij kiezen, maar hier gebruiken we de naam `OnCreate`, omdat een methode met die naam een bijzondere rol vervult in een Android-programma.

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:

```
public class HalloApp : Activity
```

In de klasse-header staat in ieder geval 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 de naam `HalloApp` gekozen. De naam moet uit één aanelkaargeschreven geheel bestaan. Om de leesbaarheid te vergroten worden daarbij hoofdletters gebruikt om de losse woorden waaruit zo'n naam bestaat te benadrukken.
Voorafgaand aan de eigenlijke klasse-header kan nog extra informatie worden gegeven over hoe de

klasse verwerkt moet worden door de compiler en hulpprogramma's daarvan. Bij onze klasse staat er ook zo'n *attribuut*:

```
[ActivityAttribute(Label = "Hello", MainLauncher = true)]
```

Compilatie-eenheid: groepje klassen in een file
De programmatekst staat opgeslagen in een tekstbestand. In één bestand kunnen meerdere klassen staan: de klasse-headers en de accolades geven duidelijk aan waar de grenzen liggen. Een tekstbestand wordt in zijn geheel door de compiler gecompileerd, en vormt dus een zogeheten *compilatie-eenheid*. In het voorbeeld staat er maar één klasse in de compilatie-eenheid.
De klassen van een programma mogen verspreid worden over meerdere bestanden, dus over meerdere compilatie-eenheden, maar dat is in dit voorbeeld niet nodig.

Using: gebruik van libraries
De bovenste regels van onze compilatie-eenheid zijn geen onderdeel van de klasse:

```
using Android.OS;
using Android.App;
using Android.Widget;
using Android.Graphics;
```

Met deze regels geven we aan dat in het programma klassen gebruikt mogen worden die beschikbaar zijn in een viertal libraries. Eén van die klassen is bijvoorbeeld de klasse `TextView` die beschikbaar is in de library `Android.Widget`.

Opdrachten
In het voorbeeld worden twee soorten opdrachten gebruikt: toekenningsopdrachten, en methode-aanroepen.
Een *toekenningsopdracht* is te herkennen aan het symbool `=` in het midden. Hiermee geef je een variabele een nieuwe waarde. Een voorbeeld is:

```
scherm = new TextView(this);
```

De variabele `scherm` krijgt hiermee een waarde toegekend, namelijk een nieuw `TextView`-object.
Een ander soort opdracht is de *methode-aanroep*. Hiermee zet je een bepaalde methode aan het werk. Een methode is een groepje opdrachten, en door de aanroep van de methode zullen deze opdrachten woren uitgevoerd. Een voorbeeld is:

```
scherm.SetBackgroundColor(Color.Yellow);
```

Hiermee wordt de methode `SetBackgroundColor` aan het werk gezet. Tussen de haakjes achter de naam van de methode kan nog extra informatie worden meegegeven die voor de methode van belang is.

Object: groepje variabelen
Een object is een groepje variabelen dat bij elkaar hoort, en die je als één geheel kunt behandelen. In het voorbeeld is `scherm` zo'n object. De variabelen in het object kun je met aparte toekenningsopdrachten een waarde geven. Met deze twee opdrachten:

```
scherm.Text     = "Hallo!";
scherm.TextSize = 80;
```

krijgen de variabelen `Text` en `TextSize` een waarde. Deze variabelen zijn een onderdeel van het object `scherm`. In de linkerhelft van de toekenningsopdracht staan de naam van het object, gevolgd door een punt, gevolgd door de naam van de variabele binnen het object. In de rechterhelft van de toekenningsopdracht staat de nieuwe waarde die de variabele krijgt.

Methoden hebben een object onderhanden
Ook bij de aanroep van een methode kunnen we een object vermelden. In deze methode-aanroep:

```
scherm.SetBackgroundColor(Color.Yellow);
```

wordt het object `scherm` onder handen genomen door de methode `SetBackgroundColor`. Met 'onder handen nemen' bedoelen we dat de methode de variabelen van het object mag bekijken en/of veranderen. Deze methode verandert variabelen van het object `scherm` op een zodanige manier, dat het scherm wanneer het aan de gebruiker getoond wordt een gele achtergrondkleur heeft.

De naam van het object staat vooraan, daarna volgt een punt, en daarachter staat de naam van de methode die wordt aangeroepen. Tussen de haakjes staat overige informatie die van belang is, in dat geval de gekozen kleur van de achtergrond.

Klasse: type van een object
Objecten hebben een type. Het object `scherm` bijvoorbeeld heeft het type `TextView`. Dit is aangegeven bij de *declaratie* van de variabele `scherm`:

```
TextView scherm;
```

Deze variabele maakt dat de variabele `scherm` een object van het type `TextView` kan aanduiden. Het type van een object, zoals hier `TextView`, is een *klasse*. De auteur van de klasse bepaalt uit welke variabelen objecten bestaan, en door welke methoden ze onder handen genomen kunnen worden. Zo heeft de auteur van `TextView` bedacht dat een object met het type `TextView` variabelen `Text` en `TextSize` heeft, en onder handen genomen kan worden door de methode `SetBackgroundColor`.
Omdat onze variabele `scherm` is gedeclareerd als `TextView`, is een toekenning als

```
scherm.TextSize = 80;
```

en een methode-aanroep als

```
scherm.SetBackgroundColor(Color.Yellow);
```

inderdaad mogelijk.

2.3 Opbouw van een Android-programma

We bespreken het programma nu nog eens, ditmaal om te zien hoe de opdrachten, methoden en klassen gebruik maken van de libraries die bedoeld zijn om Android-programma's te schrijven.

Activity: wat een app doet
De levensloop van een app is gemodelleerd in de library `Android.App`. Hierin zit een klasse `Activity` die een bijzondere rol speelt in het Android operating system.
Op het moment dat een gebruiker een app opstart, maakt het operating system een object aan van de klasse `Activity`. In dit object zitten alle variabelen die van belang zijn voor het beheer van de app. Android zorgt ervoor dat dit object vervolgens onder handen wordt genomen door de methode `OnCreate`. De opdrachten in deze methode bepalen dus wat de app doet. De naam `Activity` is heel toepasselijk gekozen: het bepaalt de activiteit die de app onderneemt. Een bepaalde app kan meerdere activiteiten ondernemen, maar in eenvoudige apps is er maar één activiteit.

Een eigen subklasse van Activity
Als programmeur van een app wil je natuurlijk zelf bepalen wat je app precies doet: dit is niet iets wat al in de library is vastgelegd. Aan de andere kant wil je niet *alles* wat de app moet doen zelf uitprogrammeren. Bijvoorbeeld, dat de app verdwijnt als je op de Back-knop van je telefoon drukt is iets wat in alle apps hetzelfde is.
We maken daarom in ons programma een klasse die een *uitbreiding* is van de in de library al bestaande klasse `Activity`. Alle standaard-gedrag van een app krijg je daarmee kado, en in het programma hoeft alleen maar het voor deze app specifieke gedrag te worden geprogrammeerd.
In C# is er een notatie om aan te geven dat een klasse een uitbreiding is van een andere klasse. In de header van onze enige klasse gebruiken we deze notatie:

```
public class HalloApp : Activity
```

Onze klasse heet `HalloApp`, en is een uitbreiding van de library-klasse `Activity`. Zo'n uitbreiding wordt een *subklasse* genoemd. Omgekeerd is `Activity` de *superklasse* of *base class* van `HalloApp`.

Attributen
In C# is er een notatie om bij een klasse aan te geven hoe de hulpprogramma's die het programma verwerken met de klasse moeten omgaan. Deze zogeheten *attributen* staan tussen vierkante haken boven de klasse-header.
Omdat een app in principe uit meerdere activities kan bestaan, is het in Android verplicht om bij één activity aangeven dat het deze activiteit is die moet worden ondernomen als de gebruiker de app start. Ook al heeft ons programma maar één activiteit, toch moet daarbij worden aangegeven dat het de `MainLauncher` is.

We schrijven daarom boven de header van onze klasse:

```
[ActivityAttribute(Label = "Hello", MainLauncher = true)]
```

Hiermee geven we aan dat het deze subklasse van `Activity` is, waarvan het operating system een object zal aanmaken op het moment dat de app gelanceerd wordt. En we maken meteen gebruik van de gelegenheid om de naam in de titelregel van de app te kiezen.

Herdefinitie van OnCreate

Bij de start van een app roept het operating system de methode `OnCreate` aan. Die methode bestaat in de klasse `Activity`, en doet wat er in elke app gebeuren moet tijdens het creëren ervan. Als je als programmeur wilt dat er in jouw zelfgemaakte app nog meer gebeurt, dan kun je de methode `OnCreate` opnieuw definiëren in een subklasse van `Activity`. Dit is wat we doen in de klasse `HalloApp`: we geven een definitie van de methode `OnCreate`. In de header staat het woord `override` om aan te geven dat deze methode in de plaats komt van de oorspronkelijke methode `OnCreate` in de klasse `Activity`. Op het moment dat het operating system de app creëert (of preciezer: de als `MainLauncher` aangemerkte activiteit) wordt dus onze eigen versie van `OnCreate` aangeroepen. Daarmee hebben we de macht in handen gekregen om te bepalen wat de app gaat doen.

Doordat we zo eigenwijs zijn om deze methode een nieuwe invulling te geven, gebeurt er nu niet meer wat in elke app in ieder geval ook moet gebeuren. Dat is nou ook wel weer jammer, want daardoor zou de app geen titelregel krijgen en zelfs helemaal niet meer verschijnen. Als eerste opdracht in de her-gedefinieerde versie van `OnCreate` schrijven we daarom:

```
base.OnCreate(b);
```

Dit zorgt ervoor dat alsnog de oorspronkelijke versie van `OnCreate`, zoals die in de klasse `Activity` is gedefinieerd, ook wordt aangeroepen. Daarna staan we echt in de startblokken om ook nog iets extra's te doen.

View: wat een app laat zien

Een app communiceert met de gebruiker door middel van een *view*. Vrijwel elke app maakt zo'n view aan, anders valt er voor de gebruiker niets te zien. In bijzondere gevallen zijn er apps zonder view denkbaar, bijvoorbeeld een app die er op de achtergrond voor zorgt dat er muziek wordt afgespeeld. Maar meestal is er wel een view, en het is de taak van de methode `OnCreate` om er een aan te maken.

Er zijn verschillende soorten views: je kunt met de gebruiker communiceren met teksten, maar ook met plaatjes, drukknoppen, schuifregelaars, invulvelden, enzovoorts. Voor elk soort view is er in de library een subklasse van `View` beschikbaar. In ons programma kiezen we voor de klasse `TextView`. Via een object van deze klasse kun je een tekst aan de gebruiker tonen.

In de body van de methode `OnCreate` declareren we daarom een variabele van het type `TextView`:

```
TextView scherm;
```

en we zorgen er voor dat er ook echt zo'n object wordt aangemaakt:

```
scherm = new TextView(this);
```

Zoals elk object bevat ook een `TextView`-object variabelen, die we met een toekenningsopdracht kunnen veranderen:

```
scherm.Text     = "Hallo!";
scherm.TextSize = 80;
```

Sommige variabelen, zoals de variabele waarin de achtergrondkleur van de view wordt bewaard, mogen we niet direct veranderen met een toekenningsopdracht. Wel kunnen we het object onder handen nemen met een methode-aanroep met het gewenste effect:

```
scherm.SetBackgroundColor(Color.Yellow);
scherm.SetTextColor      (Color.DarkBlue);
```

Helemaal logisch is dit niet: je zou toch verwachten dat je ook de kleuren via een toekenningsopdracht zou kunnen veranderen, of omgekeerd misschien dat je de tekst van een `TextView` kunt aanpassen met een aanroep van een methode `SetText`. Zo heeft de auteur van `TextView` het echter niet gewild. Met zo'n moment van onoplettendheid van de programmeur van de library-klasse zullen we moeten leven: het is zoals het is...

De laatste opdracht in de methode `OnCreate` zorgt ervoor dat het nu geheel naar smaak geconfigureerde `TextView`-object daadwekelijk gebruikt gaat worden als userinterface van onze app:

```
this.SetContentView(scherm);
```

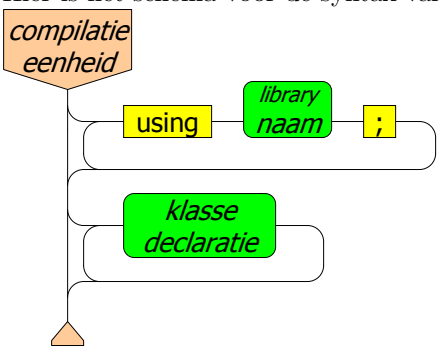
2.4 Syntax-diagrammen

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 deel-constructie. 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 kiezen van een route 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 A.

Syntax van compilatie-eenheid

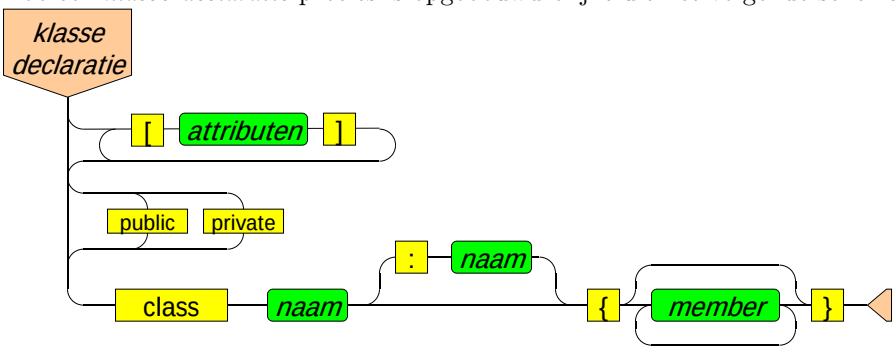
Hier is het schema voor de syntax van een *compilatie-eenheid*:



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

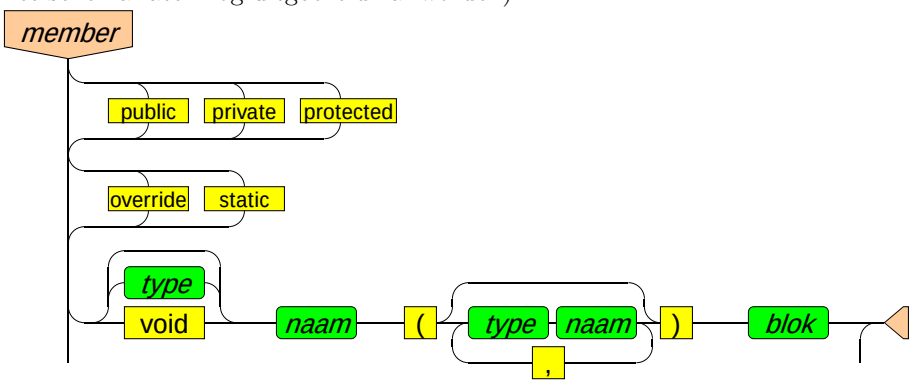
Hoe een *klasse-declaratie* precies is opgebouwd blijkt uit het volgende schema:



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.
Het schema biedt de mogelijkheid om achter de naam van de klasse een dubbele punt en de naam van een reeds bestaande klasse te schrijven. Deze mogelijkheid hebben we in het voorbeeldprogramma benut om onze klasse `HalloApp` een subklasse te laten worden van de bestaande library-klasse `Activity`.

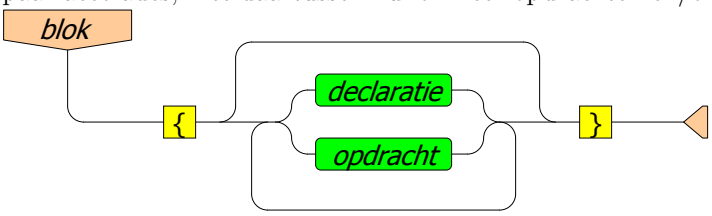
Syntax van member
Er zijn verschillende soorten *members* mogelijk in een klasse, maar de belangrijkste is de methode-definitie. De syntax daarvan is voorlopig als volgt (de doodlopende einden onderaan geven aan dat het schema later nog uitgebreid zal worden):



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

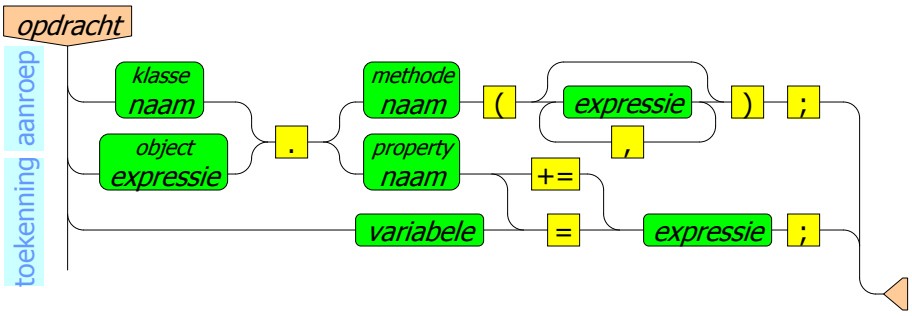
```
protected override void OnCreate(Bundle b)
```

gevolgd door het blok met de methode-body een geldige *member* vormt. In plaats van `protected` kan er blijkbaar ook wel eens `public` of `private` staan, of helemaal niets. In sommige methoden zal er in plaats van `override` wel eens `static` staan, of ook hier weer helemaal niets. In plaats van `void` staat er ook wel eens een *type* (wat dat ook moge zijn), of alweer helemaal niets, de haakjes zijn weer wel verplicht en daar staat soms ook weer iets tussen, waar `Bundle b` een voorbeeld van is.
Bij elke methode maakt de programmeur zo zijn keuzes. We zagen al dat `override` betekent dat een methode uit de superklasse opnieuw gedefinieerd wordt. Wat de betekenis van woorden als `protected` en `void` is bespreken we later.
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 in 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:



Verschillende routes door dit schema hebben we nodig gehad voor het construeren van de opdrachten in het voorbeeldprogramma.

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.

Semantiek van een toekenningsopdracht
Als een toekenningsopdracht door de processor wordt uitgevoerd, dan wordt eerst de waarde van expressie aan de rechterkant van het =-teken bepaald. De variabele aan de linkerkant van het =-teken krijgt daarna die waarde.
Je kunt dit teken het beste niet als 'is' uitspreken, maar als 'wordt': de waarde variabele *is* namelijk nog niet gelijk aan de waarde van de expressie, maar hij *wordt* het door het uitvoeren van de opdracht.

2.5 Methodes

Methodes definiëren en aanroepen
We krijgen van twee kanten te maken met een methode:

- In het programma staan *definities* van methoden. In de body van de methode wordt vastgelegd hoe de methode werkt. Het voorbeeldprogramma bevat één definitie: die van de methode `OnCreate`.
- Door middel van een opdracht kun je een methode *aanroepen*. Als gevolg van zo'n aanroep worden de opdrachten in de body van de methode uitgevoerd. Het voorbeeldprogramma bevat aanroepen van de methoden `SetBackgroundColor`, `SetTextColor` en `SetContentView`.

Methodes die je aanroept moeten ergens zijn gedefinieerd. Soms gebeurt dat in je eigen programma, maar in dit geval zijn ze afkomstig uit libraries: de eerste twee staan in de library `Android.Widget`, en de derde in de library `Android.App`.
Methodes die je in een programma definieert zijn natuurlijk bedoeld om aan te roepen. Soms gebeurt dat in je eigen programma, maar in dit geval gebeurt dat vanuit het operating system: de methode `OnCreate` is immers de methode die door het operating system wordt aangeroepen op het moment dat de `Activity` die als `MainLauncher` is benoemd wordt gestart.

Parameters van een methode
Bij de aanroep van een methode kun je extra informatie vermelden die bij het uitvoeren van de methode van belang is. In het geval van de aanroep

```
scherm.SetBackgroundColor(Color.Yellow);
```

is dat de kleur die we als achtergrond willen gebruiken. Dit heet het *meegeven van een parameter* aan een methode. In de header van de methode staat een *declaratie van de parameter* die aan de

methode kan worden meegegeven. Zo zal in de header van de methode `SetBackgroundColor` een declaratie als `Color c` staan. Dat zie je in het programma echter niet, want deze methode is in de library gedefinieerd.
Wel zie je in het voorbeeldprogramma de header van de methode `OnCreate`:

```
protected override void OnCreate(Bundle b)
```

In deze header wordt een parameter van het type `Bundle` gedeclareerd. Blijkbaar moet er bij aanroep van `OnCreate` een `Bundle`-object worden meegegeven. Die aanroep zie je in het programma echter niet, want deze methode wordt vanuit het operating system aangeroepen.
Het is trouwens nu nog niet duidelijk waar die `Bundle`-parameter eigenlijk goed voor is. Toch moet hij worden gedeclareerd, want het operating system geeft altijd een `Bundle`-object mee bij de aanroep van `OnCreate`.

this: het object dat de methode onder handen neemt
Elke methode heeft een object onder handen, of anders gezegd: het bewerkt een object. Dat object staat voor de punt in de methode-aanroep. In het voorbeeldprogramma nemen de methodes `SetBackgroundColor` en `SetTextColor` het object `scherm` onder handen. Dat object is een `View`, of meer precies: een `TextView` (zo is `scherm` immers gedeclareerd), en het is dat object dat van een achtergrond- en tekstkleur wordt voorzien.
Ook de methode `OnCreate` heeft een object onder handen gekregen toen hij werd aangeroepen door het operating system. Dat object is een `Activity`, of meer precies: een `HalloApp` (in die klasse is `OnCreate` immers gedefinieerd).
Bij de aanroep van `SetContentView` wordt datzelfde object ook weer verder onder handen genomen door `SetContentView`. Het is immers de `Activity` die zojuist is gecreëerd die een view moet krijgen. Binnen een methode kun je het object-onder-handen aanduiden met `this`. In de body van `OnCreate` is `this` dus het `HalloApp`-object dat door `OnCreate` wordt bewerkt. Het is ditzelfde object dat ook door `SetContentView` onder handen genomen moet worden. Daarom staat `this` voor de punt bij aanroep van `SetContentView`:

```
this.SetContentView(scherm);
```

De view die de gebruikersinterface van onze app gaat vormen, in dit geval `scherm`, geven we mee als parameter.

base: het object zonder hergedefinieerde methoden
Een bijzondere aanroep is nog de eerste opdracht in de body van `OnCreate`:

```
base.OnCreate(b);
```

We moeten hierbij bedenken dat de definitie van `OnCreate` in de klasse `HalloApp` een *herdefinitie* is van de methode `OnCreate` zoals die ook al in de superklasse `Activity` bestond. We hebben het operating system verleid om onze her-gedefinieerde methode aan te roepen, maar wat er in de oorspronkelijke methode gebeurde blijft ook van belang. Daarom roepen we deze oorspronkelijke methode aan.
Het keyword `base` duidt hetzelfde object aan als `this`: het object dat de huidige methode onder handen heeft. Het verschil met `this` is echter dat `base` het type van de superklasse heeft. Dus in het voorbeeld is `base` een `Activity`-object, terwijl `this` een `HalloApp`-object is. Daardoor wordt door de aanroep met `base` de oorspronkelijke versie aangeroepen. De `Bundle` met de naam `b` wordt hierbij ongewijzigd meegegeven als parameter.
Zouden we de aanroep doen door `this.OnCreate(b);` dan wordt niet de oorspronkleijke versie van de methode aangeroepen, maar de methode zelf. Dat is ongewenst, want het eerste dat die methode dan weer doet is zichzelf aanroepen, en we raken verstrikt in een oneindige keten van een zichzelf aanroepende methode. Filosofisch is dat wel interessant, maar de app raakt er door bevroren en lijkt niets meer te doen.

2.6 Een layout met meerdere views

Een app die alleen maar een simpele tekst in beeld brengt wordt al gauw saai. Gelukkig is het mogelijk om meerdere views tegelijk in beeld te brengen. Als voorbeeld ontwikkelen we in deze sectie een app met twee views: een analoge klok èn een digitale klok.
blz. 24 Het programma staat in listing 2 en figuur 6 toont de app in werking.

Figuur 6: De app Klok in werking

View maakt iets zichtbaar in een app

Een `View` is een object dat iets zichbaar kan maken: een tekst, een plaatje, een kaart, een bedienings-element, enzovoorts. Bijna elke app maakt in `OnCreate` een `View`-object aan, want als er niets te zien is heb je weinig aan een app. Een app zonder view is wel toegestaan, want in zeldzame gevallen hoeft een app niet zichbaar te zijn: bijvoorbeeld een app die verantwoordelijk blijft voor het afspelen van muziek, of die op de achtergrond de GPS-locatie logt, of telefoon-oproepen automatisch beantwoordt.

Het vorige voorbeeld gebruikte een `TextView`. In de library `Android.Widgets` zijn nog veel meer subklassen van `View` beschikbaar:

- `TextView` om een tekst te tonen
- `EditView` om een tekst te tonen die de gebruiker ook kan veranderen
- `ImageView` om een plaatje te tonen
- `Button` om een knop te tonen die de gebruiker kan indrukken
- `SeekBar` om een schuifregelaar te tonen die de gebruiker kan bedienen
- `AnalogClock` om een complete wijzerklok te tonen, die ook automatisch loopt
- `TextClock` om een digitale klok te tonen, die ook automatisch loopt

In het nieuwe voorbeeld gebruiken we een `AnalogClock` en een `TextClock`.

LinearLayout groepeert views

Het aanmaken van views gebeurt in feite altijd op dezelfde manier. Je declarareert een variabele van de gewenste klasse, en je geeft de variabele als waarde een nieuw gemaakt object. Bij een `TextView` ging dat zo:

```
TextView scherm;
scherm = new TextView(this);
```

Voor de twee soorten klok in dit programma gebeurt dat met:

```
AnalogClock wijzerklok;
wijzerklok = new AnalogClock(this);
TextClock tekstklok;
tekstklok = new TextClock(this);
```

Waarschijnlijk kun je nu wel raden hoe je te werk moet gaan als je ooit eens een `ImageView` of een `Button` nodig hebt.

```
using Android.OS;       // vanwege Bundle
using Android.App;      // vanwege Activity
using Android.Widget;   // vanwege AnalogClock, TextClock, LinearLayout
using Android.Graphics; // vanwege Color

[ActivityAttribute(Label = "Klok", MainLauncher = true)]
public class KlokApp : Activity
{
    protected override void OnCreate(Bundle b)
    {
        base.OnCreate(b);

        AnalogClock wijzerklok;
        wijzerklok = new AnalogClock(this);
        wijzerklok.SetBackgroundColor(Color.Yellow);

        TextClock tekstklok;
        tekstklok = new TextClock(this);
        tekstklok.Format24Hour = "EEE HH:mm:ss";
        tekstklok.TextSize = 50;

        LinearLayout stapel;
        stapel = new LinearLayout(this);
        stapel.Orientation = Orientation.Vertical;

        stapel.AddView(wijzerklok);
        stapel.AddView(tekstklok);

        this.SetContentView(stapel);
    }
}
```

Listing 2: Klok/KlokApp.cs

Is de view eenmaal gecreëerd, dan kun je er nog wat eigenschappen van veranderen door middel van methode-aanroepen en/of toekenningsopdrachten:

```
wijzerklok.SetBackgroundColor(Color.Yellow);
tekstklok.Format24Hour = "EEE HH:mm:ss";
tekstklok.TextSize = 50;
```

We hebben nu twee verschillende views, maar het probleem is dat we bij de aanroep van `SetContentView` maar één view als parameter kunnen meegeven. De oplossing is dat we nog een derde view aanmaken, ditmaal van het type `LinearLayout`:

```
LinearLayout stapel;
stapel = new LinearLayout(this);
```

Zo'n layout-view kan andere views groeperen. Dat gebeurt door de te groeperen views, in dit geval de twee klokken, mee te geven aan herhaalde aanroepen van `AddView`:

```
stapel.AddView(wijzerklok);
stapel.AddView(tekstklok);
```

Tenslotte kan de `LinearLayout` als geheel gebruikt worden als gebruikersinterface van onze app:

```
this.SetContentView(stapel);
```

Configuratie van views

Elke view heeft eigenschappen die je kunt manipuleren met toekenningsopdrachten en/of methode-aanroepen. Een view is een object, en een object is een groepje variabelen: in die variabelen worden de eigenschappen bewaard.

De auteur van de klasse bepaalt welke eigenschappen er zijn, en welke methodes er aangeroepen kunnen worden. Je moet dat opzoeken in de documentatie van de klasse, en de ontwikkelomgeving wil er ook nog wel eens bij helpen. Sommige eigenschappen zijn geldig bij verschillende typen views. We hebben bijvoorbeeld de `TextSize` ingesteld van zowel een `TextView` als een `TextClock`:

```
scherm.TextSize = 80;
tekstklok.TextSize = 50;
```

En we hebben de achtergrondkleur veranderd van zowel een `TextView` als een `AnalogClock`:

```
scherm.SetBackgroundColor(Color.Yellow);
wijzerklok.SetBackgroundColor(Color.Yellow);
```

Andere eigenschappen zijn heel specifiek voor een bepaald type view. Zo heeft alleen een `TextClock` een `Format24Hour` eigenschap, en heeft alleen een `LinearLayout` een `Orientation`:

```
tekstklok.Format24Hour = "EEE HH:mm:ss";
stapel.Orientation = Orientation.Vertical;
```

Sommige eigenschappen zijn zo wel duidelijk (zoals `TextSize` en de achtergrondkleur). Andere eigenschappen vereisen wat toelichting, zoals de codering van de opbouw van het 'format' van de tekstklok: `mm` staat voor minuten, `ss` voor seconden, `HH` voor uren – dat is nog wel te begrijpen. Dat `EEE` de dag van de week laat zien is al minder logisch, en dat er verschil is tussen `HH` (voor 24-uurs uren) en `hh` (voor 12-uurs uren) moet je ook maar net weten. Je kunt gemakkelijk in dit soort feitenkennis verdrinken, en je moet het vooral niet allemaal proberen te onthouden. Wel is het handig om een globaal beeld te hebben van wat zoal de mogelijkheden zijn van de verschillende typen views.

Klasse-hierarchie van views

Alle views zijn subklasse van de klasse `View`. Je kunt dat in het programma niet zien: dit is vastgelegd in de library waarin deze klassen zijn gedefinieerd. Hierin staat bijvoorbeeld dat de klasse `TextView` een subklasse is van `View`:

```
class TextView : View { ... }
```

En ook `AnalogClock` is een subklasse van `View`:

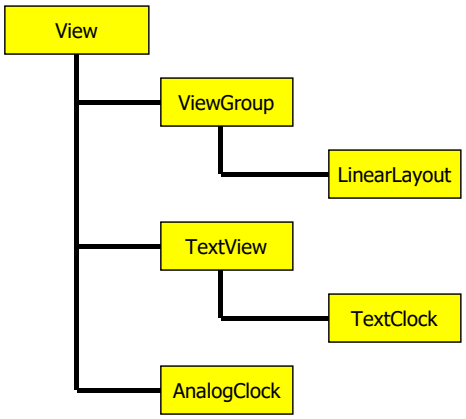
```
class AnalogClock : View { ... }
```

Dat van beide de achtergrondkleur kan worden ingesteld met `SetBackgroundColor` is geen toeval, want deze methode is gedefinieerd in de klasse `View`. Daardoor kunnen beide typen view (en alle andere subklassen van `View` dus ook) gebruik maken van deze methoden.

De klasse `TextClock` is op zijn beurt een subklasse van `TextView`.

```
class TextClock : TextView { ... }
```

Een eigenschap als `TextSize` is gedefinieerd in de klasse `TextView`. Daarom kan deze voor een `TextView` gebruikt worden, maar ook voor `TextClock`: dat is immers een subklasse daarvan.
In een schema is de hiërarische samenhang tussen de klassen duidelijker te tonen dan met een reeks klasse-headers. Dit is hoe de subklassen van `View` op elkaar voortbouwen:



Hieruit blijkt dat de klasse `LinearLayout` niet een directe subklasse is van `View`, maar dat er nog een klasse `ViewGroup` tussen zit. Dit roept de vraag op welke subklassen van `ViewGroup` er dan nog meer zijn, en in welke zin die verschillen van `LinearLayout`. Dat is nog wel eens leuk om uit te zoeken in de documentatie van de library.

Obsoleted klassen
Sommige klassen zijn zeer universeel bruikbaar, zoals `TextView` en `Button`. Met eigenschappen kun je ze naar behoefte configureren, en dat maakt ze bruikbaar in veel programma's. Andere klassen dienen een erg specifiek doel: een `AnalogClock` heb je maar zelden nodig in een programma.
Eigenlijk is het een beetje onzinnig om zulke specifieke klassen in de standaardlibrary op te nemen: dit is meer iets voor een extra library die je bij gelegenheid nog eens apart zou kunnen downloaden. De auteurs van de `Android.Widget` library hebben dat inmiddels ook bedacht, en lijken er spijt van te hebben dat ze `AnalogClock` in de library te hebben gezet. De klasse is namelijk sinds enige tijd[1] in de library gemarkeerd als *obsoleted*. Dat betekent dat wie de klasse in een programma gebruikt door de ontwikkelomgeving gewaarschuwd wordt dat deze klasse in de toekomst nog wel eens uit de library verwijderd zal worden.
Het woord 'obsoleted' is C#-jargon. In de Java-wereld spreekt men van 'deprecated' klassen en methodes. Met je taalgebruik verraad je je subcultuur.
Het is eigenlijk niet verstandig om obsoleted klassen in je programma's te gebruiken, want je loopt er het risico mee dat je programma in de toekomst niet meer opnieuw gecompileerd kan worden. Maar ach, die `AnalogClock`: hij is zo mooi... Geniet er nog maar van zolang het kan!

[1] Ik zou hier wel willen vermelden sinds welke versie dat het geval is, maar daar is moeilijk achter te komen. Als je met Google zoekt op 'AnalogClock obsoleted' dan krijg je alleen maar artikelen te zien die uitleggen dat een wijzerklok als zodanig niet meer van deze tijd is...

Hoofdstuk 3

En. . . aktie!

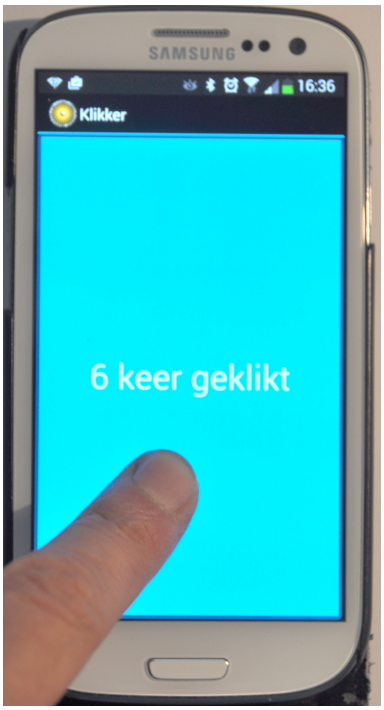
De apps in het vorige hoofdstuk waren alleen om naar te kijken. Maar een touchscreen heet niet voor niets een *touch*screen: je wilt ook dat er iets gebeurt als je het aanraakt! In dit hoofdstuk ontwikkelen we daarom twee apps waar de gebruiker met zijn vingers aan kan zitten.

3.1 Klikken op buttons

Klikker: een app met een teller

In deze sectie bespreken we de app 'Klikker'. De userinterface van deze app bestaat uit één grote button, die de gebruiker kan indrukken. Op de button verschijnt dan een tekst die aangeeft hoe vaak de button is ingedrukt. Je zou met deze app in de hand bij de ingang van een evenement kunnen gaan staan om het aantal bezoekers te tellen.

Het programma staat in listing 3 en figuur 7 toont de app in werking. blz. 28



Figuur 7: De KlikkerApp in werking

Button: een view om op te klikken

Net als de HalloApp in het vorige hoofdstuk heeft KlikkerApp een methode `OnCreate` waarin de userinterface wordt opgebouwd. Deze methode wordt door het operating system aangeroepen op het moment dat de app wordt gelanceerd.

```
using System;           // vanwege EventArgs
using Android.App;      // vanwege Activity
using Android.Widget;   // vanwege Button
using Android.OS;       // vanwege Bundle

[ActivityAttribute(Label = "Klikker", MainLauncher = true)]
public class KlikkerApp : Activity
{
    int teller;
    Button knop;

    protected override void OnCreate(Bundle b)
    {
        base.OnCreate(b);
        this.teller = 0;
        this.knop = new Button(this);
        this.knop.Text     = "Klik hier!";
        this.knop.TextSize = 40;
        this.knop.Click   += this.klik;
        this.SetContentView(knop);
    }

    public void klik(object o, EventArgs ea)
    {
        this.teller = this.teller + 1;
        this.knop.Text = this.teller.ToString() + " keer geklikt";
    }
}
```

Listing 3: Klikker/KlikkerApp.cs

In plaats van een `TextView` gebruikt deze app echter een `Button`, die we de toepasselijke naam `knop` geven. Ook `Button` is een subklasse van `View`, en daarom kan onze button `knop` gebruikt worden als parameter van `SetContentView`.

Events en EventHandlers
Een button kan door de gebruiker worden ingedrukt. Niet echt natuurlijk, het is in feite tikken op het scherm, maar de gebruiker ervaart dat als drukken op een echte knop. De button krijgt een andere kleur zodra de gebruiker hem aanraakt, en krijgt weer zijn oorspronkelijke kleur bij het loslaten. Samen geeft dat de 'look&feel' van een drukknop. Dit alles is in de library geprogrammeerd, we hoeven daar in ons programma niets voor te doen.
Maar we willen natuurlijk ook dat er iets zinvols gebeurt bij het indrukken van de knop. Dat moeten we wel in het programma opschrijven.
Het indrukken van een knop is een voorbeeld van een *event*: een gebeurtenis, vaak veroorzaakt door de gebruiker. Andere voorbeelden van events zijn het selecteren van een item in een lijstje, het verschuiven van een schuifregelaar, of het intikken van tekst. Maar niet altijd is de gebruiker de veroorzaker van een event: het afgaan van een wekker (althans, een geprogrammeerde versie daarvan) is ook een event.
In een programma kunnen we reageren op een event door een *event-handler* te schrijven. Een event-handler is een methode die automatisch wordt aangeroepen op het moment dat er een event optreedt. In het Klikker-programma is de methode `klik` een event-handler die moet reageren op het indrukken van de knop. Je ziet dat je zelf een naam mag bedenken voor zo'n event-handler.
Om ervoor te zorgen dat de event-handler inderdaad wordt aangeroepen als het event optreedt, moet je de event-handler *registreren*. Dat moet eenmalig gebeuren. De registratie vindt daarom typisch plaats in de methode `OnCreate`. De registratie gebeurt met de opdracht

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

en ziet er daarmee vrijwel hetzelfde uit als de opdrachten waarmee de andere eigenschappen van de knop worden vastgelegd:

```
knop.Text     = "Klik hier!";
knop.TextSize = 40;
```

We moeten `+=` in plaats van `=` schrijven omdat er in principe meerdere event-handlers geregistreerd kunnen worden voor hetzelfde event (al is dat ongebruikelijk). Rechts van het `+=` teken staat de naam van de methode die we als event-handler willen gebruiken. Dit is niet een aanroep van de methode: er staan geen haakjes achter de naam. De aanroep wordt (niet in ons programma, maar door het operating system) pas gedaan als er inderdaad een event optreedt.
Welke events er bij een view kunnen optreden is vastgelegd in de klasse van de view. In de klasse `Button` is bepaald dat er een `Click`-event bestaat. Datzelfde geldt ook voor de overige eigenschappen van een view, zoals `Text` en `TextSize`.

Definitie van de event-handler
De event-handler `klik` moet natuurlijk wel bestaan. Daarom definiëren we een methode `klik` in ons programma. Een event-handler methode moet altijd twee parameters hebben: een van het type `object`, en een van het type `EventArgs`. Via het `object` kun je bepalen welk object de veroorzaker is van het event, maar in dit programma is dat ook zo wel duidelijk, omdat er maar één button is. Via de `EventArgs` kan nog nadere informatie over het event worden opgevraagd, maar bij een button-klik is dat nauwelijks nodig: het belangrijkste is *dat* de knop is ingedrukt.
In de body van de event-handler komen de opdrachten te staan die moeten worden uitgevoerd als reactie op het event. Om te beginnen is dat het bijhouden van de telling. We gebruiken daartoe een variabele waarin een getal kan worden opgeslagen: `teller`. Op het moment dat de knop wordt ingedrukt moet de waarde van deze variabele één groter worden dan hij was. We schrijven daarom een toekenningsopdracht:

```
this.teller = this.teller + 1;
```

Je kunt het `=` teken hierbij maar beter niet als 'is' uitspreken, maar liever als 'wordt'. De waarde *is* immers niet zichzelf plus 1 (wiskundige gesproken is dat onzin), maar hij *wordt* zijn oude waarde plus 1.
Daarna willen we de nieuwe waarde van de variabele ook zichtbaar maken als opschrift van de knop. We doen daarom opnieuw een toekenning aan de `Text`-eigenschap van de knop:

```
knop.Text = teller.ToString() + " keer geklikt";
```

Member-variabelen: gedeclareerd in de klasse
De variabele `knop` is nodig in beide methoden: in `OnCreate` om hem klaar te zetten, en in `klik` om het opschrift te veranderen. Ook de variabele `teller` is nodig in beide methoden: in `OnCreate` om hem zijn startwaarde 0 te geven, en in `klik` om hem één groter te maken.
We declareren de variabelen daarom niet in een van de methoden, maar direct in de klasse. Op deze manier kunnen beide methoden gebruik maken van de variabele.
In feite vormen deze variabelen het permanente geheugen van de app: de variabelen die boven in de klasse zijn gedeclareerd zijn in alle methoden beschikbaar. Het zijn deze variabelen die samen het `KlikkerApp`-object vormen (bedenk: een object is een groepje variabelen dat bij elkaar hoort). Alle methoden in de klasse hebben zo'n object onder handen, en *daarom* mogen ze de variabelen gebruiken. Om aan te geven dat de variabele afkomstig is uit het object-dat-de-methode-onderhanden-heeft, schrijven we dat het onderdeel is van het object `this`, zoals in `this.teller` en `this.knop`.
Variabelen die bovenin de klasse zijn gedeclareerd heten *member-variabelen*: ze zijn de onderdelen van het object dat door de klasse wordt beschreven. Dit in tegenstelling tot variabelen die *in* een methode zijn gedeclareerd (zoals de variabele `scherm` in de HalloApp): dat zijn tijdelijke variabelen die alleen geldig zijn zolang de methode bezig is.

Strings en getallen
Bij het vergelijken van deze twee opdrachten:

```
knop.Text     = "Klik hier!";
knop.TextSize = 40;
```

valt het op dat de waarde van een eigenschap soms een getal is, en soms een tekst. Zo'n tekst wordt een *string* genoemd, naar de beeldspraak van 'aan een touwtje geregen letters'. Elke eigenschap heeft z'n eigen type; in deze twee voorbeelden zijn die types `string` en `int`.
Het type `int` is het type van gehele getallen, dat is getallen zonder komma en decimalen. De naam van dit type is kort voor *integral number*, dat is: geheel getal.
In een programma kun je waarden van het type `int` opschrijven met cijfertjes: dat spreekt voor zich. Een waarde van het type `string` moet tussen dubbele aanhalingstekens staan. Alle symbolen daartussen, inclusief de spatie en het uitroepteken, worden letterlijk genomen en komen dus allemaal op de knop te staan.
In de opdracht waarmee de tekst op de knop wordt aangepast wordt de string ter plaatse opgebouwd uit twee onderdelen:

```
knop.Text = this.teller.ToString() + " keer geklikt";
```

Hier wordt de tekst tussen de aanhalingstekens weer letterlijk genomen. Let op de spatie aan het begin: zonder die spatie zou het getal direct tegen de tekst aan komen te staan. De waarde van `this.teller` is een getal: deze variabele is immers als `int` gedeclareerd. Door deze waarde onder handen te nemen door de methode `ToString` wordt hij naar een string geconverteerd. De string die daar het resultaat van is, en de letterlijke string tussen aanhalingstekens, worden met de `+` operator aan elkaar geplakt. In de context van twee strings is de `+` operator dus niet 'optellen', maar 'samenvoegen'. De string die daar het resultaat van is, wordt gebruikt als het nieuwe opschrift van de knop.

Starten en her-starten van apps
Hoe sluit je het uitvoeren van een app eigenlijk af? Dat kan op het eerste gezicht op twee manieren:

- met de 'back' knop van de telefoon
- met de 'home' knop van de telefoon

In beide gevallen kom je dan meestal in het hoofdmenu van waaruit je de app had opgestart. Er is echter een verschil tussen deze twee manieren om de app te sluiten! Je merkt dat als je de KlikkerApp een tijdje gebruikt, afsluit, en daarna weer opstart.
Als je de app had afgesloten met 'back', dan is hij definitief gestopt. Start je de app daarna weer opnieuw op, dan begint de telling weer bij 0. Bij het opstarten wordt namelijk een nieuw `KlikkerApp`-object aangemaakt, waarvan dan om te beginnen weer `OnCreate` wordt aangeroepen.
Als je de app had afgesloten met 'home', dan wordt hij tijdelijk gepauzeerd. Je kunt tussendoor andere apps gebruiken, maar als je de KlikkerApp weer opstart, dan is de telling gewoon nog waar

hij gebleven is. In deze situatie wordt er namelijk *niet* een nieuw object aangemaakt, en wordt ook `OnCreate` niet aangeroepen.
Helemaal gegarandeerd is dit gedrag niet. Het operating system kan er voor kiezen om gepauzeerde apps zelfstandig te killen. Dat gebeurt echter alleen als daar een dringende aanleiding voor is, bijvoorbeeld gebrek aan geheugen of gebrek aan energie.

Roteren van apps
Dankzij de ingebouwde rotatiesensor die veel devices hebben, kunnen apps zich aanpassen aan de manier waarop de gebruiker het scherm vasthoudt: verticaal of horizontaal ('portrait mode' of 'landscape mode', naar de vorm die portretten en schilderijen van een landschap typisch hebben). Ook onze HalloApp en KlikkerApp vertonen dit gedrag, vooropgesteld dat het apparaat waarop ze draaien inderdaad een rotatiesensor heeft.
Eigenlijk is dat wel mooi: de gebruiker kan de app zo nog flexibeler gebruiken. Het is misschien wel een last voor de programmeur dat die er niet bij voorbaat van uit kan gaan dat het scherm een bepaalde afmeting heeft. Maar dat kan toch al niet vanwege de vele verschillende schermen die er gebruikt worden.
In het geval van de KlikkerApp komen we echter voor een onaangename verrassing te staan: bij het roteren van het scherm wordt de app namelijk helemaal opnieuw opgestart: een nieuw object, een nieuwe aanroep van `OnCreate`; bijgevolg begint de telling opnieuw, net zoals dat bij de 'back' knop van de telefoon gebeurde. Roteren van het apparaat is dus nogal een drastische gebeurtenis!
Er zijn drie manieren om hier iets aan te doen:

- De gebruiker kan, via het instellingen-menu van het device, het rotatiegedrag helemaal uitschakelen.
- De programmeur kan, via het `ActivityAttribute` in de header van de klasse, specificeren dat het programma altijd in portrait-, dan wel landscape-mode wordt uitgevoerd. In het volgende hoofdstuk geven we daar een voorbeeld van.
- De programmeur kan de `Bundle` parameter van `OnCreate` inspecteren. Daaraan kun je zien of het een rotatie-herstart betreft. De toestand van het programma (bijvoorbeeld: de waarde van `teller`) in het vorige leven kan in zo'n `Bundle` bewaard blijven. Je moet dat dan wel expliciet uitprogrammeren. Bij elke verandering van de teller moet je dit dan ook documenteren in de `Bundle`, ten bate van een eventueel volgend leven van de app. Het voert een beetje te ver om dat hier nu helemaal te gaan doen, maar het geeft in ieder geval aan waar die mysterieuze `Bundle`-parameter voor bedoeld is.

3.2 Een kleurenmixer

Mixer: een app om kleuren te mixen
In deze sectie bespreken we de app 'RGB-Mixer'. Met drie schuifregelaars kan de gebruiker een kleur mixen met de gewenste hoeveelheid rood, groen en blauw. De app toont de resulterende kleur, en geeft ook de mengverhouding aan, zodat de kleur gemakkelijk in bijvoorbeeld een webpagina kan worden gebruikt. Als je creativiteit tekortschiet kun je op een knop drukken om een willekeurige kleur te genereren (en nog eens, en nog eens, net zolang totdat je de kleur mooi vindt).
Het programma staat in listing 4 en listing 5 (dit is één bestand, maar het is te veel om op een bladzijde te passen). In figuur 8 is de app in werking te zien. blz. 33 blz. 34

SeekBar: een view als schuifregelaar
Zoals gebruikelijk wordt de userinterface van het programma opgebouwd in `OnCreate`. De schuifregelaars zijn objecten van het type `SeekBar`. We maken er drie, met de namen `rood`, `groen` en `blauw`. Zo'n `SeekBar` heeft een eigenschap `Progress`, waarmee de positie van het schuivertje kan worden vastgelegd of opgevraagd, en een eigenschap `Max`, die de waarde bij de maximale positie van het schuivertje bepaalt. Verder is er een event `ProgressChanged`, zodat we een event-handler kunnen registreren. We registreren voor elk van de schuifregelaars dezelfde methode `veranderd`. Deze methode zal dus elke keer worden aangeroepen als de gebruiker met zijn vingers aan een van de schuifregelaars zit. Als de gebruiker langzaam beweegt, zal dat meerdere keren achter elkaar gebeuren.
De layout van de drie schuifregelaars en de `Button` (die natuurlijk ook weer een `Click` event-handler heeft) wordt bepaald met een `LinearLayout`. Omdat we de `Orientation` daarvan `Vertical` hebben gemaakt, worden de vier views verticaal gestapeld. Elk van de drie schuifregelaars krijgt

Figuur 8: De app Mixer in werking

een toepasselijke achtergrondkleur.
Als kleine variatie op de vorige keer dat we `LinearLayout` gebruikten, specificeren we nu ook nog wat extra opties die de afmetingen van de views bepalen. Zo leggen we de hoogte van de schuifregelaars vast op 120 beeldpunten (de default-grootte is namelijk aan de kleine kant). De breedte groeit mee met de totaal beschikbare ruimte, omdat we hierbij `MatchParent` hebben gekozen. De marge onder de views zetten we op 30, zodat er wat ruimte tussen de schuifregelaars komt.

Color: type van een kleur-object
Tot nu toe hebben we steeds constante kleuren gebruikt, waarvan er een handjevol in de library zijn gedefinieerd: `Color.Red`, `Color.Yellow`, enzovoorts. Maar het is ook mogelijk om eigen kleuren te maken. Daartoe kunnen we een *variabele* van het type `Color` declareren:

```
Color kleur;
```

Omdat een kleur een object is (een groepje variabelen dat bij elkaar hoort), moeten we het object ook aanmaken:

```
kleur = new Color(r, g, b);
```

Hierbij zijn `r`, `g` en `b` de hoeveelheid rood, groen en blauw die in de mengkleur aanwezig moeten zijn, elk op een schaal van 0 tot en met 255. De aldus gemengde kleur kunnen we, net zoals we dat eerder met constante kleuren hebben gedaan, gebruiken als achtergrondkleur van de knop:

```
knop.SetBackgroundColor(kleur);
```

Omdat de mengkleur hier maar één keer gebruikt wordt, is het zelfs niet nodig om het nieuwe `Color`-object eerst in een variabele op te slaan. In de eindversie van het programma geven we de mengkleur direct mee als parameter van `SetBackgroundColor`:

```
knop.SetBackgroundColor(new Color(r,g,b));
```

blz. 34 Dit alles staat in de body van de methode `veranderd` in listing 5, die is geregistreerd als event-handler van de schuifregelaars. Daardoor wordt de kleur van de button onmiddellijk aangepast zodra de gebruiker de instellingen van de schuifregelaars wijzigt.

Random: type van een random-generator
De methode `kies` is de event-handler van het `Click`-event van de button `knop`. Deze methode wordt dus aangeroepen als de gebruiker de knop indrukt. Het is de bedoeling dat er dan een willekeurige kleur wordt uitgekozen.

```
using System;           // vanwege EventArgs, Random
using Android.OS;       // vanwege Bundle
using Android.App;      // vanwege Activity
using Android.Widget;   // vanwege SeekBar, Button, LinearLayout
using Android.Graphics; // vanwege Color

[ActivityAttribute(Label = "RGB-Mixer", MainLauncher = true)]
public class MixerApp : Activity
{
    SeekBar rood, groen, blauw;
    Button knop;

    protected override void OnCreate(Bundle b)
    {
        base.OnCreate(b);

        LinearLayout stapel;
        stapel  = new LinearLayout(this);
        rood  = new SeekBar(this);
        groen = new SeekBar(this);
        blauw = new SeekBar(this);
        knop  = new Button (this);

        stapel.Orientation = Orientation.Vertical;
        rood.Max  = 255;
        groen.Max = 255;
        blauw.Max = 255;
        knop.TextSize = 30;
        knop.Text = "mix een kleur";

        rood .SetBackgroundColor(Color.Red);
        groen.SetBackgroundColor(Color.Green);
        blauw.SetBackgroundColor(Color.Blue);

        rood .ProgressChanged += this.veranderd;
        groen.ProgressChanged += this.veranderd;
        blauw.ProgressChanged += this.veranderd;
        knop.Click += this.kies;

        LinearLayout.LayoutParams par;
        par = new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MatchParent, 120);
        par.BottomMargin = 30;
        stapel.AddView(rood,  par);
        stapel.AddView(groen, par);
        stapel.AddView(blauw, par);
        stapel.AddView(knop);
        this.SetContentView(stapel);
    }
```

Listing 4: Mixer/MixerApp.cs, deel 1 van 2

```
    public void veranderd(object o, EventArgs ea)
    {
        int r, g, b;
        r = rood .Progress;
        g = groen.Progress;
        b = blauw.Progress;

        knop.Text = $"R={r} G={g} B={b}  RGB=0x{r:X2}{g:X2}{b:X2}\nmix een kleur";
        knop.SetBackgroundColor(new Color(r, g, b));
    }
    public void kies(object o, EventArgs ea)
    {
        Random genereer;
        genereer = new Random();
        rood .Progress = genereer.Next(256);
        groen.Progress = genereer.Next(256);
        blauw.Progress = genereer.Next(256);
    }
}
```

Listing 5: Mixer/MixerApp.cs, deel 2 van 2

Het genereren van een willekeurig getal gaat met behulp van een *random generator*. Het gebruik daarvan is eenvoudig: een random-generator is een object van het type `Random`. Zoals elk object kun je deze aanmaken door een variabele te declareren, en het object met `new` te creëren:

```
Random genereer;
genereer = new Random();
```

Daarna kun je de methode `Next` aanroepen op elk moment dat je een random getal nodig hebt. We doen dat drie keer, om de hoeveelheid rood, groen en blauw in de mengkleur te bepalen. De parameter geeft aan hoe groot het getal (net niet) mag worden: door de aanroep van `Next(256)` krijgen we een geheel getal uit de range van 0 tot en met 255.
De gegenereerde getallen gebruiken we om de `Progress`-eigenschap van de schuifregelaars te veranderen. Dat wordt direct zichtbaar voor de gebruiker, want veranderen van de `Progress`-eigenschap met een toekenningsopdracht verschuift ook automatisch het schuivertje. Op zijn beurt genereert dat weer een `ProgressChanged`-event, en omdat we de methode `veranderd` hebben geregistreerd als event-handler van dat event, zal ook die methode vervolgens worden aangeroepen.

Hexadecimale notatie
Op het eerste gezicht is het wat merkwaardig dat de bovengrens voor kleurtinten die bij het construeren van een `Color` het getal 256 is. Dit is ongeveer het aantal kleurtinten dat het menselijk oog kan onderscheiden, dus op zich is het niet zo gek, maar waarom heeft de maker van de klasse `Color` voor zo'n krom getal gekozen en niet voor 100, 200 of 250?
In feite is 256 helemaal niet zo'n krom getal, als we niet in het tientallig stelsel rekenen maar in het zestientallig stelsel. Het zestientallig stelsel wordt gehanteerd door Martianen, die zoals bekend niet tien maar zestien vingers hebben. Zij kennen daarom, net als wij, de cijfers `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` en `9`, maar daarna komen er nog zes cijfers, alvorens ze overgaan tot een tweede positie om het aantal zestientallen aan te duiden. De precieze vorm van de zes extra Martiaanse cijfers is wat lastig te tekenen, dus daarom zullen we ze gemakshalve aanduiden met `A`, `B`, `C`, `D`, `E` en `F`. Dit zijn dus hier even geen letters, maar extra cijfers! Het Martiaanse cijfer `B` komt overeen met onze elf, `C` is twaalf, enzovoorts.
Pas als de Martianen al hun vingers hebben afgeteld, gaan ze over naar een tweecijferig getal: dus na `E` en `F` komt het getal `10`. Dit Martiaanse getal komt overeen met ons getal zestien. Daarna gaat het verder met `11` (zeventien), en zo door tot `19` (vijfentwintig), en daarna ook nog `1A` (zesentwintig) tot `1F` (eenendertig). Pas dan komt `20` (tweeëndertig). En op zeker moment komen ze bij `30` (achtenveertig), `40` (vierenzestig), en jawel: `A0` (honderdzestig). Nu worden

Martianen tamelijk oud, maar op zeker moment zijn ook zij door de twee-cijferige getallen heen: na `FF` (tweehonderdvijfenvijftig) komt `100` (tweehonderdzesenvijftig). Dus onze 255 is precies het grootste getal dat de Martianen nog net met twee cijfers kunnen noteren.
De kleinste eenheid van computergeheugen is de bit: aan of uit, stroom of geen stroom, magnetisch of niet magnetisch, licht of donker, of hoe de informatie maar wordt opgeslagen. Je kunt dit ook noteren als 0 en 1, de spreekwoordelijke 'nullen en enen' waarmee de computer rekent. Met twee bits kun je $2 \times 2 = 4$ combinaties maken: 00, 01, 10, en 11. We hebben hier te maken met het tweetallig stelsel (een computer heeft twee vingers): na de cijfers 0 en 1 is het al op, en moeten we naar een tweede positie: 10 tweetallig is twee, 11 is drie, en dan is het al weer op.
Met 3 bits kun je $2^3 = 8$ combinaties maken: 000, 001, 010, 011, 100, 101, 110, 111. Historisch is het zo gegroeid dat acht bits worden gegroepeerd tot wat we een byte noemen. In een byte kun je dus $2^8 = 256$ verschillende getallen opslaan, van 00000000 tot en met 11111111. Het wordt wel wat onoverzichtelijk met die lange rijen nullen en enen. Het is makkelijker om ze in groepjes van vier te pakken, en te noteren met een Martiaans cijfer: van `00` tot en met `FF`.
Martianen bestaan niet echt. Maar hun cijfers zijn wel handig als je bytes wilt aanduiden, omdat je dan precies met twee cijfers af kunt. De technische term voor dit zestientallig stelsel is *hexadecimaal* (van het Griekse hexa=zes en deka=tien). Je kunt de hexadecimale getal-notatie gebruiken in C#. Om duidelijk aan te geven dat we met hexadecimale cijfers te maken hebben en niet met gewone decimale cijfers, moet zo'n hexadecimaal getal beginnen met `0x`. Dus `10` is gewoon tien, maar `0x10` is zestien, en `0x1A` is zesentwintig.
Hexadecimale getallen worden vaak gebruikt om kleuren aan te duiden. Je kunt de drie benodigde bytes dan mooi overzichtelijk aanduiden met zes hexadecimale cijfers, meestal in de volgorde rood-groen-blauw. Dat wordt bijvoorbeeld gebruikt in HTML, de codeertaal voor webpagina's (al wordt daar dan weer niet `0x`, maar `#` gebruikt als prefix voor hexadecimale getallen).
Met onze kleurenmixer wordt het makkelijk gemaakt om de kleur direct in HTML te gebruiken, omdat naast de decimale representatie van rood, groen en blauw ook het 6-cijferige hexadecimale getal van de totale kleur wordt getoond. Die kun je direct overnemen in HTML.

String-formatting

Hoe kunnen we de waarden van de drie kleurcomponenten overzichtelijk aan de gebruiker tonen (om te beginnen eerst maar eens in de decimale notatie)? Op dezelfde manier als we in de Klikker app hebben gedaan, zou dat zo kunnen:

```
knop.Text = "R=" + r.ToString() + " G=" + g.ToString() + " B=" + b.ToString;
```

waarbij `r`, `g` en `b` de `int`-variabelen zijn die we willen laten zien. Let op de combinatie van stukjes tekst tussen aanhalingstekens (die wordt letterlijk gebruikt, inclusief de spatie), en expressies als `r.ToString()` (daarvan wordt de huidige waarde bepaald en gebruikt in de string). Alle onderdelen worden met `+` aan elkaar gekoppeld.
Hoewel dit conceptueel de eenvoudigste manier is, is het in de praktijk nogal een gedoe om teksten waarin waarden van variabelen woren gebruikt samen te stellen. Daarom is er een notatie beschikbaar waarmee dit gemakkelijker kan worden opgeschreven. De notatie is nieuw in C# versie 6 (van juli 2015), en vereist dus de 2015 editie van de compiler om te kunnen gebruiken. Het gaat zo:

```
knop.Text = $"R={r} G={g} B={b}";
```

Je kunt dus volstaan met één lange tekst, waarvan de letterlijkheid wordt onderbroken door variabelen (of zelfs hele berekeningen) tussen accolades te zetten. Het is ook niet meer nodig om `ToString` steeds aan te roepen, dat gebeurt automatisch. De hele string moet vooraf worden gegaan door een dollar-teken om dit mogelijk te maken. Deze notatie staat bekend als een *geïnterpoleerde string*, omdat de letterlijke teksten en de expressies door elkaar heen staan.
Om het nog flexibeler te maken, mag je tussen de accolades ook nog extra aanwijzingen schrijven om het getalstelsel en het aantal gewenste cijfers te bepalen. Dit is net wat we nodig hebben om tweecijferige hexadecimale getallen te maken:

```
knop.Text = $"RGB={r:X2}{g:X2}{b:X2}";
```

Of alles samen in één geheel:

```
knop.Text = $"R={r} G={g} B={b} RGB=0x{r:X2}{g:X2}{b:X2}\nmix een kleur";
```

De code `\n` staat hierbij voor een overgang naar een nieuwe regel: dat mag in alle strings, niet alleen in geïnterpoleerde strings.

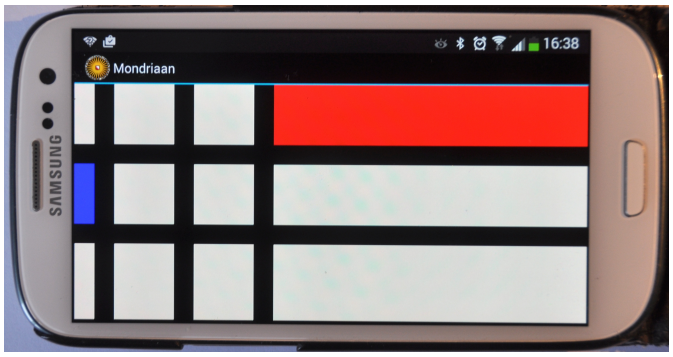
Hoofdstuk 4

Methoden om te tekenen

4.1 Een eigen subklasse van View

Grafische uitvoer

Door het combineren van `View`-objecten, zoals `TextView` en `Button`, in een `LinearLayout` kun je een complexe scherm-layout opbouwen. Maar het blijven wel voorgedefinieerde vormen, en je bent dus niet helemaal vrij om te bepalen hoe het scherm er uit komt te zien.
Gelukkig kun je zelf nieuwe soorten `View`-objecten maken, als je bent uitgekeken op de standaard-views. Je kunt daar dan weer libraries van bouwen, en natuurlijk kun je ook een library vol met handige `View`-objecttypen van iemand anders overnemen.
In deze sectie bekijken we het programma Mondriaan, dat gebruik maakt van de mogelijkheid om een vrije tekening te maken. Het programma maakt een schilderij in de Stijl van Mondriaans "compositie met rood en blauw". Het plaatje is *niet* in een bitmap opgeslagen (dan hadden we het eenvoudig in een `ImageView` kunnen laten zien), maar wordt door het programma zelf getekend. Het programma staat in listing 6; in figuur 9 is dit programma in werking te zien. blz. 38



Figuur 9: De app Mondriaan in werking

Een eigen subklasse van View

Het programma bestaat ditmaal uit twee klassen: `MondriaanApp` en `MondriaanView`. De klasse `MondriaanApp` is de gebruikelijke subklasse van `Activity`, waarin de methode `OnCreate` de userinterface opbouwt. De opdrachten in deze methode lijken sterk op die in eerdere voorbeelden:

- de HalloApp, waarin een `TextView` werd neergezet
- de KlikkerApp, waarin een `Button` werd neergezet

In deze MondriaanApp gebruiken we echter niet een bestaande `View`, maar een zelfgemaakte `MondriaanView`.
In de tweede klasse in dit programma wordt gedefinieerd wat zo'n `MondriaanView` is. Bekijk als eerste de header van deze klasse:

```
public class MondriaanView : View
```

Achter de dubbelepunt staat dat onze klasse een subklasse is van de library-klasse `View`. Daarom geniet onze klasse alle voorrechten die elke `View` heeft. Een object ervan kan bijvoorbeeld worden meegegeven bij de aanroep van `SetContentView`.

```
/* Dit programma tekent een Mondriaan-achtige
   "Compositie met rood en blauw"
*/
using Android.OS;         // vanwege Bundle
using Android.App;        // vanwege Activity
using Android.Views;      // vanwege View
using Android.Graphics;   // vanwege Color, Paint, Canvas
using Android.Content;    // vanwege Context
using Android.Content.PM; // vanwege ScreenOrientation

[ActivityAttribute(Label = "Mondriaan", MainLauncher = true,
                   ScreenOrientation = ScreenOrientation.Landscape)]
public class MondriaanApp : Activity
{
    protected override void OnCreate(Bundle b)
    {
        base.OnCreate(b);
        MondriaanView schilderij;
        schilderij = new MondriaanView(this);
        this.SetContentView(schilderij);
    }
}
public class MondriaanView : View
{
    public MondriaanView(Context c) : base(c)
    {   this.SetBackgroundColor(Color.AntiqueWhite);
    }

    protected override void OnDraw(Canvas canvas)
    {
        base.OnDraw(canvas);

        int breedte, hoogte, balk, x1, x2, x3, y1, y2;
        breedte = this.Width;
        hoogte  = this.Height;
        x1 =  50; x2 = 250; x3 = 450;
        y1 = 150; y2 = 350;
        balk = 50;

        Paint verf;
        verf = new Paint();

        // zwarte balken
        verf.Color = Color.Black;
        canvas.DrawRect(x1, 0, x1+balk, hoogte, verf);
        canvas.DrawRect(x2, 0, x2+balk, hoogte, verf);
        canvas.DrawRect(x3, 0, x3+balk, hoogte, verf);
        canvas.DrawRect(0, y1, breedte, y1+balk, verf);
        canvas.DrawRect(0, y2, breedte, y2+balk, verf);

        // gekleurde vlakken
        verf.Color = Color.Blue;
        canvas.DrawRect(0, y1+balk, x1, y2, verf);
        verf.Color = Color.Red;
        canvas.DrawRect(x3+balk, 0, breedte, y1, verf);
    }
}
```

Listing 6: Mondriaan/MondriaanApp.cs

In de klasse `View` is het zo geregeld dat de methode `OnDraw` automatisch wordt aangeroepen op het moment dat de view getekend moet worden. In onze subklasse `MondriaanView` kunnen we een eigen invulling geven aan `OnDraw` door deze methode met `override` opnieuw te definiëren. Dit is hetzelfde mechanisme als de her-definitie van de methode `OnCreate` in een subklasse van `Activity`. En net als daar is ook nu weer de eerste opdracht in de body van `OnDraw` een aanroep van de oorspronkelijke versie:

```
protected override void OnDraw(Canvas canvas)
{
    base.OnDraw(canvas);
```

Canvas: iets om op te schilderen

Het operating system dat de methode `OnDraw` aanroept, geeft daarbij een object van het type `Canvas` mee als parameter. De definitie van de methode moet daarom in zijn header aangeven zo'n `Canvas`-object te verwachten. In de body mogen we dat object gebruiken, en dat komt goed uit: op een canvas kun je namelijk tekenen! Het is letterlijk het 'schilderslinnen' waarop we een schilderij kunnen maken. In de klasse zit daartoe een aantal methoden. Bij aanroep daarvan worden als parameter nadere details over de positie en/of de afmeting van de te tekenen figuur meegegeven. Met een `Canvas`-object `c` onder handen kun je bijvoorbeeld de volgende methoden aanroepen:

- `c.DrawLine(x1, y1, x2, y2, verf);` tekent een lijn tussen twee punten
- `c.DrawRect(links, boven, rechts, onder, verf);` tekent een rechthoek op de aangegeven positie
- `c.DrawCircle(midx, midy, straal, verf);` tekent een cirkel met aangegeven middelpunt
- `c.DrawOval(links, boven, rechts, onder, verf);` tekent een ovaal binnen de aangegeven rechthoek
- `c.DrawText(tekst, x, y, verf);` tekent een tekst op de aangegeven plek
- `c.DrawColor(kleur);` vult de hele canvas met de aangegeven kleur
- `c.DrawBitmap(bitmap, x, y, verf);` 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 10.

Als laatste parameter hebben we steeds de variabele `verf` meegegeven. Bij aanroep van een `Draw`-methode moet je namelijk een `Paint`-object meegeven die aangeeft hoe ('met welke verf') er geschilderd moet worden. Je kunt zo'n `Paint`-object gemakkelijk aanmaken:

```
Paint verf;
verf = new Paint();
```

Zoals te verwachten viel, heeft een `Paint` een kleur:

```
verf.Color = Color.Blue;
```

Als je met deze verf een van de `Draw`-methoden aanroept, wordt de figuur in blauwe verf getekend. Daarnaast heeft `Paint` nog een aantal andere eigenschappen die niet helemaal overeenstemmen met de verf-metafoor: de dikte van lijnen die getekend worden, het lettertype van teksten, en een schaal-factor waarmee je vergroot of verkleind kunt tekenen. Sommige eigenschappen, zoals `Stroke` voor de lijndikte, kun je direct toekennen; andere eigenschappen kun je veranderen door aanroep van methoden als `SetTypeface()`.

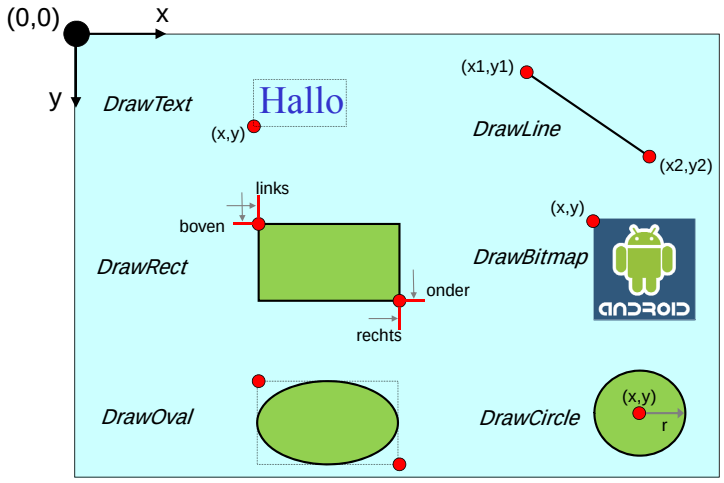
In listing 6 gebruiken we de methode `DrawRect` om een aantal rechthoeken te tekenen. Door de juiste `Paint` mee te geven worden sommige rechthoeken zwart, en andere gekleurd. blz. 38

Klassen beschrijven de mogelijkheden van objecten

Alle methoden uit de klasse `Canvas` kun je aanroepen, als je tenminste de beschikking hebt over een object van het type `Canvas`. Dat is in de body van de teken-methode geen probleem, want die methode heeft een `Canvas`-object als parameter. 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 `Canvas`-object "kan" teksten, lijnen, rechthoeken en ovalen 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



Figuur 10: Enkele methoden uit de klasse Canvas

blijkbaar onthouden wat de waarde van die property was. Dat klopt ook wel met de manier waarop
blz. 2 in sectie 1.2 over objecten werd gesproken: een object is een groepje variabelen. Inmiddels hebben
blz. 2 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 `Canvas`-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!".

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 onze klasse `MondriaanView` hebben we ook een constructormethode gedefinieerd:

```
public MondriaanView(Context c) : base(c)
{   this.SetBackgroundColor(Color.AntiqueWhite);
}
```

De constructie van een object van (een subklasse van) `View` is een goed moment om de achtergrondkleur er van vast te leggen.

Syntactisch wijkt de methode-header van een constructor-methode af van andere methodes: er staat *niet* het woord `void` in de header, en ook niet `override`. Aan het eind van de header, maar nog voor de accolade-openen van de body, is er de gelegenheid om de constructor-methode van de klasse waarvan dit een subklasse is aan te roepen. In dit geval is dat de klasse `View`, die bij deze gelegenheid wordt aangeduid met `base`.

De constructormethode van `View` heeft een `Context`-parameter nodig. Daarom geven we de constructor van `MondriaanView` ook een `Context`-parameter, zodat we die meteen aan `base` kunnen doorgeven.

De manier waarop een constructormethode zijn oorspronkelijke versie aanroept wijkt dus iets af van gewone methoden: ook daar kun je de versie van de methode in de oorspronkelijke klasse aanroepen, maar dan *in de body*, meestal als eerste opdracht. Dit is het geval bij

```
protected override void OnDraw(Canvas canvas)
{
    base.OnDraw(canvas);
    ...overige opdrachten...
}
```

De constructormethode van `MondriaanView` wordt aangeroepen vanuit de methode `OnCreate` in `MondriaanApp`:

```
MondriaanView schilderij;
schilderij = new MondriaanView(this);
```

Je ziet hier dat `this`, dat is het object van de `MondriaanApp` zich blijkbaar kan gedragen als een `Context`. Op een soortgelijke manier konden de subklassen van `View` die we in eerdere programma's hebben gebruikt, zoals `TextView`, `LinearLayout` en `Button`, zich gedragen als `View`.

4.2 Variabelen

Variabele: declaratie, toekenning, gebruik

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

In de HalloApp in listing 1 was er een variabele om een `TextView` in op te slaan, zodat we er daarna properties van kunnen veranderen, en hem meegeven aan `SetContentView`: blz. 15

```
TextView scherm;
scherm = new TextView(this);
this.SetContentView(scherm);
```

In de KlikkerApp in listing 3 was er een variabele `teller` om een getal in te bewaren, dat het aantal kliks bijhoudt: blz. 28

```
int teller;
teller = teller + 1;
knop.Text = teller.ToString();
```

Het type van de variabele was in het eerste geval de klasse `TextView`, in het tweede geval het standaardtype `int`.

Variabelen van type int

In het voorbeeldprogramma in listing 6 worden drie vertikale zwarte balken getekend. Dat had gekund met de volgende opdrachten: blz. 38

```
canvas.DrawRect( 10, 0, 20, 100, verf);
canvas.DrawRect( 50, 0, 60, 100, verf);
canvas.DrawRect( 90, 0,100, 100, verf);
```

De eerste twee getallen geven de plaats aan van de linkerbovenhoek van de balken: 10, 50 en 90 beeldpunten vanaf de linkerrand, tegen de bovenrand aan. De laatste twee getallen die van de rechter-onderhoek.

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 x-coördinaat van de rechteronderhoek veranderden. Dat is nogal een gedoe.

Een oplossing is het gebruik van variabelen. We introduceren twee variabelen voor de dikte van de balk en de hoogte ervan, laten we zeggen `balk` en `hoogte`:

```
canvas.DrawRect( 10, 0, 10+balk, hoogte, verf);
canvas.DrawRect( 50, 0, 50+balk, hoogte, verf);
canvas.DrawRect( 90, 0, 90+balk, hoogte, verf);
```

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 een methode, parameters worden gedeclareerd tussen de haakjes in de methode-header;
- variabelen krijgen een waarde door een toekennings-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 standaardtype. Er zijn maar een handjevol standaardtypen. Andere standaardtypen die we nog zullen tegenkomen zijn `float` (getallen die wel cijfers achter de 'drijvende 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 `DrawRect` is zinvol met een `Canvas`-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 `DrawText` 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.

4.3 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 toekenningsopdracht 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 of vermenigvuldiging een rol spelen, zoals `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
blz. 38 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 vertikale 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 beginnen). 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:

```
canvas.DrawRect(x1, 0, x1+balk, hoogte, verf);
canvas.DrawRect(x2, 0, x2+balk, hoogte, verf);
canvas.DrawRect(x3, 0, x3+balk, hoogte, verf);
canvas.DrawRect(0, y1, breedte, y1+balk, verf);
canvas.DrawRect(0, y2, breedte, y2+balk, verf);
```

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-coordinaat die één balkbreedte groter is dan de y-coordinaat van de eerste balk:

```
verf.Color = Color.Blue;
canvas.DrawRect(0, y1+balk, x1, y2, verf);
```

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 naaast 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 toekenningsopdrachten 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.

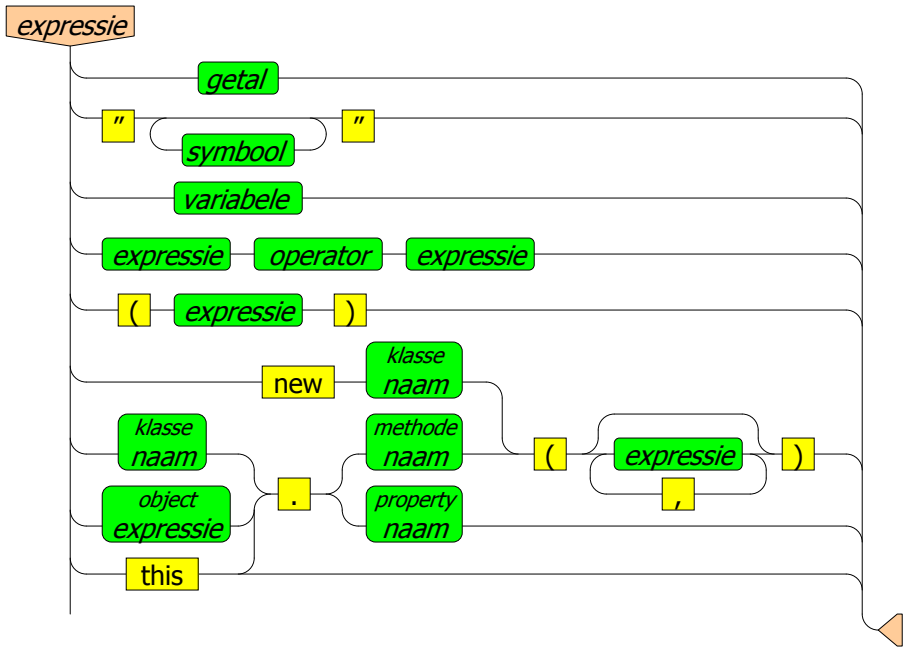
Expressies met een object-waarde
Het begrip 'waarde' van een expressie is niet beperkt tot getal- en string-waarden. Expressies kunnen van elk type zijn waarvan je ook variabelen kunt declareren, dus naast de standaardtypen `int` en `string` kunnen dat ook object-typen zijn, zoals `Color`, `Form`, of `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 TextView(this)` heeft een `TextView`-object als waarde, de expressie `new Color(100,150,200)` heeft een `Color`-object als waarde.

Syntax van expressies
De syntax van expressies tot nu toe wordt samengevat in het syntax-diagram in figuur 11. Er is speciale syntax voor een constant getal en een constante string (tussen dubbele 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. Vaak 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. Dit was bijvoorbeeld het geval bij de aanroep van `new TextView(this)`.

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:

> 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:



Figuur 11: (Vereenvoudigde) syntax van een expressie

- 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

```
this.SetContentView(scherm);
```

Een methode-aanroep die een expressie vormt is

```
teller.ToString()
```

In dit geval staat er dus geen puntkomma achter! Deze expressie moet deel uitmaken van een groter geheel, bijvoorbeeld als rechterkant van een toekenningsopdracht:

```
knop.Text = teller.ToString() + " keer geklikt";
```

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

Void-methoden
Of een methode bedoeld is om aan te roepen als opdracht of als expressie, wordt bepaald door de auteur van de methode. Bij `ToString` is het duidelijk de bedoeling dat de methode-aanroep een string als waarde heeft, en deze aanroep is dan ook een expressie. De methode `SetContentView` 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-methoden 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.

4.4 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 ervan 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-tekens 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

```
scherm.Text = "hallo";
```

en

```
scherm.Text = "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.

4.5 Declaraties met initialisatie

Combineren van declaratie en toekenning

Aan alle variabelen zul je ooit een waarde toekennen. De variabele moet een waarde hebben gekregen voordat je hem in een berekening gebruikt. Als je dat vergeet, geeft de compiler een foutmelding: 'use of unassigned local variable'.

Variabelen die je niet in een berekening gebruikt, hoef je geen waarde te geven. Maar als je een variabele niet gebruikt, is de hele declaratie zinloos geworden. Dat is niet fout, maar wel verdacht, en daarom geeft de compiler in dat soort situaties een waarschuwing: 'the variable is declared but never used'.

Omdat een toekenning aan een variabele dus vrijwel onvermijdelijk is, is er een notatie om de declaratie van een variabele met de eerste toekenning aan die variabele te combineren. In plaats van

```
int breedte;
breedte = 200;
```

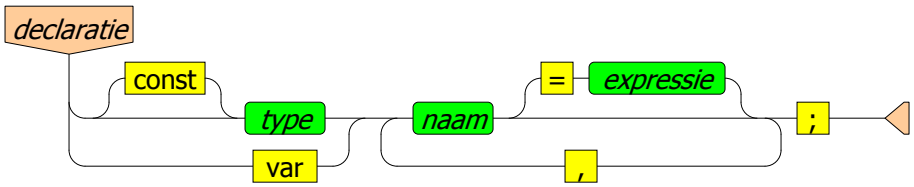
mogen we ook schrijven:

```
int breedte = 200;
```

Dit kan/mag alleen bij de *eerste* toekenning aan de variabele. Het is dus niet de bedoeling dat je bij elke toekenning opnieuw het type erbij gaat schrijven. Je zou de variabele dan steeds opnieuw declareren, en de compiler zal reageren met een foutmelding: 'local variable is already defined'.

Syntax van declaraties

De eerste toekenning aan een variabele heet een *initialisatie*. Dit is de uitgebreide syntax van declaraties waarin zo'n initialisatie is opgenomen.



const: declaratie van variabele die niet varieert

Variabelen kunnen veranderen – het woord zegt het al. De waarde verandert bij elke toekenningsopdracht aan die variabele.

Soms is het handig om een bepaalde waarde een naam te geven, als die waarde veel in een programma voorkomt. In een programma met veel wiskundige berekeningen is het bijvoorbeeld handig om eenmalig te schrijven:

```
double PI = 3.1415926535897;
```

Daarna kun je waar nodig de variabele `PI` gebruiken, in plaats van elke keer dat hele getal uit te schrijven. Het is in dit geval niet de bedoeling dat de variabele later in het programma nog wijzigt – echt variabel is deze variabele dus niet. Om er voor te zorgen dat dat niet per ongeluk zal gebeuren (bijvoorbeeld door een tikfout bij het intikken van het programma), kun je bij de declaratie met het keywoord `const` aangeven dat de variabele helemaal niet varieert, maar constant blijft. De variabele moet dan meteen bij de declaratie een waarde krijgen, en er mag later niet meer een nieuwe waarde aan worden toegekend. Die waarde mag ook niet afhangen van variabelen die zelf niet `const` zijn.

var: automatische type-bepaling in declaraties

Uit het syntax-diagram blijkt ook dat je in plaats van het type het woord `var` mag schrijven. In dit geval is de initialisatie verplicht (om het schema niet te gecompliceerd te maken is dat niet in het diagram weergegeven). Het type van de variabele wordt dan automatisch bepaald aan de hand van de waarde van de initialisatie. Dus in de declaraties

```
var n = 10;
var s = "Hallo";
```

krijgt variabele `n` het type `int`, en variabele `s` het type `string`.

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.

4.6 Methode-definities

Alle methodes die we tot nu toe hebben geschreven, waren *her*-definities van methoden uit de klasse waarvan onze klasse een subklasse is. De naam was daarom steeds al bepaald door de auteur van de oorspronkelijke klasse: `OnCreate` in (onze subklasse van) `Activity`, en `OnDraw` in (onze subklasse van) `View`. Het wordt tijd om zelf eens een eigen methode te schrijven, en die ook zelf aan te roepen.
Als je een vierkant tekent met twee schuine lijntjes erbovenop heb je een simpel huisje getekend. Het voorbeeldprogramma in deze sectie tekent drie huisjes. In figuur 12 is het resultaat te zien.

Namespace: klassen die bij elkaar horen
Net als het vorige programma bestaat dit programma uit een subklasse van `Activity`, en een subklasse van `View`. We hebben deze twee klassen nu echter in aparte bestanden gezet, die te zien zijn in listing 7 en listing 8.
blz. 50
blz. 51
Omdat deze klassen elkaar nodig hebben (de activity maakt een object van de view aan) moeten ze elkaar kunnen vinden. Als ze niet in hetzelfde bestand staan gaat dat niet vanzelf. We maken daarom een `namespace` aan met de naam `Huizen`, en schrijven in beide bestanden dat de klasse zich in deze namespace bevindt. Een namespace is simpelweg een groepje klassen die elkaar mogen gebruiken zonder dat dat met `using` hoeft te worden vermeld.

Orde in de chaos
Het programma zou de drie huisjes kunnen tekenen met de volgende `OnDraw`-methode:

```
protected override void OnDraw(Canvas canvas)
{
    base.OnDraw(canvas);
    Paint verf = new Paint();
    // kleine huisje links
    canvas.DrawRect( 20, 60,  60,100, verf);
    canvas.DrawLine( 14, 66,  40, 40, verf);
    canvas.DrawLine( 40, 40,  66, 66, verf);
    // kleine huisje midden
    canvas.DrawRect( 80, 60, 120,100, verf);
    canvas.DrawLine( 74, 66, 100, 40, verf);
    canvas.DrawLine(100, 40, 126, 66, verf);
    // grote huis rechts
    canvas.DrawRect(140, 40, 200,100, verf);
    canvas.DrawLine(130, 70, 170, 10, verf);
    canvas.DrawLine(170, 10, 210, 66, verf);
}
```

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 methode-aanroepen 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 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 `DrawRect` en de twee aanroepen van `DrawLine`). Die drie opdrachten zijn dus een

goede kandidaat om in een methode te zetten; in de methode `OnDraw` komen dan alleen nog maar drie aanroepen van deze nieuwe methode te staan. De opzet van het programma wordt dus als volgt:

```
public class HuizenView : View
{
    private void tekenHuis( iets )
    {   iets .DrawRect( iets );
        iets .DrawLine( iets );
        iets .DrawLine( iets );
    }
    protected override void OnDraw(Canvas canvas)
    {
        base.OnDraw(canvas);
        iets .tekenHuis( iets );
        iets .tekenHuis( iets );
        iets .tekenHuis( iets );
    }
}
```

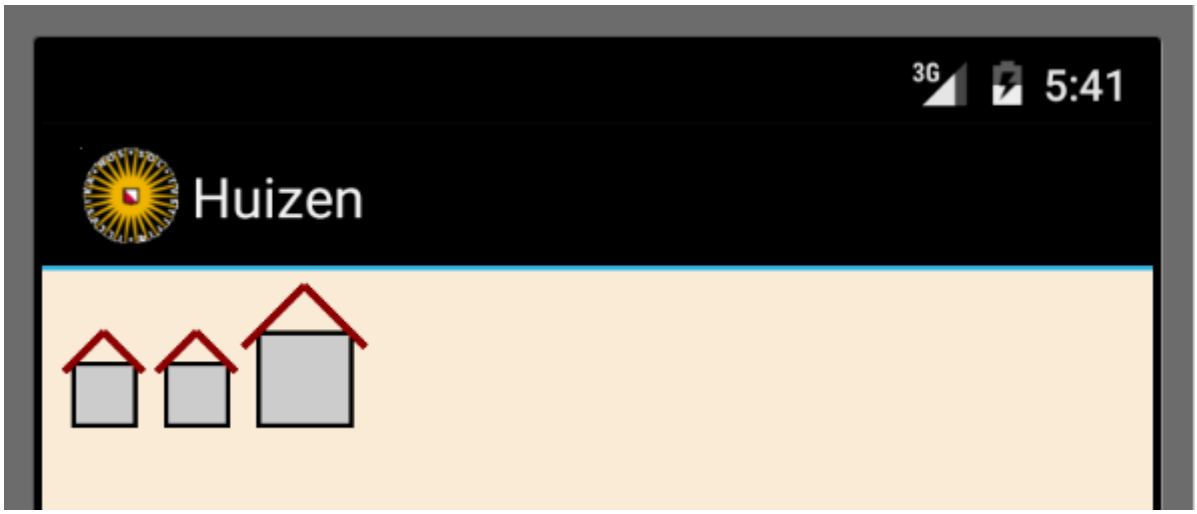
Er zijn dus twee methoden: naast de hergedefinieerde `OnDraw` is er een tweede methode die één huis tekent, en die we daarom `tekenHuis` noemen. De naam mag vrij worden gekozen, en het is een goed idee om die naam de taak van de methode te laten beschrijven.
De volgorde waarin de methoden in de klasse staan is niet van belang. De opdrachten in de body van een methode worden pas uitgevoerd als de methode wordt aangeroepen. De methode `OnDraw` wordt automatisch aangeroepen als de `View` getekend moet worden. Pas als de methode `OnDraw` een aanroep doet van de methode `tekenHuis`, worden de opdrachten in de body van de methode `tekenHuis` uitgevoerd. Als dat klaar is, gaat `OnDraw` 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 `OnDraw` wordt er een huis getekend, en pas daarna gaat het weer verder op de plaats van waaruit `OnDraw` 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 `DrawRect` 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 `DrawRect` bijvoorbeeld, krijgt een `Canvas`-object onder handen.
Tot nu toe hebben we daar het `Canvas`-object voor gebruikt dat we als parameter van `OnDraw` meekrijgen. De parameter van de methode `OnDraw` 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 `Canvas`-object beschikbaar is, en dat kunnen we doen door `tekenHuis` een `Canvas`-object als parameter te geven. In de body van `tekenHuis` kunnen we die parameter dan mooi gebruiken voor de punt in de aanroep van `DrawRect` en `DrawLine`:

```
private void tekenHuis(Canvas c, iets )
{   c.DrawRect( iets );
    c.DrawLine( iets );
    c.DrawLine( iets );
}
```

Je mag als programmeur de naam van de parameter vrij kiezen; hier hebben we de naam `c` 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 `DrawRect` schrijven we nu het `Canvas`-object `c`.
De naam van het *type* van de parameter mag je niet zomaar kiezen: het object-type `Canvas` is een bestaande bibliotheek-klasse, en die mogen we niet ineens `Linnen` of iets dergelijks gaan noemen.
Nu we in de header van de methode `tekenHuis` gespecificeerd hebben dat de eerste parameter een `Canvas`-object is, moeten we er voor zorgen dat bij aanroep van `tekenHuis` ook inderdaad



Figuur 12: Het programma HuizenApp in werking

```
using Android.OS;          // vanwege Bundle
using Android.App;         // vanwege Activity

namespace Huizen
{
    [ActivityAttribute(Label = "Huizen", MainLauncher = true)]
    public class HuizenApp : Activity
    {
        protected override void OnCreate(Bundle b)
        {
            base.OnCreate(b);
            this.SetContentView(new HuizenView(this));
        }
    }
}
```

Listing 7: Huizen/HuizenApp.cs

```
using Android.Views;      // vanwege View
using Android.Graphics;   // vanwege Color, Paint, Canvas
using Android.Content;    // vanwege Context

namespace Huizen
{
    public class HuizenView : View
    {
        public HuizenView(Context c) : base(c)
        {
            this.SetBackgroundColor(Color.AntiqueWhite);
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);
            this.tekenHuis(canvas,  20, 100, 40);
            this.tekenHuis(canvas,  80, 100, 40);
            this.tekenHuis(canvas, 140, 100, 60);
        }

        private void tekenHuis(Canvas canvas, int x, int y, int breedte)
        {
            Paint verf = new Paint();

            // Gevel van het huis
            verf.SetStyle(Paint.Style.Fill);
            verf.Color = Color.LightGray;
            canvas.DrawRect(x, y - breedte, x + breedte, y, verf);
            verf.SetStyle(Paint.Style.Stroke);
            verf.Color = Color.Black;
            verf.StrokeWidth = 3;
            canvas.DrawRect(x, y - breedte, x + breedte, y, verf);

            // Twee lijnen voor het dak
            int topx = x + breedte / 2;
            int topy = y - 3 * breedte / 2;
            int afdak = breedte / 6;

            verf.Color = Color.DarkRed;
            verf.StrokeWidth = 5;
            canvas.DrawLine(x - afdak, y - breedte + afdak, topx, topy, verf);
            canvas.DrawLine(topx, topy, x + breedte + afdak, y - breedte + afdak, verf);
        }
    }
}
```

Listing 8: Huizen/HuizenView.cs

een `Canvas`-object wordt meegegeven. De aanroep van `tekenHuis` vindt plaats vanuit de methode `OnDraw`, en daar hebben we gelukkig een `Canvas`-object beschikbaar: de parameter die `OnDraw` zelf meekrijgt. De aanroepen van `tekenHuis` komen er dus als volgt uit te zien:

```
protected override void OnDraw(Canvas canvas)
{   base.OnDraw(canvas);
    iets.tekenHuis(canvas, iets );
    iets.tekenHuis(canvas, iets );
    iets.tekenHuis(canvas, iets );
}
```

De methode `tekenHuis` wordt alleen maar door `OnDraw` 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 `OnDraw` 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 `DrawRect` heeft een `Canvas`-object onder handen, omdat `DrawRect` in de klasse `Canvas` staat.

Welnu, de methoden `OnDraw` en `tekenHuis` staan in de klasse `HuizenView`, en hebben dus blijkbaar een `HuizenView`-object onder handen. Zo'n `HuizenView`-object is in de methode `OnCreate` van `HuizenApp` gecreëerd, en de methode `OnDraw` heeft dat object onder handen. In de body van `Ondraw` 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 `OnDraw` aan te geven dat bij de aanroep van `tekenHuis` hetzelfde object onder handen genomen moet worden als dat `OnDraw` zelf al onder handen heeft:

```
protected override void OnDraw(Canvas canvas)
{   base.OnDraw(canvas);
    this.tekenHuis(canvas, iets );
    this.tekenHuis(canvas, iets );
    this.tekenHuis(canvas, 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.

4.7 Op zoek naar parameters

Parameters maken methoden flexibeler

Het administratieve werk –zorgen dat alle methoden over de benodigde Canvas- en HuizenView-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 (`DrawRect` 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 `DrawRect` en `DrawLine`.

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

```
canvas.DrawRect( 20, 60,  60, 100, verf);
canvas.DrawRect( 80, 60, 120, 100, verf);
canvas.DrawRect(140, 40, 180, 100, verf);
```

De eerste twee getallen zijn de coördinaten van de linkerbovenhoek van de rechthoek, de laatste twee getallen die van de rechteronderhoek. Omdat we vierkanten tekenen zijn de verschillen van de x-coordinaat steeds gelijk aan de verschillen van de y-coordinaat: 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 `DrawRect` die nodig heeft, berekenen we met een expressie:

```
private void tekenHuis(Canvas c, int x, int y, int br)
{   Paint verf = new Paint();
    c.DrawRect( x, y-br, x+br, y, verf);
    c.DrawLine( iets , verf);
    c.DrawLine( iets , verf);
}
protected override void OnDraw(Canvas canvas)
{   base.OnDraw(canvas);
    this.tekenHuis(canvas,  20, 100, 40);
    this.tekenHuis(canvas,  80, 100, 40);
    this.tekenHuis(canvas, 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(Canvas c, int x, int y, int br)
{   int topx, topy;
    topx  = x + br/2;
    topy  = y - 3*br / 2;
    Paint verf = new Paint();
    c.DrawRect(x, y-br, x+br, y, verf);
    c.DrawLine(x, y-br, topx, topy, verf);
    c.DrawLine(topx, topy, x+br, y-br, verf);
}
```

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 eerst 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 8 ook nog een variabele gedeclareerd die bepaalt hoe ver het dak uitsteekt naast het huis. Deze variabele `afdak` is afhankelijk van de breedte blz. 51

van het huis: een groter huis krijgt ook een groter afdak.
Ook wordt in de listing `DrawRect` tweemaal aangeroepen, met verschillende `Paint`. Bij de eerste aanroep zorgt

```
verf.SetStyle(Paint.Style.Fill);
verf.Color = Color.LightGray;
```

er voor dat het vierkant helemaal wordt opgevuld met lichtgrijze verf; voor de tweede aanroep zorgt

```
verf.SetStyle(Paint.Style.Stroke);
verf.Color = Color.Black;
```

voor een zwarte buitenrand.

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-coordinaat 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 `Paint`-object apart als parameter kunnen meegeven, want wie weet willen we later nog wel eens een huisje met een andere kleur tekenen. En dan nog een, 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 5

Objecten en methoden

5.1 Variabelen

Declaratie: aangifte van het type van een variabele

In sectie 4.2 hebben we gezien dat je de variabelen die 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 blz. 41

```
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:

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 toekenningsopdracht, zoals

```
x = 20;
```

De situatie in het geheugen wordt daarmee:

Met een tweede toekenningsopdracht 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:

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:

Merk op dat met toekenningsopdrachten 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 toekenningsopdrachten 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 = 3.14159265;
```

d [3.14159265]

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 [10.0]

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

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

d [3.33333333]

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
`sbyte`	1 byte	−128	127
`short`	2 bytes	−32768	32767
`int`	4 bytes	−2147483648	2147483647
`long`	8 bytes	−9223372036854775808	9223372036854775807
`byte`	1 byte	0	255
`ushort`	2 bytes	0	65535
`uint`	4 bytes	0	4294967295
`ulong`	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
`float`	4 bytes	7	3.4×10^{38}
`double`	8 bytes	15	1.7×10^{308}
`decimal`	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.

5.2 Objecten

Object: groepje variabelen dat bij elkaar hoort
Een variabele is een geheugenplaats met een naam, die je kunt veranderen met een toekenningsopdracht. 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 plek op het scherm 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 Activity zijn variabelen nodig om de naam en het icoon in de titelbalk te bewaren, de status van de app, de rotatie, en de View die als contentview is ingesteld. Het is natuurlijk erg gemakkelijk om een Activity in zijn 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 2.2 gedaan: door het meegeven van een `Color`-object aan `SetBackgroundColor` blz. 16 werd de achtergrondkleur van een `TextView` 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 voor activities, views, colors, buttons en files objecten gebruikt, zonder dat de programmeur de opbouw van die objecten in detail kent. Die details worden (gelukkig) afgeschermd in de library-klasssen.
Van de meeste standaard-objecten (views, 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 verborgen 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 dan moet je natuurlijk wel weten hoe je zelfbedachte 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 opvragen (bijvoorbeeld de `Length` van een `string`-object) of veranderen (bijvoorbeeld de `TextSize` van een `TextView`-object), en welke methoden je ermee kunt aanroepen (bijvoorbeeld de methode `DrawLine` van een `Canvas`-object). Daarnaast kun je het object natuurlijk ook als geheel gebruiken: door het mee te geven aan een methode (bijvoorbeeld een `TextView`-object aan de methode `SetContentView`

van een `Activity`-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 opgvragen, 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 maar beschikbaar 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

```
canvas.DrawLine(10, 10, 30, 30, verf);
```

worden de variabelen van het `Canvas`-object `canvas` bekeken om te bepalen in welk window de lijn getekend moet worden. Maar het object `canvas` zelf verandert niet als gevolg van deze aanroep.
Andere methoden zullen (de variabelen van) het object blijvend veranderen. Bijvoorbeeld, door de methode-aanroep

```
scherm.SetBackgroundColor(Color.White);
```

wordt een variabele in het `scherm`-object veranderd die de achtergrondkleur bewaart.

Sommige methoden veranderen objecten
Een ander voorbeeld van objecten die blijvend kunnen veranderen zijn objecten van het type `Color`. Zo'n object kun je je voorstellen als de specificatie van een kleur: de hoeveelheid rood, groen en blauw die in de kleur aanwezig zijn. Een `Color`-object in het geheugen opgeslagen als drie getallen `R`, `G` en `B`. Ook is er een constructor-methode die de drie genoemde getallen als parameter krijgt. Zo kunnen we dus een `Color`-object maken:

```
Color c;
c = new Color(96, 255, 0);
```

Later in het programma kun je het `Color`-object gebruiken, bijvoorbeeld in de toekenningsopdracht

```
verf.Color = c;
```

Een ander voorbeeld van een object is een `Rect`-object, waarmee de positie van een rechthoek wordt vastgelegd. In een `Rect`-object zitten vier getallen, die de boven-, linker-, onder- en rechterkant voorstellen. Ook hier kun je een object aanmaken door aanroep van de constructormethode:

```
Rect r;
r = new Rect(100, 200, 150, 300);
```

Een `Rect`-object kun je gebruiken om een rechthoek te tekenen. Eerder hebben we dat gedaan door vier losse getallen mee te geven bij de aanroep van `DrawRect`:

```
canvas.DrawRect(100, 200, 150, 300, verf);
```

Maar aan een variant van de methode `DrawRect` kun je een compleet `Rect`-object meegeven:

```
canvas.DrawRect(r, verf);
```

5.3 Object-variabelen

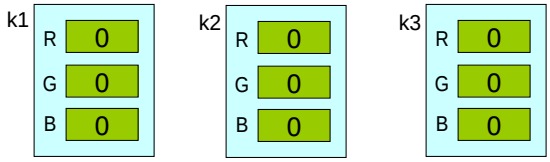
We gaan nu bekijken wat er precies in het geheugen gebeurt als je een variabele declareert met een object-type, en wat als je er waarden aan toekent met toekenningsopdrachten. Als demonstratie dient het programma CopyDemo. In listing 9 staat de programmatekst van de `View`-subklasse die in dit programma wordt gebruikt. De `Activity`-subklasse is niet zo interessant; het enige wat die doet is een `CopyDemoView`-object aanmaken en meegeven aan `SetContentView`. blz. 60
In figuur 13 is het programma in werking te zien. Heel erg spannend is de output niet, want het tekent alleen drie cirkels en drie rechthoeken. 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 `Color` gedeclareerd, en drie variabelen van het type `Rect`. Er worden `new` objecten gemaakt, er worden properties van veranderd, en daarna worden ze gebruikt om dingen 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 `Color` in de library als `struct` is gedefinieerd, terwijl het type `Rect` in de library als `class` is gedefinieerd.

Color-variabelen bevatten waardes
In het programma worden drie `Color`-variabelen gedeclareerd:

```
Color k1, k2, k3;
```

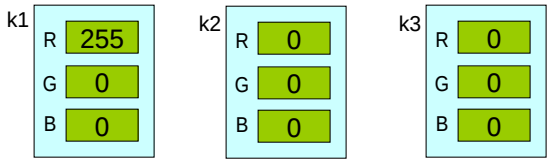
Er wordt daarmee ruimte in het geheugen gereserveerd voor drie objecten. Hoe zo'n object intern is opgebouwd kun je nooit helemaal zeker te weten komen, maar voor een `Color` is het aannemelijk dat daarin de drie kenmerken van een kleur worden opgeslagen: de hoeveelheid rood, groen en blauw. Alle variabelen 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 je nu dus als volgt voorstellen:



Door de toekenningsopdracht

```
k1 = new Color(255, 0, 0);
```

krijgt de variabele `k1` een nieuwe waarde. De situatie in het geheugen wordt daarmee:



We gebruiken deze kleur in de verf waarmee we een cirkel tekenen:

```
verf.Color = k1;
canvas.DrawCircle(100, 200, 50, verf);
```

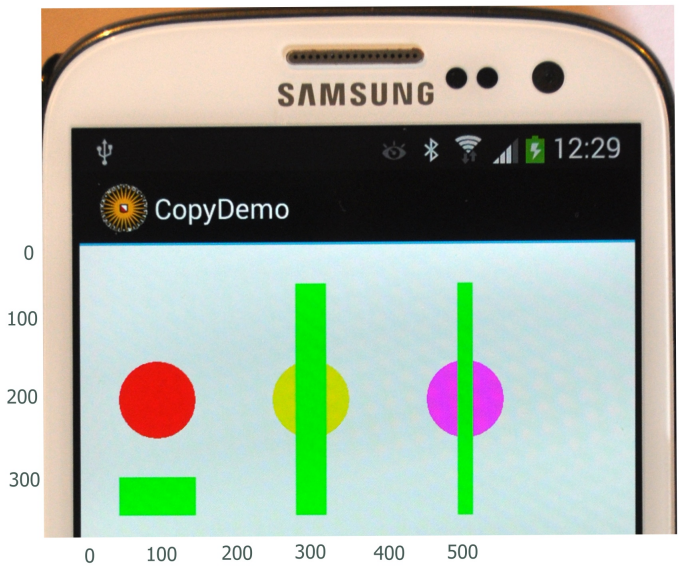
Hiermee verschijnt een rode cirkel op het scherm, want alleen de rood-component van de kleur heeft waarde 255.
Vervolgens worden in het programma twee toekenningsopdrachten gedaan aan `k2` en `k3`:

```
k2 = k1;
k3 = k1;
```

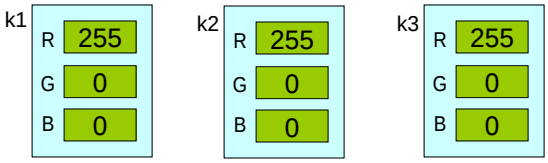
Die twee variabelen worden daarmee een kopie van de variabele `k1`. Nadat de toekenningen zijn gedaan hebben de drie objecten dus allemaal dezelfde waarde:

```
using Android.Views;      // vanwege View
using Android.Graphics;   // vanwege Color, Paint, Canvas
using Android.Content;    // vanwege Context

namespace CopyDemo
{
    public class CopyDemoView : View
    {
        public CopyDemoView(Context c) : base(c)
        {
            this.SetBackgroundColor(Color.White);
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);
            Paint verf = new Paint();

            Color k1, k2, k3;
            Rect  r1, r2, r3;

            // Maak een Color, en gebruik die om een cirkel te tekenen
            k1 = new Color(255, 0, 0);   // rood
            verf.Color = k1;
            canvas.DrawCircle(100, 200, 50, verf);

            // Maak twee kopieen van de Color
            k2 = k1;
            k3 = k1;
            // Verander de gekopieerde kleuren, en gebruik die om nog twee cirkels te tekenen
            k2.G = 255;   // rood+groen=geel
            verf.Color = k2;
            canvas.DrawCircle(300, 200, 50, verf);
            k3.B = 255;   // rood+blauw=roze
            verf.Color = k3;
            canvas.DrawCircle(500, 200, 50, verf);

            verf.Color = Color.Green;
            // Maak een Rect, en gebruik die om een blok te tekenen
            r1 = new Rect(50, 300, 150, 350);
            canvas.DrawRect(r1, verf);

            // Maak twee kopieen (?) van de Rect
            r2 = r1;
            r3 = r1;
            // Verander de gekopieerde Rect-objecten, en gebruik ze om nog twee blokken te tekenen
            r2.Left  = 280;
            r2.Right = 320;
            r2.Top   =  50;
            canvas.DrawRect(r2, verf);
            r3.Left  = 490;
            r3.Right = 510;
            canvas.DrawRect(r3, verf);
        }
    }
}
```

Listing 9: CopyDemo/CopyDemoView.cs

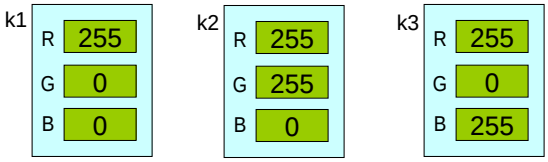


Figuur 13: Het programma CopyDemo in werking



Dat gaat veranderen als we de properties van de objecten `r2` en `r3` aanpassen:

```
k2.G = 255;
k3.B = 255;
```



De kleur `k2` stelt nu de kleur geel voor (de rood-component was al 255, en nu is de groen-component dat ook geworden; de mengkleur van rood plus groen is geel). De kleur `k3` stelt de kleur roze voor (rood plus blauw geeft roze).
Met deze nieuwe kleuren kunnen we cirkels op het scherm tekenen, en zoals verwacht krijgen we dan een gele en een roze cirkel te zien:

```
verf.Color = k2;
canvas.DrawCircle(300, 200, 50, verf);
verf.Color = k3;
canvas.DrawCircle(500, 200, 50, verf);
```

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

```
Rect r1, r2, r3;
```

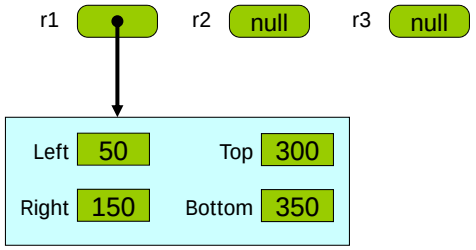
wordt niet ruimte gereserveerd voor de objecten zelf, maar voor *verwijzingen* naar de objecten. Direct na de declaratie wijzen deze verwijzings-variabelen nog nergens naar, en hebben ze een neutrale waarde: de speciaal voor dit doel bestaande waarde `null`.

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 toekenningsopdracht:

```
r1 = new Rect(50, 300, 150, 350);
```

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 vier waarden die we aan de constructormethode meegegeven, worden gebruikt om de vier eigenschappen van het object vast te leggen. De situatie in het geheugen is na de toekenningsopdracht dan als volgt:



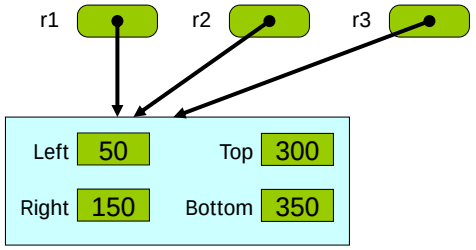
Met een aanroep van `DrawRect` kunnen we de rechthoek die door `r1` wordt beschreven op het scherm tekenen.

```
verf.Color = Color.Green;
canvas.DrawRect(r1, verf);
```

Als je de coordinaten napuzzelt, kun je zien dat de rechthoek onder de rode cirkel komt te staan. Verderop in het programma staan twee toekenningsopdrachten aan `r2` en `r3`:

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

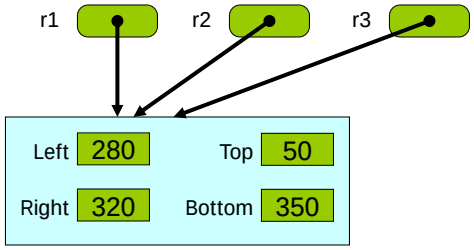
Het is hier dat er iets anders gebeurt dan in het geval van de `Color`-objecten. Nu wordt namelijk niet het complete object gekopieerd, maar alleen de verwijzing erheen. Het effect op het geheugen is dus als volgt:



We kunnen nu enkele properties van de `r2` veranderen:

```
r2.Left  = 280;
r2.Right = 320;
r2.Top   =  50;
canvas.DrawRect(r2, verf);
```

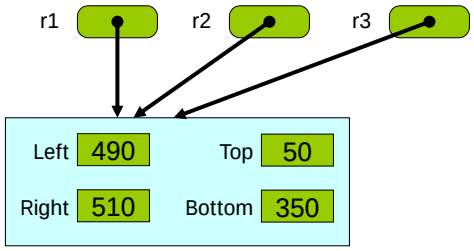
De horizontale plaatsing van deze rechthoek is anders (van 280 tot 320) dan die van de eerste rechthoek. Hij begint ook hoger (50), maar de onderkant is niet veranderd en dus nog steeds 350.



De tweede rechthoek is smaller dan de eerste, en wordt dwars over de tweede cirkel getekend; zie de scherm-afbeelding in figuur 13.
In het schema valt het op dat hiermee in feite ook het object `r1` is veranderd. Niet de variabele `r1`, want die bevat nog precies dezelfde pijl als voorheen, maar het object waar `r1` naartoe wijst is veranderd, want dit is hetzelfde object als dat waar `r2` naar wijst, en dat hebben we net aangepast. Het beeld op het scherm wijzigt echter niet: `r1` was al getekend voordat z'n object stiekem was veranderd.
Nu veranderen we de horizontal positie van `r3`, en tekenen ook deze op het scherm:

```
r3.Left  = 490;
r2.Right = 510;
canvas.DrawRect(r3, verf);
```

En hoewel we niets aan de `Top` van `r3` hebben veranderd, verschijnt ook deze rechthoek bovenaan het scherm, alsof de `Top` gelijk is aan 50! 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 naartoe wijzen.



Zou `Rect`, net als `Color`, niet een `class` maar een `struct` zijn geweest, dan zou de derde rechthoek *onder* de cirkel zijn verschenen, omdat zijn `Top` dan 300 was gebleven.

Voorbeelden van objectverwijzingstypen
In de libraries die met C# worden meegeleverd is 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 getallen: de hoeveelheid rood, groen en blauw, en de als 'alfa' bekend staande doorschijnendheid), `DateTime` (een groepje variabelen dat een datum en een tijd kan bevatten), `TimeSpan` (een groepje variabelen dat een tijds*duur* kan bevatten).
- typen van objecten met een wat complexere structuur, die een zeer natuurlijke eenheid vormen: `String` (een tekst bestaande uit nul of meer lettertekens), `Typeface` (een lettertype), `Bitmap` (een afbeelding).
- typen van objecten die nodig zijn om een interactieve interface te maken: `TextView` (tekstpaneeltje op het scherm), `Button` (een drukknop op het scherm), `SeekBar` (een schuifregelaar), `EditText` (een invulveld voor de gebruiker).

- typen van objecten die een bepaald kunstje kunnen uitvoeren, zoals `Canvas` (om een tekening te maken)
- typen van objecten waarmee files en internet-verbindingen gemanipuleerd kunnen worden: `File`, `URL`, `FileReader`, `FileWriter` en vele anderen.

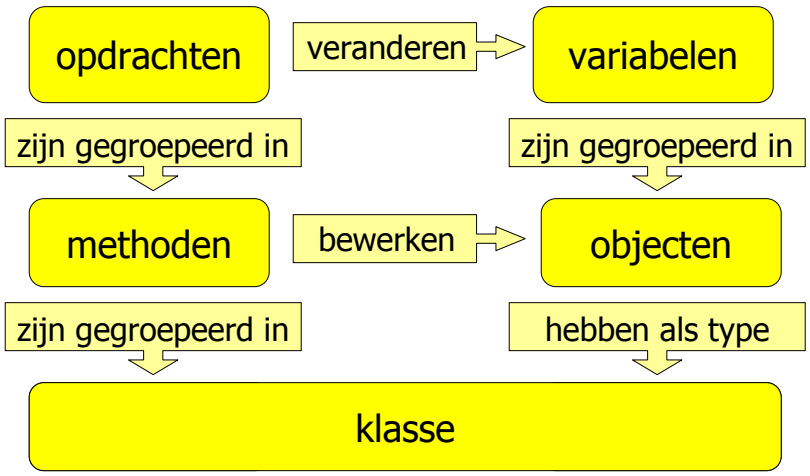
Van al deze typen kunnen variabelen worden gedeclareerd. In het geval van `struct` typen bevat de variabele dan het object zelf, in het geval van `class` typen is de variabele een verwijzing die kan wijzen naar het eigenlijke object.

Klasse: groepje methoden èn type van object

blz. 2 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 `DrawRect` bijvoorbeeld beschikbaar in de klasse `Canvas`.
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;  Rect r;
```

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



Figuur 14: De twee rollen van het begrip "klasse"

5.4 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 operanden 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 toekenningsopdracht 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: de decimale punt maakt dat dit vanhet type 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 wèl 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 `double`s 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, dan 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. Vooral voor de operator `+` wordt dat door library-auteurs vaak gedaan.

5.5 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
```

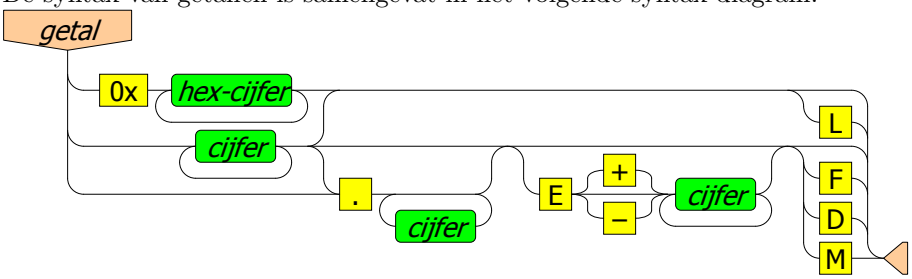
Voor hele grote, of hele kleine getallen kun je de 'wetenschappelijke notatie' gebruiken, bekend van de rekenmachine:

`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, en een `L` achter een `long`.
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.
De syntax van getallen is samengevat in het volgende syntax-diagram:



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:

```
"\"To be or not to be\", that's the question."
"gewone slash: / backslash: \\ "
```

Met behulp van de backslash kunnen nog een aantal andere bijzondere tekens worden aangeduid: een regelovergang door `\r\n`, een tabulatie door `\t` en het symbool met Unicode-nummer (hexadecimaal) 26a3 door `\u26a3`. Dat laatste is vooral handig om symbolen die niet op het toetsenbord voorkomen in een string te zetten.

5.6 Static properties

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 Android.Graphics;`
- bij het opvragen/wijzigen van een property: `naam.Length` of `scherm.TextSize`
- bij het aanroepen van een methode: `scherm.SetBackgroundColor(k);` of `canvas.DrawCircle(0,0,10,verf);`
- bij het opvragen van een standaardkleur: `Color.Yellow`

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.TextSize` is de lettergrootte van een `TextView`-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 tekstgrootte' 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 tekstgrootte van `TextView` bepalen, maar wel van een bepaalde `TextView` zoals `scherm`.

Hoofdstuk 6

Touch & go

In hoofdstuk 3 zagen we hoe je een button laat reageren als de gebruiker hem indrukt, door een eventhandler te koppelen aan het `Click`-event. In hoofdstuk 4 zagen we hoe je op een `View` zelf een tekening kunt maken door de `OnDraw`-methode opnieuw te definiëren. Leuker is het als de gebruiker direct kan ingrijpen op de tekening, door die op een bepaalde plaats aan te raken. Om dat te doen gaan we een eventhandler schrijven voor het `Touch`-event.
Behalve een aanraakscherm zitten er nog veel meer sensoren in een smartphone. In sectie 6.3 gebruiken we de magnetische-veldsensor om een kompas te maken.
In de voorbeeldprogramma's gebruiken we ook twee nieuwe opdracht-vormen: de if-opdracht en de foreach-opdracht. Ook gebruiken we enkele nieuwe standaard-klassen, zoals `RectF` en `List`.

6.1 Touch

Voorbeeld: puntenklikker
In de volgende sectie bekijken we een puntenklikker, waarmee de gebruiker op allerlei plaatsen op het scherm een markering kan neerzetten. In figuur 15 is dit programma in werking te zien. Als voorbereiding daarop schrijven we in deze sectie een programma waarmee de gebruiker één punt kan markeren.
Op het punt waar de gebruiker het scherm aanraakt komt ons Sol-logo te staan. Als de gebruiker daarna met zijn vinger over het scherm beweegt, volgt het logo de beweging. Wanneer de gebruiker het scherm loslaat, blijft het logo op die plek staan. Raakt de gebruiker het scherm daarna op een andere plek aan, dan verplaatst het logo naar die plek.
Zoals gewoonlijk bestaat het programma uit twee delen: een subklasse van `Activity` en een subklasse van `View`. De subklasse van `Activity` laten we hier niet zien, omdat hij vrijwel hetzelfde is als in eerdere programma's: er wordt een object aangemaakt van de `View`-subklasse, en die wordt tot `ContentView` van de app gemaakt. De subklasse `PuntenKlikkerView0` van `View` is wel interessant; deze staat in listing 10. blz. 71

Het Touch-event van een View
De klasse `PuntenKlikkerView1` bestaat onder andere uit een constructormethode. Contructormethodes heten altijd hetzelfde als de klasse, dus in dit geval is dat `PuntenKlikkerView1`. Construtormethodes worden automatisch aangeroepen als je een `new` object van de klasse maakt; in dit geval gebeurt dat in de `Activity`-subklasse `PuntenKlikkerApp`. Daarnaast is er een `override`, oftewel een herdefinitie, van de methode `OnDraw`, die moet tekenen wat er in de `PuntenKlikkerView1` te zien is.
Het belangrijkste wat er in de constructormethode gebeurt, is het koppelen van een eventhandler aan het `Touch`-event:

```
this.Touch += RaakAan;
```

Hiermee wordt geregeld dat de `RaakAan`-methode zal worden aangeroepen op het moment dat er een `Touch`-event optreedt, dat wil zeggen de gebruiker het scherm aanraakt, eroverheen beweegt, of het weer loslaat.
Voor de punt van `Touch` staat het object waarvan we de aanrakingen willen weten. Hier is dat `this`, het object dat door de constructormethode `PuntenKlikkerView1` wordt geconstrueerd. Dat is dus: de hele tekening.

Figuur 15: De PuntenKlikker in werking

De Touch-eventhandler

De methode `RaakAan` die we als eventhandler gebruiken moet er dan natuurlijk wel zijn. We definiëren dus ook deze methode. Zoals gebruikelijk bij eventhandlers heeft deze methode twee verplichte parameters. De tweede parameter is ditmaal niet van type `EventArgs` (zoals dat in eerdere eventhandlers het geval was) maar een subklasse daarvan: `TouchEventArgs`. Dit is een object waarin nadere details over het soort event dat is opgetreden beschikbaar zijn.
Bij het `Click`-event van een `Button` was het voldoende om te weten *dat* de button is ingedrukt. We hebben daar de `EventArgs`-parameter dan ook niet gebruikt. Maar nu willen we ook weten *waar* de view is aangeraakt, en nu komt die parameter dus goed van pas. De x- en y-coöordinaat van de aangeraakte plaats kunnen worden opgevraagd door aanroep van de methoden `GetX` en `GetY`, waarbij het object `Event` onder handen wordt genomen, dat op zijn beurt een property is van de `TouchEventArgs`-parameter van de eventhandler. We schrijven daarom:

```
public void RaakAan(object o, TouchEventArgs tea)
{
    float x = tea.Event.GetX();
    float y = tea.Event.GetY();
```

Het is opmerkelijk dat het resultaat van deze methoden niet een `int` is, maar een `float`. Posities op het scherm zijn immers aangeraakte pixels, en die zijn genummerd met gehele getallen. Dat er toch voor `float` is gekozen door de auteurs van de library, is omdat in sommige gevallen het scherm vergroot kan worden weergegeven, en dan is het wèl mogelijk om een halve pixel aan te wijzen. En omdat 7 cijfers achter de komma meer dan genoeg is, is er gekozen voor `float` en niet voor `double`.

De toestand van de puntenklikker

Het liefst zou je misschien het aangeraakte punt meteen in de eventhandler willen markeren, door daar een vierkantje, cirkeltje of een bitmap te tekenen. Maar dat kan niet: om te kunnen tekenen heb je een `Canvas` nodig, en die is in `RaakAan` niet beschikbaar. Zo'n `Canvas` is er wel in de methode `OnDraw`. In feite is dat de *enige* plek waar er een `Canvas` beschikbaar is. Tekenen moet dan ook altijd vanuit de methode `OnDraw` gebeuren.

```
using System.Collections.Generic;  // vanwege List (in tweede versie)
using Android.Views;      // vanwege View, TouchEventArgs
using Android.Graphics;   // vanwege Color, Paint, Canvas
using Android.Content;    // vanwege Context

namespace PuntenKlikker
{
    class PuntenKlikkerView1 : View
    {
        Bitmap Plaatje;

        public PuntenKlikkerView1(Context c) : base(c)
        {
            this.SetBackgroundColor(Color.White);
            this.Touch += RaakAan;
            Plaatje = BitmapFactory.DecodeResource(c.Resources, Resource.Drawable.Icon);
        }

        PointF punt;

        public void RaakAan(object o, TouchEventArgs tea)
        {
            float x = tea.Event.GetX();
            float y = tea.Event.GetY();
            this.punt = new PointF(x, y);
            this.Invalidate();
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);
            Paint verf = new Paint();

            int rx = Plaatje.Width  / 2;
            int ry = Plaatje.Height / 2;

            if (punt!= null)
                canvas.DrawBitmap(Plaatje, punt.X - rx, punt.Y - ry, verf);
        }
    }
}
```

Listing 10: PuntenKlikker/PuntenKlikkerView1.cs

Maar hoe kan `OnDraw` weten *waar* er getekend moet worden? Het aangeraakte punt is immers alleen beschikbaar als parameter in `RaakAan`, en daar kan `OnDraw` juist weer niet bij.
De oplossing van dit dilemma is dat we een variabele declareren in de klasse, dus buiten de twee methoden. We declareren daarom:

```
PointF punt;
```

als variabele waarin de coördinaten van het aangeraakte punt kunnen worden opgeslagen. Het type van deze variabele is `PointF`. Dat is een klasse die objecten beschrijft met twee variabelen: `X` en `Y`. In het geval van de klasse `PointF` zijn deze twee variabelen van het type `float`. Dat is wat we hier nodig hebben, daarom gebruiken we niet de klasse `Point`, waarin de coördinaten `int` zijn.
De toekenning aan deze variabele gebeurt in de methode `RaakAan`. We maken een nieuw `PointF`-object, waarin de coördinaten van het aangeraakte punt worden opgeslagen:

```
this.punt = new PointF(x, y);
```

De variabele wordt bekeken in de methode `OnDraw`, die dan nog kan kiezen of hij er een cirkel tekent met

```
canvas.DrawCircle(punt.X, punt.Y, 40, verf);
```

of een bitmap, zoals dat in de uiteindelijke versie van het programma gebeurt.
De waarde van variabele `punt` kan worden beschouwd als de *toestand* van het programma. Deze toestand wijzigt als gevolg van akties van de gebruiker, en de toestand wordt afgebeeld door `OnDraw`.

Invalidate: forceer tekenen van de View

De eventhandler `RaakAan` kan de toestand dan wel wijzigen, maar daarmee is de nieuwe toestand nog niet automatisch op het scherm afgebeeld. Daarvoor moet eerst nog `OnDraw` aangeroeen worden. Maar `RaakAan` kan niet zelfstandig `OnDraw` aanroepen. Die heeft namelijk een `Canvas`-parameter nodig, en die hebben we nou juist niet in `RaakAan`.
In plaats daarvan roept `RaakAan` de methode `Invalidate` aan. Dit is een methode zonder parameters, vandaar het lege paar haakjes in de aanroep

```
this.Invalidate();
```

Deze methode gebruik je op het moment dat het object dat je hem onder handen geeft (in dit geval is dat `this`, dus de complete `PuntenKlikkerView1`) niet meer up-to-date is met de gewijzigde toestand. Daarop reageert de `View` (want in die klasse is de methode `Invalidate` gedefinieerd) door de methode `OnDraw` aan te roepen. Blijkbaar heeft zo'n `View` ergens een `Canvas` beschikbaar die hij als parameter aan `OnDraw` kan meegeven.
Door de aanroep van `Invalidate` als laatste opdracht in eventhandler `RaakAan` zal dus indirect toch de methode `OnDraw` worden aangeroepen, en is de gewijzigde toestand weer zichtbaar op het scherm.

Een Bitmap uit de Resource

In plaats van een cirkeltje tekent het programma een bitmap op het aangeraakte punt. Het tekenen van een bitmap `b` gebeurt door het aanroepen van `canvas.DrawBitmap(b,x,y,verf);`, maar hoe kom je aan zo'n bitmap? Uit de fabriek!
Er is een klasse `BitmapFactory`, met daarin een statische methode waarmee je bitmaps kunt maken. Maar fabrieken hebben grondstoffen nodig, oftewel *resources*. De resources waar het hier om gaat zijn grafische bestanden waarin het plaatje van de bitmap is vastgelegd. Je kunt die met een tekenprogramma maken, en opslaan als bijvoorbeeld een JPG- of PNG-bestand. De grafische bestanden worden meeverpakt met de objectcode van het programma. Standaard wordt in ieder geval het icoon van het programma meeverpakt.
De statische methode `DecodeResource` uit de klasse `BitmapFactory` kan het grafische bestand decoderen. Als parameter moet je daartoe de resources van je programma meegeven. Die zijn beschikbaar als eigenschap van een `Context`-object. Dat is precies het object dat beschikbaar is in de constructor van een subklasse van `View`.
Als tweede parameter wil `DecodeResource` weten *welke* resource je wilt decoderen. Dat is simpelweg een `int` met een uniek nummer voor elke resource. Maar welk nummer heeft, bijvoorbeeld, het icoon?
Om daar achter te komen is er in elk programma een extra klasse `Resource`, die wordt geschreven

door de ontwikkelomgeving. De broncode hiervan zit verborgen in de map `Resources` van je project, en heet `Resource.Designer.cs`. Omdat de klasse in dezelfde namespace staat als onze eigen klasses, kun je hem gewoon gebruiken. In de klasse `Resource` zit een statische variabele `Resource.Drawable.Icon` waarin het unieke nummer van de icoon-resource zit. Dat nummer is in dit geval 2130837504, maar het is natuurlijk makkelijker om gewoon `Resource.Drawable.Icon` te schrijven.
Het aanmaken van de bitmap gaat al met al als volgt:

```
Plaatje = BitmapFactory.DecodeResource(c.Resources, Resource.Drawable.Icon);
```

Dit gebeurt in de constructormethode, om twee redenen:

- dit is de plaats waar we de benodigde `Context c` beschikbaar hebben
- het decoderen van het plaatje hoeft maar eenmaal te gebeuren, daarna kan het steeds opnieuw worden gebruikt als het getekend wordt.

Omdat de toekenning plaatsvindt in de constructor, en de variabele nodig is in een andere methode (`OnDraw`), is de variabele `Plaatje` in de klasse gedeclareerd. In feite maakt hij daarmee, samen met het aangeraakte punt, deel uit van de toestand van het programma.

Gecentreerd tekenen
Om het plaatje mooi gecentreerd op het aangeraakte punt te tekenen, geven we bij de aanroep van `DrawBitmap` een punt op dat ietsje linksboven het aangeraakte punt ligt. Het referentiepunt bij het tekenen van een bitmap is namelijk de linkerbovenhoek. Het verschil met het aangeraakte punt is de halve breedte, respectievelijk hoogte van het plaatje. Dat zijn eigenschappen die we van het plaatje kunnen opvragen:

```
int rx = Plaatje.Width  / 2;
int ry = Plaatje.Height / 2;
canvas.DrawBitmap(Plaatje, punt.X - rx, punt.Y - ry, verf);
```

De if-opdracht
Een laatste probleem is dat we de aanroep van `DrawBitmap` zoals die hierboven staat niet zomaar kunnen doen. De variabele `punt` krijgt zijn waarde immers pas in `RaakAan`. Bij de eerste aanroep van `OnDraw` (meteen aan het begin van het programma) heeft de gebruiker het scherm nog niet aangeraakt, en is de waarde van `punt` dus nog gelijk aan `null`: een verwijzing die nog nergens naar verwijst. Van dat soort *null pointers* kun je geen eigenschappen, zoals `X` en `Y`, opvragen: de variabele wijst immers nog niet naar een object.
We mogen de methode `DrawBitmap` dus alleen maar aanroepen als `punt` *niet* de waarde `null` heeft. Om dit mogelijk te maken is er in C# een aparte opdracht-vorm: de *if-opdracht*. De syntax van een if-opdracht is: het woord `if`, dan tussen haakjes een voorwaarde, en dan een willekeurige andere opdracht. De semantiek is dat de opdracht alleen wordt uitgevoerd als de voorwaarde geldig is.
In dit geval schrijven we:

```
if (punt !=  null)
    canvas.DrawBitmap(Plaatje, punt.X - rx, punt.Y - ry, verf);
```

De voorwaarde is `punt!=null`, waarbij je de operator `!=` moet lezen als 'is niet gelijk aan'. Het tekenen gebeurt dus alleen maar als de variabele `punt` inderdaad naar een object verwijst.

6.2 Lijsten

Voorbeeld: puntenklikker
We gaan het programma nu uitbreiden, zodat niet alleen op de laatst aangeraakte plaats een icoon getekend wordt, maar op *alle* plaatsen die gedurende de looptijd van het programma aangeraakt worden. In figuur 15 was dit programma al in werking te zien. Als extra tekenen we ook nog een blauwe rand die precies rond de aangeraakte punten komt te liggen.
We schrijven een `View`-subklasse `PuntenKlikker`, als aanpassing van `PuntenKlikker1` uit de vorige sectie. De `using`-regels en de constructormethode zijn precies hetzelfde als in het vorige programma. In listing 11 tonen we hier alleen het deel dat anders is dan in de vorige versie: de definitie van de methodes `RaakAan` en `OnDraw`. Verder is er natuurlijk ook weer een `Activity`-subklasse nodig, waarin zo'n `PuntenKlikker`-object wordt aangemaakt. blz. 76

De toestand van de puntenklikker

In het vorige voorbeeld noemden we de *toestand* van het programma: alle variabelen die nodig zijn zodat `OnDraw` zijn werk kan doen. In dat voorbeeld bestond de toestand uit een `Bitmap` waarin de te tekenen icoon stond opgeslagen, en een `PointF` met de coördinaten waar dat moet gebeuren.
In het nieuwe voorbeeld is de toestand ingewikkelder. De `Bitmap` is nog steeds nodig, maar we hebben niet meer genoeg aan één `PointF`-variabele. In `OnDraw` moeten immers *alle* punten die ooit zijn aangeraakt worden getekend. Bedenk dat `OnDraw` altijd met een schone lei begint te tekenen; het is niet mogelijk om er iets *bij* te tekenen bij wat er al eerder op het scherm stond.
Wat we dus nodig hebben is een object waarin niet één punt, maar een heleboel punten kunnen worden opgeslagen. Zo'n object bestaat: het type dat we hiervoor gaan gebruiken heet `List`.

List: een object met vele deel-objecten

Een `List` is een voorbeeld van een *generiek type*. Met *generiek* wordt bedoeld dat er verschillende soorten lijsten mogelijk zijn: lijsten van getallen, lijsten van teksten, lijsten van kleuren, en jawel: lijsten van punten. Zo'n lijst van punten is wat we in dit programma nodig hebben.
Een generiek type duid je aan met zijn naam, en daarachter tussen punthaken het type van de elementen van de lijst. In dit geval is dat dus `List<PointF>`. Dat type gebruiken we in een declaratie, dus

```
List<PointF> punten;
```

Zonder toekenningsopdracht is deze variabele nog `null`, dus om daadwerkelijk een lijst te maken moeten we ook nog deze opdracht schrijven:

```
punten = new List<PointF>();
```

Deze toekenningsopdracht kan in de body van constructormethode komen te staan. Maar eigenlijk is het wel zo gemakkelijk om de toekenning meteen al te doen bij de declaratie, dus:

```
List<PointF> punten = new List<PointF>();
```

Let op de plaatsing van de verschillende soorten haakjes: de punthaken (omdat het een generiek type betreft) èn de ronde haakjes (omdat het een aanroep van een constructormethode betreft). Qua structuur is deze declaratie niet wezenlijk anders dan die van

```
Color geel = new Color(255,255,0);
```

En net zoals daar het type `Color` tweemaal vermeld staat, is hier het (generieke) type `List<PointF>` tweemaal vermeld.
De declaratie (en tevens toekenning van het object) van de variabele `punten` komt direct in de klasse-body te staan, zodat beide methoden de variabele kunnen gebruiken: `RaakAan` om er een element aan toe te voegen, en `OnDraw` om ze allemaal te kunnen tekenen. Samen met de `Bitmap` vormt deze variabele de toestand van het programma.

Add: element toevoegen aan een lijst

Voor het toevoegen van een element aan een lijst kent deze de methode `Add`. De parameter van `Add` moet een element zijn dat het type heeft zoals dat bij de declaratie van de lijst tussen punthaken werd opgeschreven. In dit geval is dat `PointF`, en daarom schrijven we in de methode `RaakAan` de opdrachten:

```
float x = tea.Event.GetX();
float y = tea.Event.GetY();
PointF punt = new PointF(x, y);
punten.Add(punt);
```

Het geheel wordt gevolgd door een aanroep van `Invalidate`, maar dat was in het vorige voorbeeld ook al zo.
Zouden we het hierbij laten, dan wordt er bij *elk* `Touch`-event een punt aan de lijst toegevoegd. Dus niet alleen bij het aanraken van het scherm, maar ook met het bewegen van je vinger over het scherm. Dat geeft het op zich wel aardige effect dat je met je vinger een heel spoor van iconen kan trekken, maar dat is misschien toch te veel van het goede.
We zetten de hele zaak daarom in de body van een `if`-opdracht, die er voor zorgt dat die alleen maar wordt uitgevoerd bij het goede soort event. De header van de `if`-opdracht is:

```
if (tea.Event.Action == MotionEventActions.Down)
```

Deze controleert of het een event betreft dat correspondeert met het neerzetten van je vinger op het scherm (`Down`), en dus niet met het bewegen (`Move`) of het loslaten (`Up`).
Het dubbele `==` teken moet gelezen worden als 'is op dit moment gelijk aan', en kan gebruikt worden in dit soort voorwaarden. Dat is wat anders dan het toekenningsteken `=`, dat gelezen moet worden als 'wordt vanaf nu gelijk aan'. Dus kortweg: `==` betekent *is*, en `=` betekent *wordt*.
De vijf opdrachten in de body van deze `if`-opdracht zijn met accolades samengevoegd tot één geheel. Ze worden dus alle vijf uitgevoerd als de voorwaarde geldig is, en alle vijf overgeslagen als dat niet zo is.

De foreach-opdracht
In de methode `OnDraw` moet op alle punten van de lijst een icoon op het scherm getekend worden. Hoe je één icoon tekent hebben we al gezien in het vorige programma: met een aanroep van `DrawBitmap`. In dit geval moet dat voor elk punt in de puntenlijst gebeuren. Dat kun je eigenlijk letterlijk zo opschrijven in C#:

```
foreach (PointF p in punten)
    canvas.DrawBitmap(Plaatje, p.X - rx, p.Y - ry, verf);
```

Hierbij gebruiken we weer een nieuwe opdrachtvorm: de `foreach`-opdracht. De syntax daarvan is als volgt:

- het woord `foreach`, met daarachter tussen haakjes:
  - een declaratie van een variabele van het elementtype van de lijst, in dit geval `PointF p`
  - het woord `in`
  - een lijst-waarde, in dit geval de variabele `punten`
- een opdracht, waarin de gedeclaraarde variabele `p` gebruikt mag worden

De semantiek van deze opdrachtvorm is dat de opdracht in de body steeds opnieuw wordt uitgevoerd, waarbij de variabele dan achtereenvolgens alle elementen van de lijst als waarde heeft.

RectF: een rechthoek
Nu hoeven alleen nog het blauwe vierkant rond de punten te tekenen. Dat kan mooi worden opgelost met een variabele van het type `RectF`. Zo'n object stelt een rechthoek voor. Er is een handige methode `Union` waarmee de rechthoek een punt kan opslurpen: de rechthoek wordt indien nodig groter gemaakt om het punt te absorberen.
We roepen die methode aan voor alle punten die we in de lijst tegenkomen, dus in de body van `foreach`. Een uitzondering is het eerste punt: dan is er nog geen rechthoek, en kunnen we dus ook niet `Union` gebruiken; in dit geval maken we een nieuw `RectF` aan, die alleen het eerste punt omvat.
Na afloop van de `foreach`-opdracht kunnen we dan de rechthoek tekenen door het hele `RectF`-object mee te geven aan `DrawRect`. Om te zorgen dat de rechthoek niet door de middelpunten van de cirkel loopt, maar er netjes omheen, maken we de rechthoek nog ietsje groter door de aanroep van `Inset`.
De rechthoek kan natuurlijk niet getekend worden als er nog helemaal geen punten zijn (want waar zou die dan getekend moeten worden?). In die situatie is de body van `foreach` nul keren uitgevoerd, en heeft de rechthoek-variabele nog de waarde `null`. Een `if`-opdracht zorgt ervoor dat er alleen wordt getekend als de rechthoek niet gelijk aan `null` is.

6.3 Sensors

Voorbeeld: Kompas
Behalve een aanraakscherm zitten er nog veel meer sensoren in een smartphone. In deze sectie gebruiken we de magnetische-veldsensor om een kompas te maken.
In figuur 16 is het programma in werking te zien. De listing van de `View`-subklasse staat in listing 12. Als kompasroos gebruiken we het Sol-logo. Houd je het kompas plat en richt je je naar blz. 78 het noorden dan staat het rood/witte wapen van Utrecht rechtop. Het puntje van het wapen wijst dus altijd naar het zuiden. Boven de kompasroos staat ook de draaiingshoek nog als getal.

Een Bitmap aan de Resource toevoegen
We hebben het Sol-logo al eerder gebruikt, als icoon van al onze programma's. En net als in sectie 6.1 zouden we die bitmap kunnen gebruiken om de kompasroos te tekenen. Zoals we hieronder

```
        List<PointF> punten = new List<PointF>();

        public void RaakAan(object o, TouchEventArgs tea)
        {
            if (tea.Event.Action == MotionEventActions.Down)
            {
                float x = tea.Event.GetX();
                float y = tea.Event.GetY();
                PointF punt = new PointF(x, y);
                punten.Add(punt);
                this.Invalidate();
            }
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);
            Paint verf = new Paint();
            RectF grens = null;

            int rx = Plaatje.Width / 2;
            int ry = Plaatje.Height / 2;

            foreach (PointF p in punten)
            {
                canvas.DrawBitmap(Plaatje, p.X - rx, p.Y - ry, verf);
                if (grens == null)
                     grens = new RectF(p.X, p.Y, p.X, p.Y);
                else grens.Union(p.X, p.Y);
            }

            if (grens != null)
            {
                grens.Inset(-rx, -ry);
                verf.StrokeWidth = 3;
                verf.SetStyle(Paint.Style.Stroke);
                verf.Color = Color.Blue;
                canvas.DrawRect(grens, verf);
            }
        }
    }
}
```

Listing 11: PuntenKlikker/PuntenKlikkerView.cs, deel 2 van 2

Figuur 16: Het Kompas in werking

zullen bespreken kun je een bitmap geschaald tekenen, en op die manier kan het icoon schermvullend getekend worden. Dat wordt dan niet zo mooi, want het scherm is veel groter dan de 72 pixels van het icoon, en uitvergroot worden de losse pixels dan duidelijk zichtbaar.
We gebruiken daarom een andere bitmap, waar weliswaar ook het Sol-logo in staat, maar dan in hogere resolutie. Het is een bitmap van 1024×1024 pixels. Deze bitmap kun je toevoegen aan de resource door in de ontwikkelomgeving met rechts te klikken op de map `Drawable` in de map `Resource` van het project. Kies menu-item 'existing item' om een bestaand bestand aan te wijzen, of 'new item' om ter plaatse een nieuwe bitmap te ontwerpen.
Het grafische bestand (in bijvoorbeeld png- of jpg-formaat) wordt zichtbaar in deze map, en het wordt inderdaad ook fysiek gekopieerd naar de overeenkomstige map in de file-structuur. Zelf een bestand neerzetten in de file-structuur is niet genoeg: het moet echt op bovenbeschreven manier. Dan past de ontwikkelomgeving namelijk ook het bestand `Resource.Designer.cs` aan met een extra constante-declaratie, en kunnen we de bitmap in het programma beschikbaar krijgen met

```
Bitmap Plaatje;
Plaatje = BitmapFactory.DecodeResource(context.Resources, Resource.Drawable.UU1024);
```

De declaratie staat bovenin de klasse, zodat de variabele ook in methode `OnDraw` gebruikt kan worden. De toekenningsopdracht staat in de constructormethode, waar de benodigde `context` aanwezig is. De naam van het bitmap-bestand was `UU1024.png`, en daarom is door de ontwikkelomgeving de constante `Resource.Drawable.UU1024` aangemaakt.

Een Bitmap geschaald tekenen
Eerder gebruikten we de methode `DrawBitmp` in een vorm waarin de bitmap en de positie waar deze getekend moest worden als parameter werden meegegeven:

```
canvas.DrawBitmap(this.Plaatje, x, y, verf);
```

We willen het plaatje nu echter schermvullend in beeld krijgen. Niet elk beeldscherm is precies 1024 pixels groot, dus het plaatje moet in de meeste gevallen geschaald worden. Dit is mogelijk

```
using System;             // vanwege Math
using Android.Views;      // vanwege View
using Android.Graphics;   // vanwege Paint, Canvas
using Android.Content;    // vanwege Context
using Android.Hardware;   // vanwege SensorManager

namespace Kompas
{
    class KompasView0 : View, ISensorEventListener
    {
        Bitmap Plaatje;
        float Hoek;
        float Schaal;

        public KompasView0(Context context) : base(context)
        {
            this.SetBackgroundColor(Color.White);

            BitmapFactory.Options opt = new BitmapFactory.Options();
            opt.InScaled = false;
            Plaatje = BitmapFactory.DecodeResource(context.Resources, Resource.Drawable.UU1024, opt)

            SensorManager sm = (SensorManager)context.GetSystemService(Context.SensorService);
            sm.RegisterListener(this, sm.GetDefaultSensor(SensorType.Orientation), SensorDelay.Ui);
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);

            Schaal = Math.Min( ((float)this.Width)  / this.Plaatje.Width
                             , ((float)this.Height) / this.Plaatje.Height );

            Paint verf = new Paint();
            verf.TextSize = 30;
            canvas.DrawText(Hoek.ToString(), 100, 20, verf);
            canvas.DrawText(Schaal.ToString(), 100, 50, verf);

            Matrix mat = new Matrix();
            mat.PostTranslate(-this.Plaatje.Width / 2, -this.Plaatje.Height / 2);
            mat.PostScale(this.Schaal, this.Schaal);
            mat.PostRotate(-this.Hoek);
            mat.PostTranslate(this.Width / 2, this.Height / 2);
            canvas.DrawBitmap(this.Plaatje, mat, verf);
        }

        public void OnSensorChanged(SensorEvent e)
        {
            this.Hoek = e.Values[0];
            this.Invalidate();
        }
        public void OnAccuracyChanged(Sensor s, SensorStatus accuracy)
        {
        }
    }
}
```

Listing 12: Kompas/KompasView0.cs

met een alternatieve versie van `DrawBitmap`. Deze krijgt in plaats van de twee getallen voor de positie een `Matrix`-object als parameter mee.
Een matrix is een manier om aan te geven hoe een tekening geschaald, geroteerd, gespiegeld of verplaatst moet worden. Met één zo'n matrix kun je al dit soort opties tegelijk meegeven, en dat is natuurlijk gemakkelijker dan wanneer `DrawBitmap` een ellenlange lijst parameters zou hebben. Wel moeten we nu een `Matrix`-object aanmaken, en daarin vastleggen wat de weergave-opties zijn. In dit geval is dat schaling met een zekere schaalfactor `Schaal`:

```
Matrix mat = new Matrix();
mat.PostScale(Schaal, Schaal);
canvas.DrawBitmap(this.Plaatje, mat, verf);
```

De schaalfactor wordt tweemaal meegegeven aan `PostScale`, omdat de schaalfactor in de lengte- en breedterichting in principe verschillend mag zijn (om een plaatje uit te rekken).
De schaalfactor wordt bepaald uit de verhouding van de afmetingen van het plaatje (`this.Plaatje`) en de hele view (`this`). We berekenen de verhouding apart voor de breedte en de hoogte. Door hiervan de kleinste te nemen (met de wiskundige hulpfunctie `Math.Min`), zorgen we ervoor dat het plaatje altijd helemaal in beeld is.

```
Schaal = Math.Min( ((float)this.Width)  / this.Plaatje.Width
                 , ((float)this.Height) / this.Plaatje.Height );
```

Probeer zelf te bedenken wat er gebeurt als je hier `Math.Max` zou gebruiken. En waarom is de *cast*, dat is het vermelden van `(float)`, hier noodzakelijk?

Pixel density
Moderne smartphones hebben een steeds hogere *pixel density*: ze hebben een hogere resolutie dan eerdere modellen, terwijl het scherm even groot blijft. De pixels zijn dus kleiner dan op eerdere devices. Dit heeft tot gevolg dat een bitmap op een modern scherm er kleiner uit zou zien dan op een device met een lagere pixel density.
Android probeert hiervoor te compenseren. De bitmap factory maakt het plaatje automatisch groter als het gedecodeerd wordt op een scherm met hoge pixel density. Een bitmap die oorspronkelijk 72 pixels breed is, zal op een 720p scherm daarom een `Width` van 144 pixels hebben. In het geheugen neemt de bitmap dan wel vier keer zo veel ruimte in (twee keer zo breed èn twee keer zo hoog).
Voor het tonen van icoontjes is dat nog wel een leuk idee, maar bij het laden van grote bitmaps, die we sowieso zelf gaan schalen zodat ze precies schermvullend getekend worden, is het zinloos en zonde van het geheugen. Gelukkig kun je het compensatie-gedrag van hoge-densitydevices uitschakelen. Hiervoor moet je een extra parameter meegeven aan `DecodeResource`. Omdat er nog wel meer opties mogelijk zijn, en om te voorkomen dat `DecodeResource` daardoor erg veel parameters nodig zou hebben, worden de opties samen ondergebracht in een speciaal object, dat als geheel wordt meegegeven. We moeten daarom eerst zo'n object aanmaken, de gewenste optie instellen, en het object meegeven aan `DecodeResource`:

```
BitmapFactory.Options opt = new BitmapFactory.Options();
opt.InScaled = false;
Plaatje = BitmapFactory.DecodeResource(context.Resources, Resource.Drawable.UU1024, opt);
```

De gekozen optie is hier om `InScaled` de waarde `false` te geven. Met andere opties kun je de bitmap zelfs in een lagere resolutie laden: dit is handig als je een *thumbnail* van een foto wilt tonen.

Sensors en Listeners
Een Android-device bevat meestal vele sensoren: je kunt de kompasrichting detecteren, maar ook de temperatuur, lichtsterkte, enz. Alleen bij de goedkoopste modellen wordt er op sensoren bezuinigd. Wil je één of meer sensoren gebruiken, dan heb je een `SensorManager` nodig. We schrijven daarom:

```
SensorManager sm;
sm = (SensorManager)context.GetSystemService(Context.SensorService);
```

Omdat hierbij de `Context` nodig is, staat deze opdracht in de constructormethode.
Je kunt aan zo'n `SensorManager` laten weten dat je in de waarde van een bepaalde sensor geïnteresseerd bent. Omdat we een kompas aan het maken zijn, willen we hier de sensor voor het magnetisch veld aanspreken, oftewel `SensorType.Orientation`. Dat gebeurt als volgt:

```
sm.RegisterListener(this, sm.GetDefaultSensor(SensorType.Orientation), SensorDelay.Ui);
```

Hiermee registreren we een *listener*, dat is: een object dat geïnformeerd wil worden als er iets aan de sensor verandert. De drie parameters van `RegisterListener` hebben de volgende betekenis:

1. het object dat geïnformeerd moet worden: hier is dat `this`, zodat de hele `View` geïnformeerd wordt;
2. het soort sensor waar het om gaat: hier is dat `Orientation`;
3. de snelheid waarin we updates willen krijgen: hier is dat `Ui`, een snelheid die geschikt is in een userinterface (het alternatief is `Game`, voor extra snelle updates).

Interface: belofte voor methodes

De sensormanager zal de geregistreerde *listeners* informeren over veranderingen die de sensor detecteert. Dat informeren gebeurt doordat de sensormanager een afgesproken methode aanroept: `OnSensorChanged`. Die methode moet er dan natuurlijk wel zijn. Omdat we het object `this` hebben geregistreerd als listener, en dit object een `KompasView` is, moet in de klasse `KompasView` de methode `OnSensorChanged` worden gedefinieerd. En niet alleen dat: in de header van de klasse moet dit al worden aangekondigd. Dat is de reden dat er in de header van de klasse:

```
class KompasView0 : View, ISensorEventListener
```

niet alleen staat vermeld dat het een subklasse is van `View`, maar ook dat het een *implementatie* is van `ISensorEventListener`. Met andere woorden: `KompasView` belooft zich te gedragen zoals dat van een `ISensorEventListener` wordt verwacht.

Zoiets als `ISensorEventListener` heet een *interface*. Een interface is een opsomming van de benodigde methodes. Een klasse die zo'n interface in zijn header vermeldt, belooft om de methodes inderdaad te definiëren. Als dank mag een object van de klasse worden gebruikt als listener, en dat is de reden dat `this` wordt geaccepteerd als eerste parameter van `RegisterListener`.

Implementatie van interface: nakomen van de belofte

De interface `ISensorEventListener` vereist de definitie van twee methodes: niet alleen van `OnSensorChanged`, maar ook van `OnAccuracyChanged`. De laatste wordt aangeroepen als de nauwkeurigheid van de meting is veranderd. Het is verplicht om deze methode te definiëren (beloofd is beloofd!), maar omdat ons kompas zich niet druk maakt over nauwkeurigheid laten we de body leeg.

De body van `OnValueChanged` vullen we natuurlijk wel in. In de parameter van type `SensorEvent` zit nuttige informatie over de `Values` van de sensor. Bij een kompas is alleen `Values[0]` van belang: dat is de kompasrichting waarnaar het device gericht is. We slaan deze informatie op in de variabele `Hoek`, en zorgen met een aanroep van `Invalidate` dat het scherm opnieuw wordt getekend.

Een Bitmap geroteerd tekenen

In de methode `OnDraw` moeten we nu regelen dat het plaatje inderdaad over deze `Hoek` gedraaid wordt. Dat kan worden meegenomen in de transformaties die in de `Matrix`-parameter van `DrawBitmap` bevat zitten. Daar stond al de schaling, en we voegen er nog aan toe:

```
mat.PostRotate(-this.Hoek);
```

Als de gebruiker bijvoorbeeld naar het oosten kijkt is de waarde van `Hoek` gelijk aan 90 graden. Het plaatje moet 90 graden *tegen* de klok in gedraaid worden om te zorgen dat de noordpijl van het kompas links op het scherm terecht komt. Dit is de reden van het minteken in deze opdracht. Maar er moet nog meer gebeuren met de matrix. Het roteren gebeurt namelijk rond het punt $(0, 0)$, dat is een punt in de linker-bovenhoek van het plaatje. We willen echter dat het roteren rond het *middelpunt* van het plaatje gebeurt. Daarom beschrijven we achtereenvolgens de volgende transformaties in de matrix:

```
mat.PostTranslate(-this.Plaatje.Width / 2, -this.Plaatje.Height / 2);
```

Dit transleert (verschuift) het plaatje met z'n halve breedte en hoogte naar links/boven. Nu staat het middelpunt van het plaatje dus op $(0, 0)$.

```
mat.PostScale(this.Schaal, this.Schaal);
```

De schaling moet natuurlijk ook nog steeds gebeuren. Het middelpunt van het plaatje staat nog steeds op $(0, 0)$, alleen is het plaatje groter of kleiner geworden.

```
mat.PostRotate(-this.Hoek);
```

Dit voert de draaiing uit. Omdat het middelpunt nog steeds op $(0, 0)$ staat, wordt het plaatje rond zijn middelpunt gedraaid.

```
mat.PostTranslate(this.Width / 2, this.Height / 2);
```

Dit verschuift het plaatje naar rechts/onder, en wel met de halve breedte/hoogte van het *scherm*. Daarmee komt het middelpunt van het (geschaalde en geroteerde) plaatje in het midden van het scherm te liggen. Daar kunnen we het tekenen:

```
canvas.DrawBitmap(this.Plaatje, mat, verf);
```

6.4 Gestures

Drag: een gesture om te verplaatsen

In veel apps kun je met beweging van vingers de situatie op het scherm manipuleren. Zo'n beweging heet een *gesture*. De simpelste gesture is puur het aanraken van het scherm. Daarmee kun je voor een `Button` een `Click`-event genereren, en op een eigen `View` een `Touch`-event.

Als je met de vinger op het scherm beweegt, heet dat een *drag* gesture. Meestal reageert de app daarop door iets te bewegen over het scherm, zodat je het met je vinger als het ware kan schuiven over het scherm. In de eerste versie van `PuntenKlikker` kon je een icoontje over het scherm slepen: dat was een drag-gesture.

Een snelle beweging over het scherm heet een *swipe* gesture. Sommige apps reageren daar op een andere manier op dan op een drag: in een foto-viewer kun je bijvoorbeeld met een drag-gesture de foto verplaatsen, terwijl je met een swipe-gesture naar de volgende foto gaat.

Bij het neerzetten van je vinger op het scherm, bij elke beweging, en bij het weer loslaten wordt er een `Touch`-event gegenereerd. In de eerste versie van `PuntenKlikker` werd dat event als volgt afgehandeld:

```
public void RaakAan(object o, TouchEventArgs tea)
{   float x = tea.Event.GetX();
    float y = tea.Event.GetY();
    this.punt = new PointF(x, y);
    this.Invalidate();
}
```

Alledrie de soorten `Touch` werden dus op dezelfde manier afgehandeld: de variabele `punt` werd gelijk gemaakt aan de huidige positie, die je uit de `TouchEventArgs` parameter kunt destilleren, en in de `OnDraw` methode werd op die plaats een bitmap getekend. Daardoor volgde het plaatje alle bewegingen van je vinger, en hadden we in feite de *drag* gesture herkend.

In de tweede versie hebben we de event-handler vervangen door:

```
public void RaakAan(object o, TouchEventArgs tea)
{   if (tea.Event.Action == MotionEventActions.Down)
    {   float x = tea.Event.GetX();
        float y = tea.Event.GetY();
        PointF punt = new PointF(x, y);
        punten.Add(punt);
        this.Invalidate();
    }
}
```

Niet alleen werd het punt nu toegevoegd aan een `List`, maar dit gebeurde alleen maar als de `Action` van het event `Down` was. De andere twee soorten (`Move` en `Up`) werden genegeerd, dus kon je met deze versie van de app het icoon, als het eenmaal was neergezet, niet meer verplaatsen.

Met `if`-opdrachten zou je de drie soorten `Touch`-akties op een verschillende manier kunnen afhandelen. Je zou bijvoorbeeld bij een `Move` kunnen kijken of de afstand tot het vorige punt (de oorspronkelijke `Down` of de vorige `Move`) klein of groot is. Is de afstand klein, dan beweegt de gebruiker langzaam en is het dus een *drag* gesture. Is de afstand groot, dan is de beweging snel en betreft het een *swipe* gesture.

Pinch: een gesture om te vergroten

Een complexere gesture is de *pinch* gesture. Hierbij raakt de gebruiker het scherm met twee vingers. Door de vingers uit elkaar te bewegen geeft de gebruiker aan dat de afbeelding op het

scherm groter moet worden; door de vingers naar elkaar toe te knijpen wordt de afbeelding juist kleiner. De gebruiker heeft met deze gesture echt het gevoel de afbeelding op het scherm direct te kunnen manipuleren.

Multitouch: raak met meerdere vingers
Een pinch-gesture werkt natuurlijk alleen maar op een *multitouch* scherm, waarop de beweging van meerdere vingers onafhankelijk van elkaar kan worden gedetecteerd. Gelukkig hebben de meeste Android-devices een multitouch-scherm. Over elke neerzetting, beweging en loslating van elke vinger wordt de `Touch`-eventhandler apart geïnformeerd. Naast `Down` en `Up` zijn er ook `Pointer2Down` en `Pointer2Up` voor het neerzetten en loslaten van de tweede vinger, en `Pointer3Down` en `Pointer3Up` voor de derde vinger die tegelijk het scherm raakt.
Het aantal vingers dat momenteel actief is kun je te weten komen via `tea.Event.PointerCount`, en als die waarde minstens 2 is, kun je de positie van de tweede vinger terugvinden via `tea.Event.GetX(1)`. De positie van de derde vinger krijg je met `tea.Event.GetX(2)`.

Werking van de Pinch
blz. 84 In listing 13 is de kompas-app uitgebreid, zo dat de gebruiker met een pinch-gesture de kompasroos kan vergroten en verkleinen. (Alleen het gedeelte dat anders is dan in de vorige versie wordt in de listing getoond). Erg nuttig is zo'n pinch-gesture niet in een kompas-app, maar het is een mooie gelegenheid om te laten zien hoe een app een pinch-gesture kan herkennen.
Voor de pinch-gesture zijn vier punten van belang:

- `start1` is de startpositie van de eerste vinger
- `huidig1` is de huidige positie van de eerste vinger
- `start2` is de startpositie van de tweede vinger
- `huidig2` is de huidige positie van de tweede vinger

Deze vier variabelen worden in de klasse gedeclareerd, zodat in de `Touch`-eventhandler (de methode `RaakAan`) ook de waarden die daar bij een vorige gelegenheid in werden achtergelaten nog bekeken kunnen worden. (Lokale variabelen worden aan het eind van de methoden weer opgeruimd).
Bij elk `Touch`-event wordt de huidige positie van de eerste vinger vastgelegd. Als er twee vingers actief zijn wordt ook de huidige positie van de tweede vinger vastgelegd.
Het moment van neerzetten van de tweede vinger (`Pointer2Down`) is belangrijk: dit is de startpositie van de pinch-gesture. De op dat moment geldende posities van beide vingers wordt vastgelegd in de start-variabelen.
Daarna wordt (meteen al, maar ook bij elke volgende `Move`) de verhouding uitgerekend tussen de startposities en de huidige posities van de twee vingers. Die verhouding bepaalt de aanpassing van de schaalfactor, en met een extra aanroep van `Invalidate` wordt die meteen in beeld gebracht.

6.5 Detectors

Detector: automatisch afhandelen van gestures
Het herkennen van de pinch-gesture is al met al nog best een gedoe. Omdat dit vaak nodig is in programma's is het ook mogelijk om dit automatisch te laten afhandelen. Voor het herkennen van een gesture zet je dan een *detector* in.
blz. 85 In listing 14 en listing 15 is het kompas-programma nogmaals aangepast, zodat het nu een detector
blz. 85 gebruikt voor het herkennen van de pinch-gesture. Het programma wordt er iets korter van, want de code om de pinch zelf te herkennen komt te vervallen. Echt makkelijker wordt het niet, want nu moet je eerst weer leren hoe je zo'n detector inzet. Het wordt er ook wat saaier van, want de precieze werking van het herkennen van de gesture blijft nu verborgen in library-methodes. Voor (nog) complexere gestures dan de pinch kan het gebruik van een detector echter toch wel gemakkelijk zijn.

Aansturen van de detector
Voor het detecteren van de pinch-gesture moet je een bijpassende detector declareren. Dit gebeurt in de klasse, omdat de variabele in meerdere methodes nodig is:

```
ScaleGestureDetector Detector;
```

In de constructormethode krijgt deze variabele zijn waarde.

```
Detector = new ScaleGestureDetector(context, this);
```

Daarnaast blijft het noodzakelijk om een handler te registreren voor het `Touch`-event:

```
this.Touch += RaakAan;
```

De event-handler zelf is nu echter zeer kort geworden. Hij bestaat uit nog maar één opdracht, waarin de informatie over het event wordt doorgespeeld aan de detector:

```
public void RaakAan(object o, TouchEventArgs tea)
{   Detector.OnTouchEvent(tea.Event);
}
```

In zijn methode `OnTouchEvent` zal de detector ongeveer hetzelfde doen als wat we in de vorige sectie zelf gedaan hebben. Vroeg of laat zal de detector concluderen dat er geschaald moet worden. De detector kan dat niet zelf doen, maar wil het vertellen aan wie het maar horen wil.

Luisteren naar de detector
We willen het plaatje aanpassen als de gebruiker pincht, dus we willen graag door de detector geïnformeerd worden. Dat informeren gebeurt doordat de detector twee speciaal daarvoor bedoelde methoden aanroept. In feite is dit hetzelfde mechanisme als het informeren over de kompasrichting, waarvoor we een `SensorManager` ingezet hebben.
Ook in dit geval is er daarom een `interface`, die we in onze klasse moeten beloven te implementeren. In de header schrijven we daarom:

```
class KompasView2 : View, ISensorEventListener,
                    ScaleGestureDetector.IOnScaleGestureListener
```

De belofte om de interface `ScaleGestureDetector.IOnScaleGestureListener` te implementeren moeten we waarmaken door inderdaad drie methoden te definiëren. De enige werkelijk nuttige daarvan is:

```
public bool OnScale(ScaleGestureDetector detector)
{   this.Schaal *= detector.ScaleFactor;
    this.Invalidate();
    return true;
}
```

De andere twee methodes moeten ook aanwezig zijn (want `IOnScaleGestureListener` eist ze alle drie), maar hebben een weinig interessante invulling:

```
public bool OnScaleBegin(ScaleGestureDetector detector)
{   return true;
}
public void OnScaleEnd(ScaleGestureDetector detector)
{
}
```

Het herkennen van de pinch-gesture is hiermee een samenspraak van ons eigen programma en de detector geworden: wij informeren de detector over het `Touch`-event door aanroep van `OnTouchEvent`, en in ruil informeert hij ons over de schaling door aanroep van `OnScale`.

6.6 Andere sensors

Sensors voor het weer
In de kompas-app registreren we een listener voor de `Orientation`-sensor. Daarmee wordt de sensor voor het magnetische veld aangesproken. Op dezelfde manier kunnen we de andere sensors gebruiken, door een ander `SensorType` mee te geven bij de aanroep van `GetDefaultSensor`. Er zijn bijvoorbeeld sensoren voor diverse aspecten van het weer en andere omgevingsfactoren:

- `AmbientTemperature` voor de temperatuur
- `Pressure` voor de luchtdruk
- `RelativeHumidity` voor de luchtvochtigheid
- `Light` voor de lichtsterkte
- `Proximity` voor de nabijheid van andere objecten (bijvoorbeeld een oor)
- `HeartRate` voor de hartslag van de gebruiker

In de methode `OnSensorChanged` testen welk type sensor het event heeft veroorzaakt:

```
public void OnSensorChanged(SensorEvent e)
{   if (e.Sensor.Type==SensorType.Proximity)
```

Afhankelijk van het sensortype bevat `e.Values` dan de door de sensor gemeten waarde of waardes.

```
        static float Afstand(PointF p1, PointF p2)
        {
            float a = p1.X - p2.X;
            float b = p1.Y - p2.Y;
            return (float)Math.Sqrt(a * a + b * b);
        }

        private PointF start1;
        private PointF start2;
        private PointF huidig1;
        private PointF huidig2;
        private float oudeSchaal;

        public void RaakAan(object o, TouchEventArgs tea)
        {
            huidig1 = new PointF(tea.Event.GetX(0), tea.Event.GetY(0));

            if (tea.Event.PointerCount == 2)
            {
                huidig2 = new PointF(tea.Event.GetX(1), tea.Event.GetY(1));
                if (tea.Event.Action == MotionEventActions.Pointer2Down)
                {
                    start1 = huidig1;
                    start2 = huidig2;
                    oudeSchaal = Schaal;
                }
                float oud = Afstand(start1, start2);
                float nieuw = Afstand(huidig1, huidig2);
                if (oud != 0 && nieuw != 0)
                {
                    float factor = nieuw / oud;
                    this.Schaal = oudeSchaal * factor;
                    this.Invalidate();
                }
            }
        }
    }
}
```

Listing 13: Kompas/KompasView.cs, deel 2 van 2

```
class KompasView2 : View, ISensorEventListener,
                    ScaleGestureDetector.IOnScaleGestureListener
{
    Bitmap Plaatje;
    float Hoek;
    float Schaal;
    ScaleGestureDetector Detector;

    public KompasView2(Context context) : base(context)
    {
        this.SetBackgroundColor(Color.White);
        Detector = new ScaleGestureDetector(context, this);
        this.Touch += RaakAan;
```

Listing 14: Kompas/KompasView2.cs, deel 1 van 3

```
    public void RaakAan(object o, TouchEventArgs tea)
    {
        Detector.OnTouchEvent(tea.Event);
    }

    public bool OnScale(ScaleGestureDetector detector)
    {
        this.Schaal *= detector.ScaleFactor;
        this.Invalidate();
        return true;
    }
    public bool OnScaleBegin(ScaleGestureDetector detector)
    {
        return true;
    }
    public void OnScaleEnd(ScaleGestureDetector detector)
    {
    }
}
```

Listing 15: Kompas/KompasView2.cs, deel 3 van 3

Sensors voor de versnelling
Speciale aandacht verdient de sensor van het type `Accelerometer`. Hiermee meet je de versnellingskrachten die op het device werken, inclusief de zwaartekracht. Het leuke is dat je de zwaartekracht in drie dimensies kunt meten:

- x: langs de horizontale richting van het scherm
- y: langs de verticale richting van het scherm
- z: loodrecht op het scherm

Als het apparaat plat op de tafel stilligt, dan zal de sensor een waarde in de z-richting van 9.8 rapporteren (de zwaartekracht op aarde bedraagt $9.8m/s^2$).
De drie componenten zijn beschikbaar in `e.Values[0]`, `e.Values[1]` en `e.Values[2]`. Uitgaande van deze waardes kun je de ruimtelijke positie van het device bepalen. De fameuze bier-drink app bepaalt bijvoorbeeld de rotatiehoek loodrecht op het scherm met:

```
float x = e.Values[0];
float y = e.Values[1];
float z = e.Values[2];
double hoek = Math.Atan2(x, y) / (Math.PI / 180);
```

Als je deze code in de kompas-app zet, en het sensortype `Orientation` vervangt door `Accelerometer` heb je een verticaal kompas, waarvan de pijl altijd naar boven wijst.

Sensors voor de locatie
Een Android device kan op verschillende manieren de locatie bepalen:

- met een ingebouwde GPS
- door meting van de signaalsterkte van verschillende GSM-zenders

Als je de locatie opvraagt zal het device naar beste kunnen een of meer van deze manieren gebruiken. Het bepalen van de locatie gebeurt op een iets andere manier dan het lezen van de andere sensoren: je gebruikt niet een `SensorManager` maar een `LocationManager`, een klasse uit de library `Android.Locations`. Het registreren van een listener voor de location is iets complexer dan voor de andere sensors:

```
LocationManager lm = (LocationManager)c.GetSystemService(Context.LocationService);
Criteria crit = new Criteria();
crit.Accuracy = Accuracy.Fine;
string lp = lm.GetBestProvider(crit, true);
lm.RequestLocationUpdates(lp, 0, 0, this);
```

Hierin is `c` de `Context` die je in de constructormethode van `View` tot je beschikking hebt. Je kunt de code ook in de methode `OnCreate` van de `Activity` schrijven: dan gebruik je `this` als context. De essentie is dat in `RequestLocationUpdates` het object `this` zich registreert als listener. In de header van zijn klasse moet hij daarom beloven een `ILocationListener` te implementeren:

```
class MijnView : View, ILocationListener
```

Deze belofte moet ook worden nagekomen door het definiëren van een methode `OnLocationChanged`. Hierin kun je dan bijvoorbeeld schrijven:

```
public void OnLocationChanged(Location loc)
{
    double noord = loc.Latitude;
    double oost  = loc.Longitude;
    string info = $"{noord} graden noorderbreedte, {oost} graden oosterlengte";
```

In Utrecht krijgt `noord` daarmee een waarde van ongeveer 52, en `oost` een waarde van ongeveer 5.

Permissie voor het opvragen van de locatie
Bij het installeren van de app moet de eigenaar van het device toestemming geven dat de app de locatie opvraagt. De programmeur moet dit aangeven in het 'Android Manifest', dat deel uitmaakt van de properties van het project. Hierin moet je onder 'Required permissions' aanvinken: `ACCESS_COARSE_LOCATION` en `ACCESS_FINE_LOCATION`.

Hoofdstuk 7

Herhaling en keuze

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

7.1 De while-opdracht

Opdrachten herhalen

In het vorige hoofdstuk zagen we de `foreach`-opdracht, waarmee een opdracht steeds opnieuw kan worden uitgevoerd: een keer voor elk element in een `List`.
Het is ook mogelijk om een opdracht herhaaldelijk uit te voeren zonder dat daarbij een list de sturende factor is. Hiervoor is er weer een andere opdracht vorm beschikbaar: 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 zèlf 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 doorgaan zolang 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 als de waarde van `x` in de body verdubbeld wordt, verhogen we ook de waarde van `n`, zodat die variabele inderdaad de telling 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 `OnDraw` tekent 10 keer dezelfde bitmap (er van uitgaande dat die beschikbaar is in de variabele `bm`) onder elkaar op het scherm:

```
protected override void OnDraw(Canvas canvas)
{
    int n = 0;
    Paint verf = new Paint();
    while (n<10)
    {   canvas.DrawBitmap( bm, 0, 50*n, verf );
        n = n+1;
    }
}
```

Behalve om het aantal herhalingen te 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 steeds verder 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.

7.2 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, vermenigvuldigingen en andere operatoren.

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 toekenningsopdracht. 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"

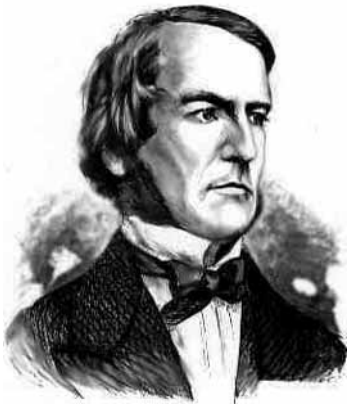
Het type bool

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

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).



Figuur 17: George Boole (1815-1864)

Een voorbeeld van een declaratie van een bool variabele, en een toekenningsopdracht daaraan:

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

Meer zinvol is 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.

7.3 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++;
}
```

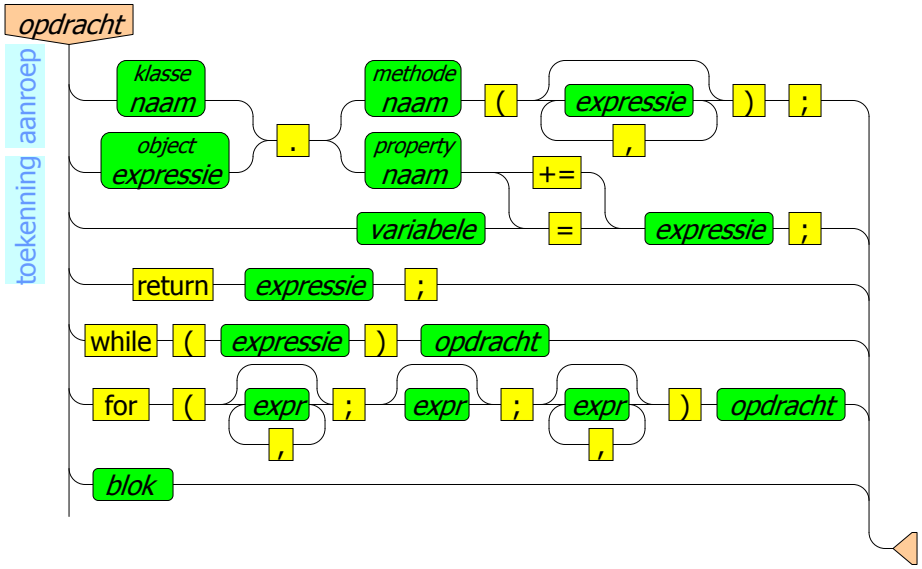
Omdat zo'n "tellende herhaling" zo vaak voorkomt, is er een aparte notatie voor beschikbaar:

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

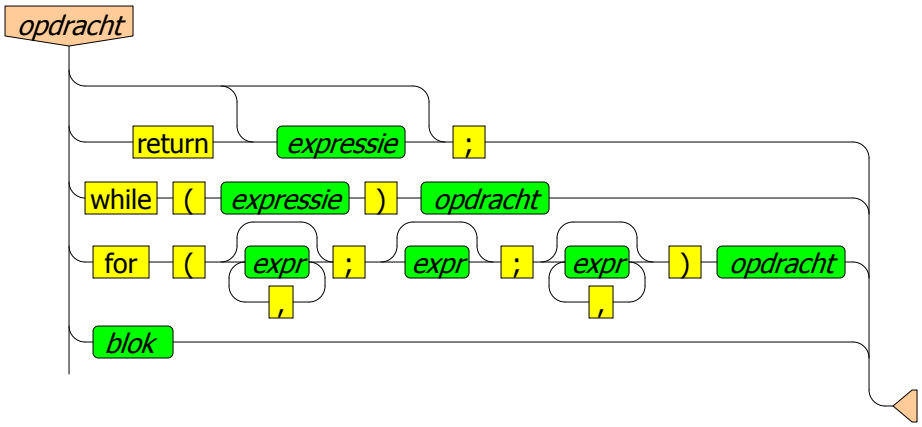
De betekenis van deze `for`-opdracht is precies dezelfde als die van de hierboven genoemde `while`-opdracht. Het voordeel is echter dat de drie dingen die met de teller te maken hebben (de beginwaarde, de eindwaarde en het ophogen) netjes bij elkaar staan in de header. Dat maakt de kans veel kleiner 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 toekennings-opdracht, 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 toekennings-opdracht, 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 toekennings-opdracht vervallen, en in plaats daarvan komt de expressie-met-puntkomma.



In bijlage A 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!

7.4 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:

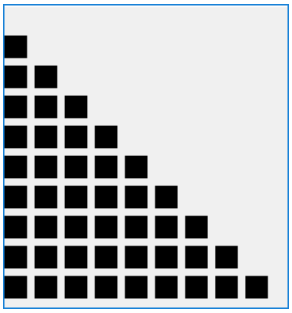
```
while (aantal<10)
{
    x = x*2;
    aantal = aantal+1;   // goed.
}
```

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++)
        canvas.DrawRect(20*x, 20*y, 20*x+15, 20*y+15, verf);
```

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 vierkant op een positie die afhangt van `x` en `y`. Het resultaat is een driehoek-vormig tableau van vierkanten:



Op de bovenste rij in dit tableau staan nul vierkanten. 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.

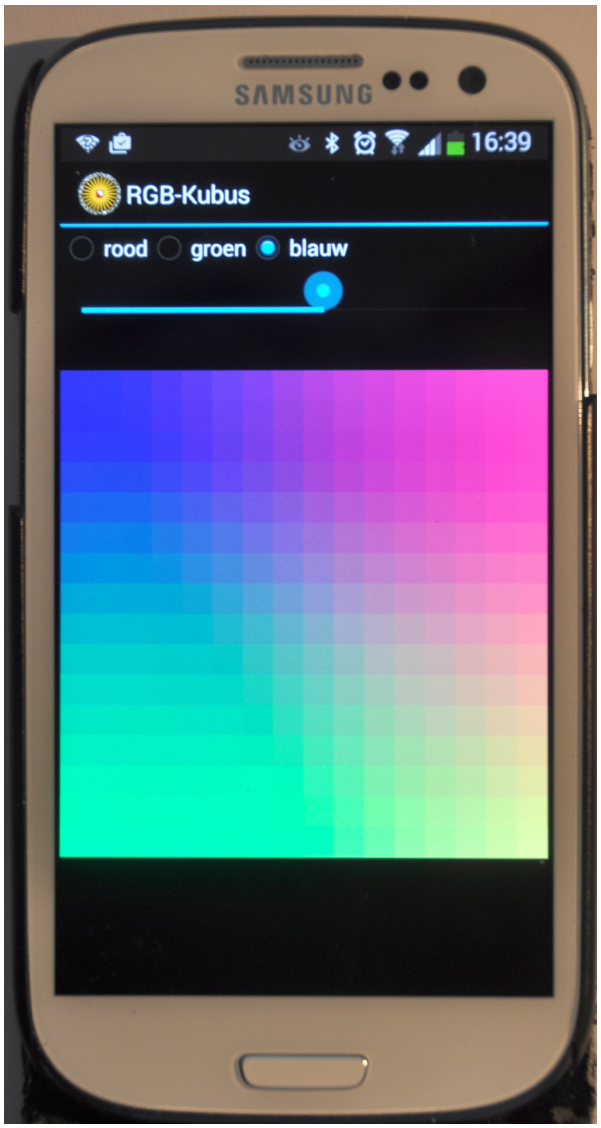
7.5 Toepassing: kleurenkubus

Userinterface van de kleurenkubus

Als toepassing schrijven we een app die een idee geeft van de mogelijke mengkleuren die er bestaan. In sectie 3.2 schreven we al een app die één mengkleur laat zien, maar nu schrijven we een app die alle mogelijke combinaties van groen en blauw laat zien. Met een schuifregelaar kan de gebruiker er dan ook nog rood bij mengen. In figuur 18 is dit programma in werking te zien.

In de userinterface zijn er ook nog drie *radio buttons* aanwezig, met het bijschrift 'rood', 'groen' en 'blauw'. Aan het begin is hiervan 'rood' aangevinkt. Maar de gebruiker kan ook voor 'groen' kiezen. De app laat dan alle mengkleuren van rood en blauw zien, en met de schuifregelaar kan het groen worden bijgemengd.

Het palet van alle mogelijke kleuren wordt vaak de *kleurenruimte* genoemd, omdat er drie dimensies (rood, groen en blauw) zijn die onafhankelijk gekozen kunnen worden. Je kunt je de kleurenruimte voorstellen als een kubus. vol met gekleurde blokjes. De app laat een doorsnede van de kubus zien, en met de schuifregelaar kun je met snijvlak door de kubus bewegen.



Figuur 18: De app Kubus in werking

Opbouw van de userinterface

De opbouw van de userinterface wordt zoals gewoonlijk geregeld in een subklasse van `Activity`. Deze staat in listing 16. blz. 96

De drie radiobuttons worden gegroepeerd in een `RadioGroup`. Dat is een subklasse van `LinearLayout`, die behalve de ordening (in dit geval `Horizontal`) ook voor zijn rekening neemt dat als je een van de buttons in de groep indrukt, bij de voorheen aangevinkte button het vinkje verdwijnt.

De naam 'radio button' verwijst naar antieke radio's (zie figuur 19), waarop vaak fysieke knoppen aanwezig waren om te kiezen uit de lange golf, middengolf en korte golf. In die knoppen zat een veermechanisme dat maakte dat de andere knoppen omhoog sprongen als er eentje werd ingedrukt.

De hele groep met radio-buttons, de schuifregelaar, en de zelfgemaakte `KubusView` zijn op hun beurt gegroepeerd in een verticale `LinearLayout`.

De knoppen hebben een event-handler voor het `Click`-event, de schuifregelaar heeft een event-handler voor het `ProgressChanged`-event. Deze eventhandlers veranderen de waarde van de variabele `Dimensie`, respectievelijk `Waarde` in het `KubusView`-object, en zorgen er daarna met `Invalidate` voor dat de view opnieuw wordt getekend.

Figuur 19: De Grundig 90U radio uit 1955, met echte radio-buttons *(foto: Gerard Tel)*

Tekenen van de kleurenkubus

Het tekenen van de afbeelding wordt zoals gewoonlijk geregeld in de methode `OnDraw` van een eigen subklasse van `View`. Deze staat in listing 17. blz. 97

In de klasse staan twee publieke variabelen gedeclareerd, die ervoor bedoeld zijn om aangepast te worden vanuit de event-handlers in de andere klasse. De tekening die in `OnDraw` wordt gemaakt, wordt beïnvloed door de waardes van deze variabelen.

Met een dubbele `for`-opdracht zorgen we er voor dat er $16 \times 16 = 256$ verschillend gekleurde vlakjes worden getekend. Voor elke vlakje wordt er een aparte `new Color` gemaakt, waarvan de kleur afhangt van de tellers `x` en `y`.

Het aantal vlakjes kan nog worden gevarieerd door de variabele `aantal` een andere waarde te geven. De variabele `stap` wordt daaraan aangepast zo dat altijd het hele bereik van 0 tot en met 255 wordt gebruikt. De variabele `diam` rekent de diameter uit die de vakjes kunnen krijgen zodat het hele tableau precies op het scherm past.

```
using System;
using Android.OS;
using Android.App;
using Android.Widget;

namespace Kubus
{
    [ActivityAttribute(Label = "RGB-Kubus", MainLauncher = true)]
    public class KubusApp : Activity
    {
        SeekBar schuif;
        RadioButton rood, groen, blauw;
        KubusView plaatje;

        protected override void OnCreate(Bundle b)
        {   base.OnCreate(b);
            LinearLayout stapel = new LinearLayout(this); stapel.Orientation = Orientation.Vertical;
            RadioGroup knoppen = new RadioGroup(this); knoppen.Orientation = Orientation.Horizontal;

            rood  = new RadioButton(this); rood.Text  = "rood";
            groen = new RadioButton(this); groen.Text = "groen";
            blauw = new RadioButton(this); blauw.Text = "blauw";
            schuif = new SeekBar(this);    schuif.Max = 255;
            plaatje = new KubusView(this);

            schuif.ProgressChanged += this.veranderd;
            rood .Click += this.dimensie;
            groen.Click += this.dimensie;
            blauw.Click += this.dimensie;

            LinearLayout.LayoutParams par;
            par = new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MatchParent, 120);
            par.BottomMargin = 30; par.TopMargin = 30;

            knoppen.AddView(rood);
            knoppen.AddView(groen);
            knoppen.AddView(blauw);
            stapel .AddView(knoppen);
            stapel .AddView(schuif, par);
            stapel .AddView(plaatje);
            this.SetContentView(stapel);
            knoppen.Check(rood.Id);
        }

        public void veranderd(object o, object ea)
        {   plaatje.Waarde = schuif.Progress;
            plaatje.Invalidate();
        }

        public void dimensie(object o, EventArgs ea)
        {   if (o == rood)  plaatje.Dimensie = 0;
            if (o == groen) plaatje.Dimensie = 1;
            if (o == blauw) plaatje.Dimensie = 2;
            plaatje.Invalidate();
        }
    }
}
```

Listing 16: Kubus/KubusApp.cs

```
using System;
using Android.Content;
using Android.Graphics;
using Android.Views;

namespace Kubus
{
    class KubusView : View
    {
        public int Waarde, Dimensie;

        public KubusView(Context c) : base(c)
        {
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);

            int aantal = 16;
            float stap = 255f / (aantal - 1);
            int diam = Math.Min(this.Width, this.Height) / aantal;
            Paint verf = new Paint();

            for (int x=0; x<aantal; x++)
            {
                for (int y=0; y<aantal; y++)
                {
                    if (this.Dimensie == 0)
                        verf.Color = new Color(this.Waarde, (int)(stap * x), (int)(stap * y));
                    else if (this.Dimensie == 1)
                        verf.Color = new Color((int)(stap * x), this.Waarde, (int)(stap * y));
                    else
                        verf.Color = new Color((int)(stap * x), (int)(stap * y), this.Waarde);

                    canvas.DrawRect(diam*x, diam*y, diam*(x+1), diam*(y+1), verf);
                }
            }
        }
    }
}
```

Listing 17: Kubus/KubusView.cs

7.6 Toepassing: Renteberekening

Rente op rente

blz. 101 Een aantal besproken ideeën komt samen in de app die te zien is in listing 18 en figuur 20 (voor respectievelijk de programmatekst en een screenshot). De userinterface is gemaakt met de Android interface designer. Daarom is de `OnCreate` methode veel korter, omdat de opbouw van de userinterface elders wordt geregeld.
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 `TextView` toegevoegd.

User interface designer

De opbouw van de userinterface is ditmaal ontworpen met de resource designer die bij Android hoort. De opbouw wordt beschreven in een XML-file die, net zoals bitmaps en andere niet-code bestanden, in de resource-directory aanwezig is. Je hoeft niet zelf de XML-file te schrijven: deze wordt automatisch aangepast als je de userinterface aanpast in de resource designer.
In figuur 21 zie je de designer in werking. Zodra je de file `Main.axml` opent, wordt deze getoond via de designer. Zorg dat je ook de Toolbox- en de Properties-windows open hebt staan (deze kun je openen via het View-menu van Visual Studio).
Vanuit de Toolbox kun je nu userinterface-elementen naar de app-in-wording slepen. In het Properties-window kun je daarvan nog extra eigenschappen aanpassen, zoals de teksten in een `TextView` of `Button`. Bovendien krijgt elk element een naam: zoiets als `"@+id/textView2"`. Deze kun je desgewenst nog aanpassen.
In het programma kun je de hele interface tegelijk neerzetten met de opdracht

```
SetContentView(Resource.Layout.Main);
```

Net zoals bij bitmaps wordt een uniek identificatienummer van de resource ontleend aan de klasse `Resource`, die aanwezig is in de automatisch gegenereerde file `Resource.Designer.cs`. Ook elk userinterface-element krijgt een eigen uniek nummer (als deze er niet meteen staan, kun je 'Rebuild' kiezen in het 'Build'-menu van Visual Studio).
In het programma kun je variabelen aanmaken waarmee je de userinterface-elementen nog kunt manipuleren. In plaats van aanroep van de constructormethode, zoals we tot nu toe steeds deden, moet je ze nu zoeken aan de hand van hun resource-identificatienummer:

```
bedragBox = FindViewById<EditText>(Resource.Id.bedragBox);
renteBox  = FindViewById<EditText>(Resource.Id.renteBox);
uitvoer   = FindViewById<TextView>(Resource.Id.uitvoer);
Button button = FindViewById<Button>(Resource.Id.knop);
```

Daarna kun je de variabelen gebruiken zoals je gewend bent, om ze nog extra eigenschappen te geven of er eventhandlers aan te koppelen:

```
bedragBox.Text = "100";
renteBox.Text = "5";
button.Click += klik;
```

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.

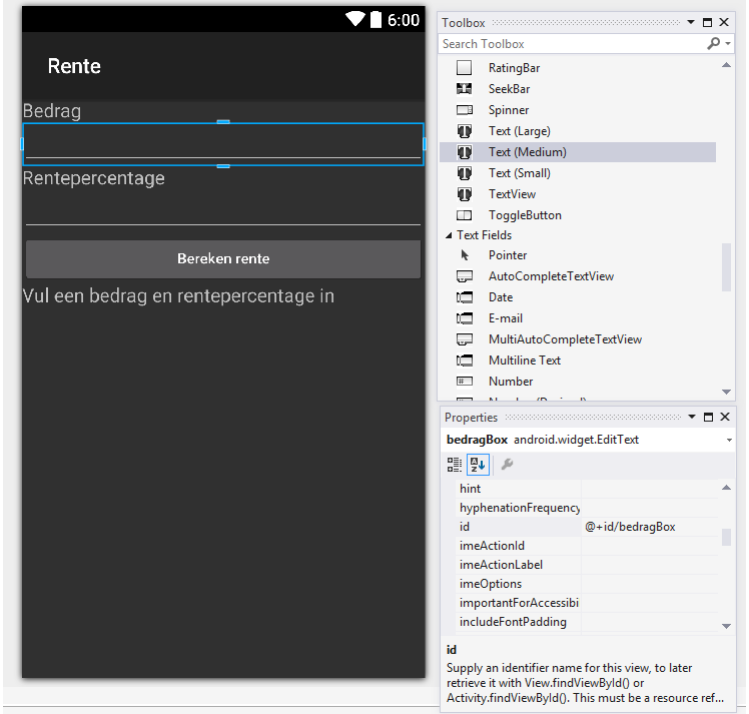
De try-catch opdracht
Je zou kunnen vermijden dat er exceptions ontstaan, door vooraf te controleren of aan alle voorwaarden is voldaan (in het geval van `int.Parse`: of de aangeboden string uitsluitend cijfer-tekens bevat). Maar dan doe je dubbel werk, want `int.Parse` doet die controle nogmaals. Beter is het om te reageren op het optreden van de exception. Dat gebeurt met de speciale `try-catch`-opdracht.
Je kunt de aanroep die mogelijkerwijs 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 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";
}
```

Figuur 20: De app Rente in werking



Figuur 21: Ontwerp van de userinterface in de resource editor

```
using System;
using Android.App;
using Android.Widget;
using Android.OS;
using Android.Views.InputMethods;   // vanwege ImeAction
using System.Globalization;         // vanwege CultureInfo
using Android.Content;

namespace Rente
{
    [Activity(Label = "Rente", MainLauncher = true)]
    public class Rente : Activity
    {
        EditText bedragBox, renteBox;
        TextView uitvoer;
        InputMethodManager imm;

        protected override void OnCreate(Bundle bundle)
        {   base.OnCreate(bundle);
            imm = (InputMethodManager)this.GetSystemService(Context.InputMethodService);
            SetContentView(Resource.Layout.Main);

            bedragBox = FindViewById<EditText>(Resource.Id.bedragBox);
            renteBox = FindViewById<EditText>(Resource.Id.renteBox);
            uitvoer = FindViewById<TextView>(Resource.Id.uitvoer);
            Button button = FindViewById<Button>(Resource.Id.knop);

            bedragBox.Text = "100";
            renteBox.Text = "5";
            button.Click += klik;
            renteBox.EditorAction += toets;
        }

        public void toets(object o, TextView.EditorActionEventArgs ea)
        {   if (ea.ActionId == ImeAction.Done)
            {   imm.HideSoftInputFromInputMethod(this.CurrentFocus.WindowToken, 0);
                klik(o, ea);
            }
        }

        public void klik(object o, EventArgs ea)
        {   try
            {   double bedrag = double.Parse(bedragBox.Text, CultureInfo.InvariantCulture);
                double rente  = double.Parse(renteBox.Text,  CultureInfo.InvariantCulture);
                uitvoer.Text = "";
                for (int jaar = 0; jaar <= 10; jaar++)
                {
                    uitvoer.Text += $"Na {jaar} jaar: {bedrag:F}\n";
                    bedrag *= (1 + 0.01 * rente);
                }
            }
            catch (Exception e)
            {   uitvoer.Text = "Getallen zijn niet correct";
            }
        }
    }
}
```

Listing 18: Rente/Rente.cs

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:p1="http://schemas.android.com/apk/res/android"
    p1:orientation="vertical"
    p1:layout_width="match_parent"
    p1:layout_height="match_parent"
    p1:id="@+id/linearLayout1">
    <TextView
        p1:text="Bedrag"
        p1:textAppearance="?android:attr/textAppearanceMedium"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/textView1" />
    <EditText
        p1:inputType="numberDecimal"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/bedragBox" />
    <TextView
        p1:text="Rentepercentage"
        p1:textAppearance="?android:attr/textAppearanceMedium"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/textView2" />
    <EditText
        p1:inputType="numberDecimal"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/renteBox" />
    <Button
        p1:text="Bereken rente"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/knop"
        p1:textAllCaps="false" />
    <TextView
        p1:text="Vul een bedrag en rentepercentage in"
        p1:textAppearance="?android:attr/textAppearanceMedium"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/uitvoer" />
</LinearLayout>
```

Listing 19: Rente/Resources/layout/Main.axml

Hoofdstuk 8

Goede bedoelingen

8.1 Een App met meerdere activiteiten

Activity en View

Tot nu toe bestonden onze apps uit één (subklasse van) `Activity`, die in zijn 'content view' gegevens aan de gebruiker presenteerde. Die 'content view' kon bestaan uit:

- een simpele `TextView` (in de Hallo-app),
- een simpele `Button` (in de Klikker-app),
- een `LinearLayout` vol met knopjes en schuifjes (in de Mixer-app)
- een zelfgemaakte subklasse van `View` (in de Mondriaan-app)

of combinaties van dit alles. Als er een zelfgemaakte `View` bij betrokken was, dan bestond het programma uit meerdere klassen, die meestal ook in aparte bestanden werden geplaatst.
Maar in alle gevallen was er maar één `Activity`, die door het operating system werd gelanceerd. In dit hoofdstuk bekijken we hoe een activity zelf ook andere activities kan opstarten.

Voorbeeld: Multi, een app met vele functies

Het voorbeeldprogramma is de app 'Multi', die vele functionaliteiten in zich verenigt. Het programma bestaat uit drie klassen, die alledrie een subklasse zijn van `Activity`. Natuurlijk wordt er maar één van deze drie klassen voorzien van het attribuut `MainLauncher=true`, zodat het operating system weet welk van de activities moet worden gelanceerd bij de start van het programma.
De broncode van deze klasse `Multi` staat in listing 20 en listing 21. De andere twee klassen zijn kleine aanpassingen van de Hallo-app (listing 22) en de Klikker-app (listing 23) die we eerder als zelfstandige apps hebben geschreven.
De Multi-app heeft een userinterface die bestaat uit zes knoppen. Het vormt een soort hoofdmenu, van waaruit de gebruiker andere activiteiten kan starten. Met twee van deze knoppen kunnen de Hallo- en de Teller-activiteit worden gestart. Twee andere knoppen starten een zogeheten *dialoog*. En tenslotte zijn er twee knoppen waarmee de gebruiker standaard-apps, zoals een webbrowser en een message-dienst, kan starten. De verzameling van deze zes functies als zodanig is niet buitengewoon nuttig; het dient hier vooral als demonstratie hoe zo'n samengestelde app is opgebouwd.
In de methode `OnCreate` wordt de userinterface opgebouwd. Elk van de zes knoppen krijgt een eigen event-handler voor het `Click`-event, genaamd `klik1` tot en met `klik6`.

blz. 108
blz. 109
blz. 110
blz. 111

8.2 Dialogen

Dialoog: pop-up window voor gegevensinvoer

Een *dialoog* is een pop-up window waarin de gebruiker gegevens kan invoeren. Zolang de dialoog actief is kan de gebruiker niet de rest van de app bedienen. De app blijft, voorzover hij niet wordt afgedekt door de dialoog, wel zichtbaar, maar is enigzins gedimd weergegeven. In figuur 22 is links het hoofdmenu te zien, en daarnaast twee verschillende soorten dialoog: de `DatePickerDialog` en de `AlertDialog`. In een dialoog zijn altijd knoppen aanwezig om de dialoog af te sluiten: met succes ('ja', 'instellen', 'ok') of zonder succes ('nee', 'annuleren', 'cancel'). De gebruiker kan de dialoog ook (zonder succes) afsluiten met de back-knop van de telefoon.

Datum invoeren met DatePickerDialog

Na het indrukken van button `b4` start er een `DatePickerDialog`. In de bijbehorende methode `klik4` staat hoe je dat doet: eerst maak je een object aan van het type `DatePickerDialog`, en vervolgens neem je dat object onder handen in een aanroep van de methode `Show`:

Figuur 22: De app Multi in werking: het hoofdmenu, de DatePickerDialog, en AlertDialog

```
DatePickerDialog d = new DatePickerDialog( this, DatumGekozen
                                         , GebDat.Year, GebDat.Month - 1, GebDat.Day);
d.Show();
```

De tweede parameter is van speciaal belang: het is een event-handler die wordt aangeroepen nadat de gebruiker de dialoog met succes heeft afgesloten. Als de gebruiker de dialoog annuleert wordt deze methode niet aangeroepen.
In het programma moet die event-handler natuurlijk wel gedefinieerd worden:

```
protected void DatumGekozen(object sender, DatePickerDialog.DateSetEventArgs e)
{
    this.GebDat = e.Date;
    b4.Text = $"Geboortedatum: {GebDat.ToString("dd-MM-yyyy")}";
}
```

De tweede parameter is, zoals gebruikelijk bij event-handlers, specifiek voor dit type event. Het bevat de gekozen datum, die we hier gebruiken om member-variabele `GebDat` (die van het type `DateTime` is) een waarde te geven. Een weergave van de datum wordt bovendien op de knop `b4` gezet.
Als de gebruiker nogmaals op de knop `b4` drukt, laat de datumkiezer de eerder gekozen datum weer zien. Dit werkt zo, omdat de onderdelen van `GebDat` bij de aanroep van de constructormethode worden meegegeven. Eigenaardig hierbij is dat de maanden in een `DateTime` zijn genummerd als 1 t/m 12, terwijl `DatePickerDialog` ze als 0 t/m 11 nummert. Daarom schrijven we de correctie `-1` in de vierde parameter.

Ja/nee vragen met AlertDialog

Een `AlertDialog` is bedoeld om dringende mededelingen aan de gebruiker te doen. Ervaren userinterface-ontwerpers weten dat ze hier spaarzaam mee om moeten gaan, want de gebruiker is niet altijd in de stemming om alerts weg te klikken.
In zijn eenvoudigste vorm is een `AlertDialog` alleen maar een bericht met een OK-knop. Maar ze zijn er ook met twee knoppen, waarmee de gebruiker een ja/nee vraag kan beantwoorden. In dit programma gebruiken we zo'n dialoog als de gebruiker knop `b6` indrukt. Dit zal de app afsluiten, maar alleen als de gebruiker de vraag 'nu al stoppen?' met 'ja' beantwoordt.
De benodigde code staat in event-handler `klik6`. Ook nu weer declareren en construeren we een object, ditmaal van type `AlertDialog.Builder`. Voordat we daarvan `Show` aanroepen, roepen we

eerst nog drie andere methoden aan om de dialoog te configureren:

```
AlertDialog.Builder alert = new AlertDialog.Builder(this);
alert.SetTitle("Nu al stoppen?");
alert.SetNegativeButton("nee", NietStoppen);
alert.SetPositiveButton("ja",  WelStoppen);
alert.Show();
```

Elk van de twee buttons (er is eventueel ook nog een derde, 'neutrale' button mogelijk) krijgt een eigen event-handler:

```
protected void NietStoppen(object o, EventArgs ea)
{
}
protected void WelStoppen(object o, EventArgs ea)
{   this.Finish();
}
```

De aanroep van `Finish` (een methode van `Activity`) in de `WelStoppen` event-handler beëindigt de app.

8.3 Lanceren van Activities

MainLauncher: de activiteit waarmee alles begint

In elke app is er precies één `Activity`-subklasse die het attribuut `MainLauncher=true` heeft. Deze wordt automatisch gelanceerd door het operating system. Andere activities kun je zelf lanceren. Dit gebeurt echter niet, zoals bij dialogen, door een object van deze klasse te creëren en er `Show` van aan te roepen. Dit is omdat activiteiten onafhankelijk van elkaar blijven doorwerken. De nieuwe activiteit komt weliswaar in beeld, maar de gebruiker kan ook de oorspronkelijke activiteit weer naar voren roepen.

Intent: bedoeling om een activiteit te beginnen

Het lanceren van een object gebeurt met behulp van een `Intent`-object, dat je kunt aanmaken. Letterlijk betekent deze naam: 'bedoeling', en inderdaad wordt zo'n object gebruikt om aan te geven dat het je bedoeling is dat er een nieuwe activiteit wordt gelanceerd. In het `Intent`-object zet je enkele properties om de details van je bedoeling te beschrijven, en daarna geef je het object als parameter mee aan `StartActivity`.

In ons programma staat een klasse `HalloAct` (zie listing 22). Dit is een subklasse van `Activity`, compleet met een eigen `OnCreate`-methode. Deze heeft natuurlijk niet het `MainLauncher=true` attribuut, want daarvan mag er in elk programm maar één zijn. blz. 110

Als de gebruiker knop `b1` indrukt, wordt er zo'n Hallo-activiteit gelanceerd, omdat in de bijbehorende event-handler `klik1` staat:

```
Intent i = new Intent(this, typeof(HalloAct));
this.StartActivity(i);
```

Let op het verschil met een dialoog: daar wordt een dialog-object *onder handen genomen* door `Show`, hier wordt het `Intent`-object *meegegeven als parameter* aan `StartActivity`.

De tweede parameter van `Intent` is bijzonder. Hij is van het type `Type`, en de enige manier om een waarde van het type `Type` te verkrijgen is het loslaten van het speciale keyword `typeof` op een klassenaam. Dit lijkt op een methode-aanroep, maar is het niet (want de parameter is geen expressie maar een klassenaam). In feite is dit een apart geval in het syntaxdiagram van *expressie*.

String-informatie doorgeven aan een activiteit

De klasse `Hallo` hadden we in hoofdstuk 2 al geschreven. Het laat het woord 'Hallo' in grote blauwe letters op een gele achtergrond zien. In dit programma hebben we er nog een kleine aanpassing aan gedaan: het is nu mogelijk om de tekst die wordt getoond bij het lanceren van de activiteit nog aan te passen.

Deze informatie maakt deel uit van het `Intent`-object waarmee we onze bedoelingen beschrijven. Met een aanroep van `PutExtra` kun je dit in het `Intent`-object aanpassen:

```
Intent i = new Intent(this, typeof(HalloAct));
i.PutExtra("boodschap", "Hallo!!!");
this.StartActivity(i);
```

Aan de ontvangende kant, dus in de methode `OnCreate` van de klasse `Hallo`, is de `Intent` waarmee hij werd gelanceerd beschikbaar:

```
string s = this.Intent.GetStringExtra("boodschap");
if (s==null) s = "geen bericht";
```

Je zou op deze manier meerdere extra's kunnen doorgeven aan de gelanceerde activiteit. Elke extra heeft een naam (in dit geval `"boodschap"`) en een waarde (in dit geval `"Hallo!!!"`).
Als aan de ontvangende kant onverhoopt blijkt dat er geen extra met de opgegeven naam bestaat, geeft `GetStringExtra` de waarde `null` terug. Omdat het gebruik van zo'n `null` string-waarde een crash tot gevolg heeft, doen we een extra check om ook in dit geval de variabele `s` een zinvolle waarde te geven. Weliswaar zal dat hier niet gebeuren (we zien immers aan de zendende kant duidelijk de overeenkomstige `PutExtra` staan!), maar zo'n defensieve programmeerstijl voorkomt urenlange zoekpartijen naar de fout als we een tikfout in de naam van de extra zouden maken.
De samenwerking van `PutExtra` aan de zendende kant, en `GetStringExtra` aan de ontvangende kant, kan in feite worden beschouwd als een manier om informatie door te geven aan een activiteit. Dus wat parameters zijn voor een methode, dat zijn extra's voor een activiteit.

Int-informatie doorgeven aan een activiteit
Behalve voor strings kan een extra ook gebruikt worden om waarden van andere types door te geven. We doen dat bij de lancering van de tweede activiteit: bij het indrukken van knop `b2` wordt de Teller-activiteit gelanceerd, en met de extra `"startwaarde"` kunnen we de startwaarde van de teller nog beïnvloeden.
Aan de zendende kant ziet dit er hetzelfde uit als in het vorige voorbeeld, dus met een aanroep van `PutExtra` die de informatie in het `Intent`-object neerzet:

```
Intent i = new Intent(this, typeof(TellerAct));
i.PutExtra("startwaarde", stand);
this.StartActivity(i);
```

Het verschil is dat de tweede parameter van `PutExtra` ditmaal een `int` is (de variabele `stand`, die als het ware de stand bijhoudt, is boven in de klasse gedeclareerd als `int`).
Aan de ontvangende kant, dus in de methode `OnCreate` van de klasse `Teller`, is er wel een verschil:

```
this.teller = this.Intent.GetIntExtra("startwaarde", 0);
```

De methode `GetIntExtra` is specifiek bedoeld om int-extra's op te halen. De tweede parameter is een default-waarde voor het geval dat de gevraagde extra niet bestaat.
Er zijn vierentwintig (!) verschillende versies van `PutExtra` beschikbaar, voor vierentwintig verschillende types (int, string, double, bool en vele andere). En evenzo zijn er vierentwintig varianten van `Get`*Blabla*`Extra` om ze weer op te halen.

Informatie teruggeven door een activiteit
We zagen dat de extra's van een `Intent` gebruikt worden om informatie door te spelen aan een activiteit, zoals parameters worden doorgegeven bij de aanroep van een methode. Is er ook een manier waarop een activiteit iets terug kan zeggen tegen degene die hem lanceerde, zoals methodes een `return`-waarde kunnen teruggeven aan de methode die ze aanriep?
Ja, dit kan, en het gebeurt eigenlijk op dezelfde manier als de informatie-overdracht de andere kant op: door middel van een `Intent`-object en de extra's daarvan. Een voorbeeld staat in de Teller-activiteit.
Speciaal voor dit doel doen we een `override` van de methode `Finish`. Dit is de methode die wordt aangeroepen bij het einde van de levenscyclus van een activiteit.

```
public override void Finish()
{   Intent i = new Intent();
    i.PutExtra("eindwaarde", this.teller);
    this.SetResult(Result.Ok, i);
    base.Finish();
}
```

Hoe kunnen we aan de andere kant, dus in de klasse `Multi`, deze waarde dan weer opvangen?
Voor dit doel moeten we de speciaal hiervoor bedoelde methode `OnActivityResult` een nieuwe invulling geven, door deze met `override` te definiëren:

```
protected override void OnActivityResult(int code, Result res, Intent data)
{   base.OnActivityResult(code, res, data);
    if (code==TellerCode && res==Result.Ok)
    {   this.stand = data.GetIntExtra("eindwaarde", 0);
        b2.Text = $"Teller: {stand}";
    }
}
```

Omdat deze methode wordt aangeroepen door *elk* van de gelanceerde deel-activiteiten, onderscheiden ze zich door middel van een unieke code. Deze code werd vastgelegd toen de activiteit werd gelanceerd. Het lanceren gebeurt namelijk door middel van `StartActivityForResult`, waaraan we de code kunnen meegeven:

```
Intent i = new Intent(this, typeof(TellerAct));
i.PutExtra("startwaarde", stand);
this.StartActivityForResult(i, TellerCode);
```

De waarde van `TellerCode` mogen we zelf kiezen. We leggen hem vast door middel van een constante, zodat we ons niet kunnen vergissen op de twee plekken waar deze code nodig is:

```
const int TellerCode = 12345;
```

Een standaard-intent: webbrowsen

Behalve eigen activiteiten kun je met behulp van een `Intent` ook standaard-apps lanceren. Voor dit doel zijn er enkele constructoren van `Intent` beschikbaar waaraan je kunt meegeven welke standaard-dienst je wilt gebruiken. Die gebruiken we om een web-browser te lanceren:

```
string Website = "http://students.uu.nl/beta/informatiekunde";
Intent i = new Intent(Intent.ActionView, Android.Net.Uri.Parse(Website));
this.StartActivity(i);
```

Op deze manier kun je standaard-diensten naadloos integreren in je eigen apps. Met de waarde `ActionView` geef je aan dat je een webbrowser wenst.

Een standaard-intent: informatie delen met vrienden

Een ander voorbeeld van een standaard-dienst is het delen van informatie met je vrienden, via mail, sms, whatsapp en what not. Ditmaal is de eerste parameter van `Intent` de waarde `ActionSend`. De gebruiker krijgt dan de keus uit alle apps die bij hun installatie hebben aangegeven dat ze berichten kunnen versturen.

```
string bericht = "Kom je op mijn verjaardagsfeestje?";
Intent i = new Intent(Intent.ActionSend);
i.SetType("text/plain");
i.PutExtra(Intent.ExtraText, bericht);
this.StartActivity(i);
```

Door andere parameters van `SetType` te gebruiken kun je ook andere media (foto's, films) versturen.

Rekenen met datums: DateTime en TimeSpan

Om toch nog iets leuks te doen met deze verder tamelijk saaie app, zit er programmatuur in waarmee je je vrienden kunt uitnodigen voor je 'verKdagdag': de dag dat je een heel duizendtal dagen oud wordt. Driemaal zo zeldzaam als een gewone verjaardag, en daarom aanleiding voor een extra groot feest (dat bovendien ook eens in een ander seizoen valt dan je gewone verjaardag). De code in methode `klik5` spreekt hopelijk voor zichzelf. Hij maakt gebruik van de standaardklassen `DateTime` (voor een datum plus een tijdstip, waarvan we in dit geval alleen het datum-gedeelte nodig hebben) en `TimeSpan` (voor een tijdsduur). Dit soort waarden kun je optellen en aftrekken zoals je dat zou verwachten (het verschil van twee `DateTime`-objecten geeft een `TimeSpan`, een `DateTime` plus een `TimeSpan` geeft een nieuwe `DateTime`, enzovoorts). Zie methode `klik5` in listing 21 voor de uitwerking hiervan. blz. 109

```
using System;
using Android.App;
using Android.Content;
using Android.Widget;
using Android.OS;

namespace Multi
{
    [Activity(Label = "Multi", MainLauncher = true)]
    public class Multi : Activity
    {
        Button b1, b2, b3, b4, b5, b6;
        int stand = 0;
        DateTime GebDat = DateTime.Now;
        const int TellerCode = 12345;
        const string Website = "http://students.uu.nl/beta/informatiekunde";

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            LinearLayout stapel = new LinearLayout(this); stapel.Orientation = Orientation.Vertical;
            b1 = new Button(this); b1.Text = "Hallo";        b1.Click += klik1; stapel.AddView(b1);
            b2 = new Button(this); b2.Text = "Teller";       b2.Click += klik2; stapel.AddView(b2);
            b3 = new Button(this); b3.Text = "Website";      b3.Click += klik3; stapel.AddView(b3);
            b4 = new Button(this); b4.Text = "Datum kiezen"; b4.Click += klik4; stapel.AddView(b4);
            b5 = new Button(this); b5.Text = "Delen";        b5.Click += klik5; stapel.AddView(b5);
            b6 = new Button(this); b6.Text = "Afsluiten";    b6.Click += klik6; stapel.AddView(b6);
            SetContentView(stapel);
        }

        public void klik1(object o, EventArgs ea)
        {
            Intent i = new Intent(this, typeof(HalloAct));
            i.PutExtra("boodschap", "Hallo!!!");
            this.StartActivity(i);
        }

        public void klik2(object o, EventArgs ea)
        {
            Intent i = new Intent(this, typeof(TellerAct));
            i.PutExtra("startwaarde", stand);
            this.StartActivityForResult(i, TellerCode);
        }
        protected override void OnActivityResult(int code, Result res, Intent data)
        {
            base.OnActivityResult(code, res, data);
            if (code == TellerCode && res == Result.Ok)
            {
                this.stand = data.GetIntExtra("eindwaarde", 0);
                b2.Text = $"Teller: {stand}";
            }
        }

        public void klik3(object o, EventArgs ea)
        {
            Intent i = new Intent(Intent.ActionView, Android.Net.Uri.Parse(Website));
            this.StartActivity(i);
        }
```

Listing 20: Multi/Multi.cs, deel 1 van 2

```
        public void klik4(object o, EventArgs ea)
        {
            DatePickerDialog d = new DatePickerDialog(this, DatumGekozen,
                                        GebDat.Year, GebDat.Month - 1, GebDat.Day);
            d.Show();
        }
        protected void DatumGekozen(object sender, DatePickerDialog.DateSetEventArgs e)
        {
            this.GebDat = e.Date;
            b4.Text = $"Geboortedatum: {GebDat.ToString("dd-MM-yyyy")}";
        }

        public void klik5(object o, EventArgs ea)
        {
            DateTime nu = DateTime.Now;
            TimeSpan tijd = nu - this.GebDat;
            int dagenOud = (int)tijd.TotalDays;
            if (dagenOud > 0)
            {
                int nachtjesSlapen = 1000 - dagenOud % 1000;
                DateTime wanneer = nu + new TimeSpan(nachtjesSlapen, 0, 0, 0);
                string feestdag = wanneer.ToString("dd MMM yyyy");
                string bericht = $"Op {feestdag} vier ik mijn {1+dagenOud/1000}e verKdagdag.\n"
                               + "Kom je ook?";

                Intent i = new Intent(Intent.ActionSend);
                i.SetType("text/plain");
                i.PutExtra(Intent.ExtraText, bericht);
                this.StartActivity(i);
            }
        }

        public void klik6(object o, EventArgs ea)
        {
            AlertDialog.Builder alert = new AlertDialog.Builder(this);
            alert.SetTitle("Nu al stoppen?");
            alert.SetNegativeButton("nee", NietStoppen);
            alert.SetPositiveButton("ja", WelStoppen);
            alert.Show();
        }
        protected void NietStoppen(object o, EventArgs ea)
        {
        }
        protected void WelStoppen(object o, EventArgs ea)
        {
            this.Finish();
        }
    }
```

Listing 21: Multi/Multi.cs, deel 2 van 2

```
using Android.OS;       // vanwege Bundle
using Android.App;      // vanwege Activity
using Android.Widget;   // vanwege TextView
using Android.Graphics; // vanwege Color

namespace Multi
{
    [ActivityAttribute(Label = "Hallo")]
    public class HalloAct : Activity
    {
        protected override void OnCreate(Bundle b)
        {
            base.OnCreate(b);

            string s = this.Intent.GetStringExtra("boodschap");
            if (s == null) s = "geen bericht";

            // of korter:
            string s2 = this.Intent.GetStringExtra("boodschap") ?? "geen bericht";

            TextView scherm;
            scherm = new TextView(this);
            scherm.Text = s;
            scherm.TextSize = 80;
            scherm.SetBackgroundColor(Color.Yellow);
            scherm.SetTextColor(Color.DarkBlue);

            this.SetContentView(scherm);
        }
    }
}
```

Listing 22: Multi/Hallo.cs

```
using System;           // vanwege EventArgs
using Android.App;      // vanwege Activity
using Android.Widget;   // vanwege Button
using Android.OS;       // vanwege Bundle
using Android.Content;  // vanwege Intent

namespace Multi
{
    [ActivityAttribute(Label = "Teller")]
    public class TellerAct : Activity
    {
        int teller;
        Button knop;

        protected override void OnCreate(Bundle b)
        {   base.OnCreate(b);
            this.teller = this.Intent.GetIntExtra("startwaarde", 0);
            this.knop = new Button(this.BaseContext);
            this.knop.Text = "Klik hier!";
            this.knop.TextSize = 40;
            this.knop.Click += this.klik;
            this.SetContentView(knop);
        }

        public void klik(object o, EventArgs ea)
        {   this.teller = this.teller + 1;
            this.knop.Text = this.teller.ToString() + " keer geklikt";
        }

        public override void Finish()
        {   Intent i = new Intent();
            i.PutExtra("eindwaarde", this.teller);
            this.SetResult(Result.Ok, i);
            base.Finish();
        }
    }
}
```

Listing 23: Multi/Teller.cs

8.4 Verkorte notaties

Veel dingen kun je op verschillende manieren programmeren. Soms kan het efficiënter (minder tijd- of geheugengebruik) zijn om iets op een bepaalde manier aan te pakken. Soms ook maakt het voor de snelheid van het programma niet uit, maar is een andere aanpak duidelijker, en gemakkelijker te debuggen. Je kunt dingen omslachtig opschrijven of compact formuleren. In deze sectie herschrijven we de klasse `Multi` waarbij we steeds voor een kortere formulering kiezen. Soms wordt het daar overzichtelijker van, soms juist niet. Kijk en oordeel zelf.

blz. 115 Het aangepaste programma staat in listing 24. Het is in ieder geval korter, want het past nu op één bladzijde (alleen de `using`-regels en de namespace-header zijn weggelaten). Sommige verkortingen werken altijd, sommige gebruiken speciale notaties die in C# in de loop van de versie-geschiedenis zijn ingevoerd, en sommige maken gebruik van handigheidjes die in de library beschikbaar zijn.

Expressies voor variabelen invullen

Een verkorting die altijd mogelijk is: als je een variabele declareert en met een toekenningsopdracht een waarde geeft, en je gebruikt die variabele vervolgens maar één keer, dan had je net zo goed de expressie uit de toekenningsopdracht meteen kunnen opschrijven in plaats van waar je de variabele gebruikt. In een simpel voorbeeld: in plaats van

```
Color c = new Color(100,50,30);
verf.Color = c;
```

had je ook meteen kunnen schrijven:

```
verf.Color = new Color(100,50,30);
```

Dit geldt ook voor objecten die meegeeft aan een methode. In plaats van

```
MondriaanView schilderij;
schilderij = new MondriaanView(this);
this.SetContentView(schilderij);
```

had je ook meteen kunnen schrijven:

```
this.SetContentView(new MondriaanView(this));
```

Deze situatie komt ook een paar keer voor in het Multi-programma. In plaats van

```
Intent i = new Intent(Intent.ActionView, Android.Net.Uri.Parse(Website));
this.StartActivity(i);
```

kunnen we `Intent`-object ook meteen meegeven aan `StartActivity`, zonder het eerste een naam te geven:

```
this.StartActivity(new Intent(Intent.ActionView, Android.Net.Uri.Parse(Website)));
```

En net een beetje anders: een object in een variabele zetten, en vervolgens onder handen nemen met een methode, zoals in:

```
DatePickerDialog d = new DatePickerDialog( this, DatumGekozen
                                         , GebDat.Year, GebDat.Month - 1, GebDat.Day);
d.Show();
```

kan ook direct met

```
new DatePickerDialog(this, DatumGekozen, GebDat.Year, GebDat.Month - 1, GebDat.Day).Show();
```

Een keten van methode-aanroepen

De hierboven genoemde aanpak om variabelen uit te sparen lukt niet als de variabele twee keer nodig is, bijvoorbeeld omdat het object, voordat het aan een methode wordt meegegeven, eerst nog onder handen genomen wordt:

```
Intent i = new Intent(this, typeof(HalloAct));
i.PutExtra("boodschap", "Hallo!!!");
this.StartActivity(i);
```

Maar de auteur van de klasse `Intent` heeft daar een handigheidje op bedacht. De methode `PutExtra` lijkt op het eerste gezicht een `void`-methode: hij doet iets, maar levert geen resultaatwaarde op. Maar in werkelijkheid levert de methode wèl iets op: het zojuist aangepaste object zelf. De methode `PutExtra` staat in de klasse `Intent`, en neemt dus een `Intent` onder handen.

De methode heeft echter ook het type `Intent` als resultaat. De auteur van de methode `PutExtra` hoefde alleen maar op de laatste regel te schrijven:

```
return this;
```

Baat het niet dan schaadt het niet: als je het resultaat negeert, zoals in de voorbeeld-aanroep hierboven, dan blijft het gewoon werken.
Maar dankzij de resultaatwaarde van `PutExtra` kun je dat resultaat meteen weer meegeven aan `StartActivity`. En daarmee is de variabele `i` nog maar één keer in gebruik, en kun je hem dus helemaal wegwerken volgens de eerder genoemde aanpak. Dan blijft over:

```
this.StartActivity(new Intent(this, typeof(HalloAct)).PutExtra("boodschap", "Hallo!!!"));
```

Veel overzichtelijker wordt het er niet van, maar het is wel leuk dat het kan...
Ook de auteur van de klasse `AlertDialog.Builder` heeft dit mogelijk gemaakt, door alle methodes in plaats van `void` het object weer te laten teruggeven. Dat maakt dat je, in plaats van vijf losse opdrachten:

```
AlertDialog.Builder alert = new AlertDialog.Builder(this);
alert.SetTitle("Nu al stoppen?");
alert.SetNegativeButton("nee", NietStoppen);
alert.SetPositiveButton("ja",  WelStoppen );
alert.Show();
```

alles in één grote opdracht kunt schrijven, waarin het resultaat van elke methode meteen weer onder handen wordt genomen door de volgende:

```
new AlertDialog.Builder(this) . SetTitle("Nu al stoppen?")
                              . SetNegativeButton("nee", NietStoppen)
                              . SetPositiveButton("ja",  WelStoppen )
                              . Show();
```

Om dit een beetje overzichtelijk te houden moet je het toch over meerdere regels verdelen, maar syntactisch is dit maar één opdracht! Behalve dat dit een gevoel van schoonheid oproept, maakt het straks ook weer een andere verkorting mogelijk.

De lambda-expressie
In een interactief programma (en welk programma is dat tegenwoordig nou niet) zijn er veel event-handlers nodig. Event-handlers zijn vaak maar korte methodes, die ook maar één keer gebruikt worden, namelijk bij het registreren van de event-handler. Bijvoorbeeld de registratie:

```
alert.SetPositiveButton("ja",  WelStoppen );
```

waarbij dan apart de methode `WelStoppen` is gedefinieerd:

```
protected void WelStoppen(object o, EventArgs ea)
{   this.Finish();
}
```

Je zou hier eigenlijk de hele methode wel meteen willen opschrijven op de plek waar de naam `WelStoppen` nu nog staat. Dit is precies wat er mogelijk is met de zogenoemde *lambda-expressie*, die bestaat sinds C# versie 3. Je kunt de hele methode aanduiden *zonder hem een naam te geven* met deze expressie:

```
(object o, EventArgs ea) => { this.Finish(); }
```

Het symbool `=>` heeft niets te maken met 'gelijk of groter' (de correcte vergelijkings-operator is `>=`), maar moet gelezen worden als een soort pijltje: ⇒.
Als deze expressie wordt gebruikt in een situatie waarin het al wel duidelijk is wat het type van de parameters is, dan mag je die types ook weglaten:

```
(o, ea) => { this.Finish(); }
```

en als de methode maar een body van één opdracht heeft, mogen de accolades en de puntkomma ook weg:

```
(o, ea) => this.Finish()
```

Wat overblijft is de essentie van de event-handler, en die kun je in zijn geheel opschrijven op de plek waar hij nodig is:

```
alert.SetPositiveButton("ja", (o, ea) => this.Finish() );
```

Omdat na de eerder al beschreven verkortingen bijna alle `klik`-eventhandlers nog maar uit één opdracht bestaan, kunnen deze ook allemaal met een lambda-expressie worden genoteerd. De namen `klik1` etcetera zijn dus ook niet meer nodig, en we kunnen meteen opschrijven:

```
b1.Click += (o,ea)=>StartActivity(new Intent(this, typeof(HalloAct)).PutExtra("boodschap", "Hallo!!!"));
```

Maakt dat het programma duidelijker? Hmm, misschien niet, maar allemaal losse event-handlers met allemaal verschillende namen is ook niet altijd even overzichtelijk!

Een alternatief voor null

De nieuwste aanwinst in C# versie 6 is een notatie voor een situatie die we in dit programma gebruikt hebben:

```
string s = this.Intent.GetStringExtra("boodschap");
if (s == null) s = "geen bericht";
```

Voor deze situatie, waar een expressie mogelijk de waarde `null` heeft, in welk geval we een default-waarde willen gebruiken, is er een nieuwe operator beschikbaar: `??`. Zo heet-ie echt: twee vraagtekens. Daarmee kun je opschrijven:

```
string s = this.Intent.GetStringExtra("boodschap") ?? "geen bericht";
```

Hoewel de ruimtewinst beperkt is, stimuleert deze notatie de voorzorgsmaatregel om waarden die `null` kunnen zijn altijd van een default-alternatief te voorzien. Dit kan veel ellende met runtime fouten voorkomen.

```
[Activity(Label = "Multi2", MainLauncher = false)] // maak dit true als je Multi2
public class Multi2 : Activity                     // wilt gebruiken i.p.v. Multi
{   Button b1, b2, b3, b4, b5, b6;
    DateTime GebDat = DateTime.Now;
    const int TellerCode = 12345;
    const string Website = "http://students.uu.nl/beta/informatiekunde";

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        LinearLayout stapel = new LinearLayout(this); stapel.Orientation = Orientation.Vertical;
        b1 = new Button(this); b1.Text = "Hallo";        stapel.AddView(b1);
        b2 = new Button(this); b2.Text = "Teller";       stapel.AddView(b2);
        b3 = new Button(this); b3.Text = "Website";      stapel.AddView(b3);
        b4 = new Button(this); b4.Text = "Datum kiezen"; stapel.AddView(b4);
        b5 = new Button(this); b5.Text = "Delen";        stapel.AddView(b5);
        b6 = new Button(this); b6.Text = "Afsluiten";    stapel.AddView(b6);

        b1.Click += (o,ea)=>StartActivity(new Intent(this, typeof(HalloAct))
                                          . PutExtra("boodschap", "Hallo!!!"));
        b2.Click += (o,ea)=>StartActivityForResult(new Intent(this,typeof(TellerAct)), TellerCode);
        b3.Click += (o,ea)=>StartActivity(new Intent( Intent.ActionView
                                                    , Android.Net.Uri.Parse(Website)));
        b4.Click += (o,ea)=>new DatePickerDialog( this, DatumGekozen
                                                , GebDat.Year,GebDat.Month-1,GebDat.Day).Show();
        b5.Click += klik5;
        b6.Click += (o, ea) => new AlertDialog.Builder(this)
                               . SetTitle("Nu al stoppen?")
                               . SetNegativeButton("nee", (ob, e) => { }           )
                               . SetPositiveButton("ja",  (ob, e) => this.Finish() )
                               . Show();
        SetContentView(stapel);
    }

    protected override void OnActivityResult(int code, Result res, Intent data)
    {   base.OnActivityResult(code, res, data);
        if (code == TellerCode && res == Result.Ok)
            b2.Text = $"Teller: {data.GetIntExtra("eindwaarde", -1)}";
    }

    protected void DatumGekozen(object sender, DatePickerDialog.DateSetEventArgs e)
    {   this.GebDat = e.Date;
        b4.Text = $"Geboortedatum: {GebDat.ToString("dd-MM-yyyy")}";
    }

    public void klik5(object o, EventArgs ea)
    {   int dagen = (int)(DateTime.Now - this.GebDat).TotalDays;
        if (dagen > 0)
        {   string bericht
            = $"Op {(DateTime.Now + new TimeSpan(1000-dagen%1000,0,0,0)).ToString("dd MMM yyyy")}"
            + $"vier ik mijn {1 + dagen / 1000}e verKdagdag.\nKom je ook?";
            this.StartActivity(new Intent(Intent.ActionSend)
                               . SetType("text/plain")
                               . PutExtra(Intent.ExtraText, bericht) );
        }
    }
}
```

Listing 24: Multi/Multi2.cs, deel 1 van 1

Hoofdstuk 9

Klassen, Strings, en Arrays

9.1 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 `Activity`, en vaak ook nog een subklasse van `View`) met daarin methoden zoals `OnCreate` respectievelijk `OnDraw`, 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 Android-weetjes 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` (wat overigens geen klasse is maar een `struct`, maar dat is nu even niet zo belangrijk). Onze klasse `Kleur` is niet Android-specifiek: het is een klasse die in alle C#-programma's gebruikt zou kunnen worden, ook voor andere platforms dan Android.
De programmatekst staat in listing 25. Een voorbeeld van hoe deze klasse in een programma blz. 118
gebruikt zou kunnen worden staat in listing 26. blz. 119

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

```
Kleur oranje, paars;
```

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

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

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

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

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

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

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

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

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

Listing 25: KleurKlasse/Kleur.cs

```
using System;

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

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

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

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

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

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

Listing 26: KleurKlasse/Voorbeeld.cs

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

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

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

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

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

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

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

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

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

Een voorbeeld van gebruik van deze constructoren is:

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

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

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

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

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

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

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

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

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

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

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

De methode ToString

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

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

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

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

Een string terug-converteren naar een object

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

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

Static declaraties

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

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

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

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

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

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

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

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

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

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

9.2 Strings

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 achter de string, 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` voor het eerst in de string voorkomt (en levert -1 als dat nergens is)
- `string Insert(int p, string s)`: voegt `s` in op positie `p` in de string
- `string[] Split()`: splitst de string op in losse woorden, en levert een *array* van strings op, met in elk array-element een enkel woord. De woorden worden gescheiden door spaties of andere *whitespace*, zoals tabulaties of regelovergangen.
- `string[] Split(char c)`: een variant van `Split`, waarbij je zelf kunt opgeven door welk symbool de woorden worden gescheiden.

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: `teller.ToString() + "keer geklikt"`.
- 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 bij 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-Œr 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 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`

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;
```

9.3 Arrays

Een string is te beschouwen als een rij lettertekens. Behalve dit soort rijtjes van `char` variabelen, is het ook mogelijk om rijtjes van andere types te maken: rijtjes van `int`, van `double` of zelfs van objecten zoals `Color` of `string`. Zo'n rij variabelen heet een *array*.

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:

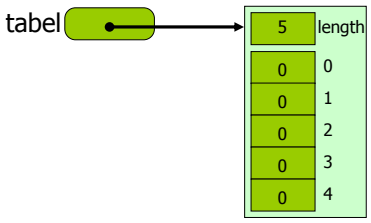
```
int [] tabel;
```

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:

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;
```

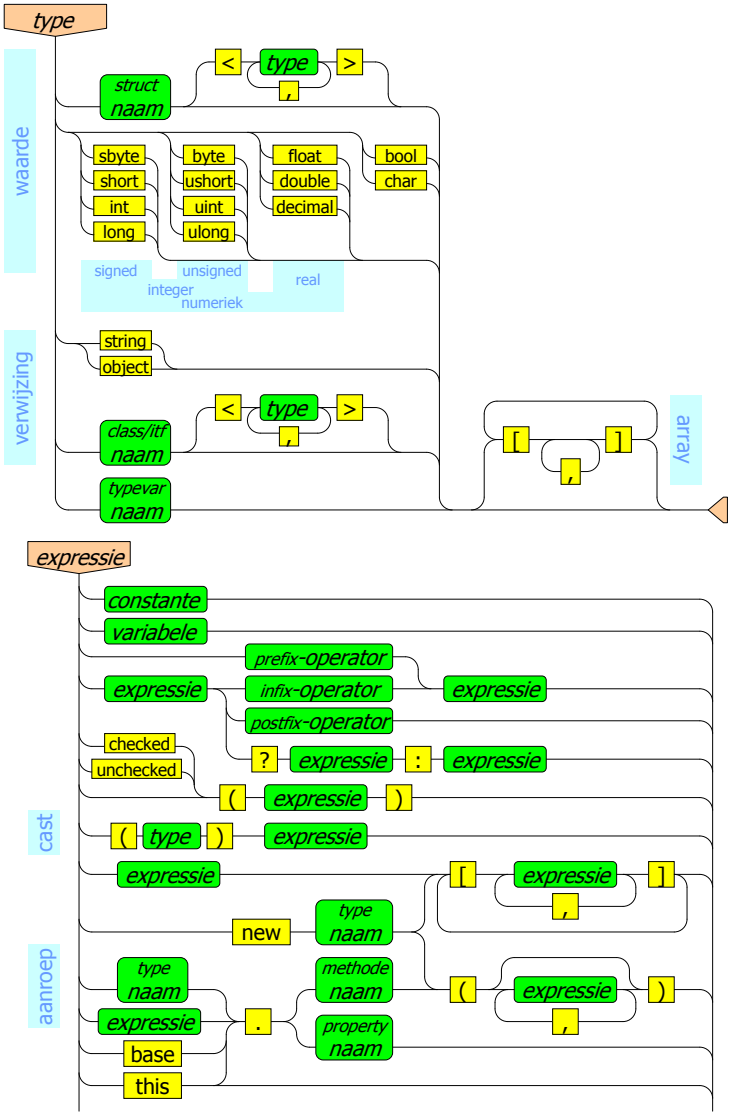
maar dat is, zeker bij lange arrays, natuurlijk erg omslachtig. Je kunt de regelmaat in deze serie opdrachten echter uitbuiten, door op de plaats waar het volgnummer (0, 1, 2, 3 en 4) staat, een variabele te schrijven. In een for-opdracht kun je deze variabele dan alle gewenste volgnummers laten langslopen. De `Length`-property kan mooi dienen als bovengrens in deze for-opdracht:

```
int nummer;
for (nummer=0; nummer<tabel.Length; nummer++)
    tabel[nummer] = 42;
```

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 fundamenteler 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 23). 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 23).
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).



Figuur 23: Syntax van ‘type’ en ‘expressie’ maakt arrays mogelijk

Initialisatie van arrays
Een array is een object. Dus als je een array declareert, maak je ruimte voor een *verwijzing* naar een array-object, 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];
```

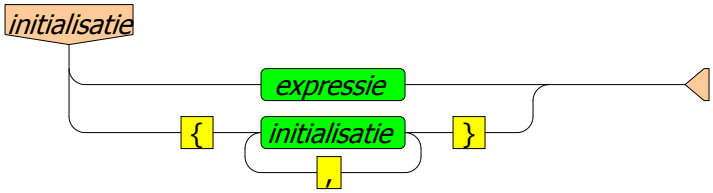
Nu bestaat de array wel, maar die zeven strings hebben ieder nog geen waarde gekregen. Dat kan nog een heel gedoe worden:

```
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 tusen 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};
```

9.4 Een programma voor tekst-analyse

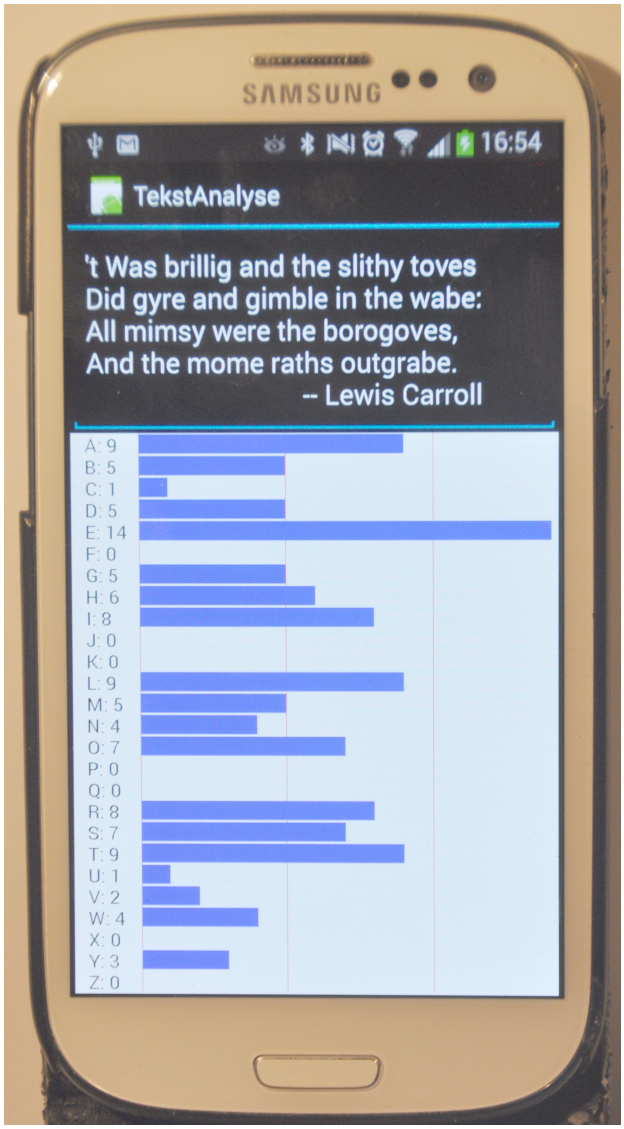
We gaan strings en arrays gebruiken in een complete app. In deze app kan de gebruiker een tekst invoeren. Van deze tekst wordt de frequentie van de letters geanalyseerd. In een staafdiagram toont de app hoe vaak elke letter in de tekst voorkomt. In figuur 24 is dit programma in werking te zien.

Opbouw van de TekstAnalyse-app
Het programma bestaat zoals gewoonlijk uit twee klassen:

blz. 132
- `TekstAnalyse`: een subklasse van `Activity` (zie listing 27)

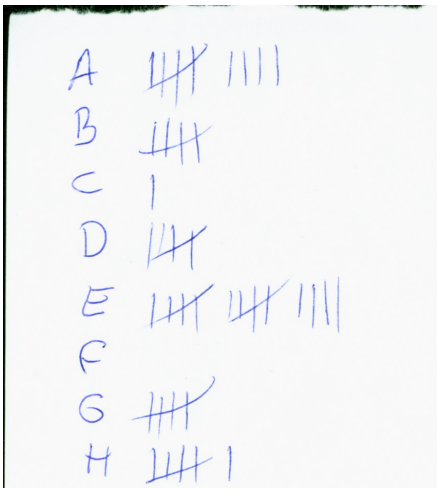
blz. 133
- `DiagramView`: een subklasse van `View` (zie listing 28)

De `Activity`-subklasse is zoals gewoonlijk verantwoordelijk voor de opbouw van de userinterface. Die bestaat uit twee onderdelen: een `EditText` waarin de gebruiker de tekst kan invoeren, en een object van de zelfgemaakte klasse `DiagramView` waarin de analyse wordt getoond.
Met behulp van een `LinearLayout` worden de twee onderdelen boven elkaar gestapeld, waarbij de `EditText` door middel van `LayoutParams` een vaste hoogte van 300 pixels krijgt toebedeeld. De rest van het scherm is voor het diagram.
Aan de `EditText` wordt een event-listener gekoppeld voor het `AfterTextChanged`-event. De event-listener `veranderd` wordt dus elke keer automatisch aangeroepen nadat de gebruiker de tekst heeft veranderd. In de body van de event-listener zorgen we ervoor dat de ingetikte tekst ook beschikbaar is een speciaal hiervoor gedeclareerde variabele `Invoer` in de klasse `DiagramView`. Een aanroep van `Invalidate` (met het diagram onder handen, niet `this`, want aan een `Activity` valt niets te invalideren) zorgt ervoor dat het plaatje opnieuw getekend wordt, en de analyse van de veranderde tekst dus getoond wordt.



Figuur 24: De app TekstAnalyse in werking

Het turven van de letterfrequentie
De klasse `DiagramView` bestaat grotendeels uit de methode `OnDraw`, die het plaatje moet tekenen. Voordat deze methode de eigenlijke tekening op het canvas schildert, moet echter eerst de analyse gedaan worden. Dit gebeurt in regels 19 tot 27. In deze regels schuilt de intelligentie van het programma; de rest is eigenlijk alleen voor het verzorgen van de lay-out van de schermen.
Hoe zou je zelf de letterfrequentie bepalen? Het zou erg vermoeiend zijn om eerst de hele tekst te scannen op het aantal A's, dan nog eens voor het aantal B's, enzovoort enzovoort. Veel handiger is het om de letters A tot en met Z op een papiertje te schrijven, en dan te 'turven' hoe vaak elke letter voorkomt. Dus zoiets als in figuur 25.



Figuur 25: Een 'turftabel' voor het bepalen van de letterfrequentie

Dit is precies wat we gaan doen met behulp van een array. We maken een array van 26 getallen. Elk getal is het aantal voorkomens van een letter: het aantal A's op plaats 0, het aantal B's op plaats 1, en zo verder tot het aantal Z's op plaats 25. De declaratie van de array

```
int[] tellers;
```

en het aanmaken van het eigenlijke array-object

```
tellers = new int[26];
```

kunnen worden gecombineerd tot een declaratie-met-initialisatie:

```
int[] tellers = new int[26];
```

Hiermee hebben we een blanco turftabel, en kan het turven beginnen!
Dit gebeurt zoals je dat zelf ook zou doen als je zonder computer zou moeten turven: één voor één behandel je alle letters uit de tekst. We gebruiken daarom een `foreach`-opdracht om alle letters uit de string `Invoer` te behandelen. De letter die aan de beurt is declareren we in de header van de `foreach`-opdracht: we noemen hem `symbool`. Dus:

```
foreach(char symbool in Invoer)
```

In de body van de `foreach`-opdracht moeten we als het ware een extra turfje zetten op de juiste plaats in de turf-tabel. Dat doen we door een variabele uit de array te selecteren, en er `++` achter te schrijven: dat betekent immers 'wordt eentje meer'.
Op welke plek moet een teller verhoogd worden? Als symbool de letter `'a'` is, moet `tellers[0]` verhoogd worden. Is het de letter `'b'` is, moet `tellers[1]` verhoogd worden, en zo verder tot voor de letter `'z'` de variabele `tellers[25]`. Het gewenste plaatsnummer kunnen we uitrekenen met de expressie `symbool-'a'`. Dit heeft de waarde 0 als `symbool` gelijk is aan `'a'`, de waarde 1 voor de volgende letter `'b'`, kortom: dit is precies het plaatsnummer dat we nodig hebben. We schrijven daarom

```
tellers[symbool-'a']++;
```

Dit alles is natuurlijk alleen mogelijk als `symbool` inderdaad een letter tussen `'a'` en `'z'` is. Daarom wordt de opdracht beveiligd met een `if`-opdracht die precies dat test. Met een soortgelijk stukje code doen we nog eens hetzelfde voor de hoofdletters tussen `'A'` en `'Z'`, zodat die ook worden meegeteld:

```
if (symbool >= 'A' && symbool <= 'Z')
    tellers[symbool-'A']++;
```

Let op de enkele quotes rond de lettertekens: het gaat hier om `char`-constanten, en niet om strings. Symbolen die niet aan de voorwaarden in de `if`-opdrachten voldoen worden genegeerd: die zijn voor de analyse niet van belang.

Het bepalen van de grootste waarde in een array

De volgende stap is om te bepalen wat de hoogste letterfrequentie is. In het voorbeeld komt de letter E het vaakst voor: 14 keer. De hoogste letterfrequentie is dus 14. We hebben dit nodig zodat we het balkje voor de letter E de volledige schermbreedte kunnen gunnen. Alle letterfrequenties zitten in de array `tellers`. We zoeken dus de maximum waarde in deze array.
Hiertoe gebruiken we een variabele `max`, die dat maximum moet gaan bevatten. We laten hem klein beginnen:

```
int max = 0;
```

Daarna inspecteren we alle 26 waarden van array met behulp van een `foreach`-opdracht. Ditmaal doorloopt de `foreach`-opdracht niet de letters van de string, maar de tellers uit de turftabel.
Voor elke teller die we tegenkomen, testen we of die misschien groter is dat wat we tot nu toe dachten dat het maximum was. Als dat zo is, dan passen we het maximum aan:

```
foreach(int a in teller)
    if (a > max)
        max = a;
```

In het voorbeeld zal de variabele `max` meteen in de eerste ronde de waarde 9 krijgen, omdat de letter A negen keer voorkomt. In de volgende ronde verandert de waarde van `max` niet, omdat de B maar vijf keer voorkomt, en ook de C en de D weten de high score niet te verbeteren. Maar de teller horend bij de letter E heeft de waarde 14, en dat is groter dan 9, dus nu verandert de waarde van `max` in 14. Daarna worden ook alle overige tellers geduldig getest, maar het maximum wordt nergens meer overtroffen. Na afloop van de `foreach`-opdracht bevat `max` dus de waarde 14, de grootste waarde in de array.

Het tekenen van het staafdiagram

We doorlopen daarna nogmaals de array met tellers. Voor elk van de 26 tellers gaan we een balkje tekenen. De essentie hiervan is:

```
char letter = 'A';
foreach (int aantal in tellers)
{
    canvas.DrawRect(x, y, x+w*aantal/max, y+h, verf);
```

De deling `aantal/max` is altijd een getal tussen 0 en 1: `max` is immers het maximum van alle aantallen, dus `aantal` zal nooit groter zijn dan `max`. Vermenigvuldigd met `w`, de totaal beschikbare schermbreedte, geeft dit de breedte van het balkje. De variabele `h` bevat de hoogte die er voor elk balkje beschikbaar is: de totale schermhoogte gedeeld door 26.
Het bijschrift bij elk balkje wordt getekend met

```
    canvas.DrawText($"{letter}: {aantal}", 20, y+h-4, verf);
```

en tenslotte zorgen we ervoor dat de waardes van `y` en `letter` worden verhoogd, zodat het volgende balkje iets lager getekend wordt, en met het correcte bijschrift:

```
    y = y + h;
    letter++;
}
```

Voordat we de balkjes tekenen, trekken we met tussenruimte van 5 eenheden ook nog rode vertikale lijntjes, zodat het diagram wat gemakkelijker is te lezen. Zie listing 28 voor de details daarvan. blz. 133
Daarin is ook te vinden hoe `x`, `y`, `w` en `h` precies hun waarde krijgen.

```
using Android.App;
using Android.Widget;
using Android.OS;
using Android.Text;
using Android.Content.PM; // vanwege ScreenOrientation

namespace TekstAnalyse
{
    [ActivityAttribute(Label = "TekstAnalyse", MainLauncher = true,
                       ScreenOrientation = ScreenOrientation.Portrait)]
    public class TekstAnalyse : Activity
    {
        EditText tekst;
        DiagramView diagram;

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            tekst = new EditText(this);
            diagram = new DiagramView(this);

            tekst.AfterTextChanged += veranderd;
            tekst.TextSize = 20;

            LinearLayout.LayoutParams par
                 = new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MatchParent, 300);
            LinearLayout stapel = new LinearLayout(this);
            stapel.Orientation = Orientation.Vertical;
            stapel.AddView(tekst, par);
            stapel.AddView(diagram);
            this.SetContentView(stapel);
        }

        private void veranderd(object sender, AfterTextChangedEventArgs e)
        {
            diagram.Invoer = tekst.Text;
            diagram.Invalidate();
        }
    }
}
```

Listing 27: TekstAnalyse/TekstAnalyse.cs

```
using System;
using Android.Content;
using Android.Graphics;
using Android.Views;

namespace TekstAnalyse
{
    class DiagramView : View
    {
        public string Invoer = "";

        public DiagramView(Context c) : base(c)
        {   this.SetBackgroundColor(Color.White);
        }

        protected override void OnDraw(Canvas canvas)
        {   base.OnDraw(canvas);

            // turf alle letters in de invoer-string
            int[] tellers = new int[26];
            foreach(char symbool in Invoer)
            {
                if (symbool >= 'a' && symbool <= 'z')
                    tellers[symbool-'a']++;
                else if (symbool >= 'A' && symbool <= 'Z')
                    tellers[symbool-'A']++;
            }
            // wat is het hoogste aantal?
            int max = 0;
            foreach (int a in tellers)
            {   if (a > max)
                    max = a;
            }
            if (max < 5)
                max = 5;  // zodat het diagram aan het begin niet overdreven grote balkjes krijgt

            Paint verf = new Paint();
            int x = 100, y=0,  w = this.Width - x - 10,  h = this.Height / 26;

            // teken gridlines
            verf.Color = Color.Red;
            for (int a=0; a<max; a+=5)
            {   int d = w*a/max;
                canvas.DrawLine(x + d, 0, x + d, h * 26, verf);
            }
            // teken een balkje met bijschrift voor elk van de 26 aantallen
            char letter = 'A';
            verf.TextSize = h - 4;
            foreach (int aantal in tellers)
            {   verf.Color = Color.Black;
                canvas.DrawText($"{letter}: {aantal}", 20, y+h-4, verf);
                verf.Color = Color.Blue;
                canvas.DrawRect(x, y+1, x+w*aantal/max, y+h-2, verf);
                y += h;  letter++;
            }
        }
    }
}
```

Listing 28: TekstAnalyse/DiagramView.cs

Hoofdstuk 10

In lijsten en bewaren

10.1 ListView: een lijst op het scherm

In een `List` of in een array kun je grotere hoeveelheden gegevens opslaan. We gebruikten in sectie 6.2 een `List<PointF>` om alle aangeraakte schermposities op te slaan (zodat we op een scherm logo's konden tekenen op al die posities). En in sectie 9.4 gebruikten we een array van getallen, een variabele dus van het type `int[]`, om de frequentie van letters in een tekst te turven. Het opslaan van gegevens in een `List` of in een array kan dus meerdere doelen dienen. Maar soms wil je de inhoud van zo'n datastructuur direct aan de gebruiker laten zien. Dat kan met behulp van een `ListView`.

Voorbeeld: Kleurenlijst

In listing 29 staat het voorbeeldprogramma dat we in deze sectie bespreken. Het toont een lijst van kleur-namen aan de gebruiker. De gebruiker kan er daar interactief een aantal van aanvinken. Daarna kan de gebruiker de uitgekozen kleurnamen via sociale media delen. In figuur 26 is dit programma in werking te zien. blz. 137

ListView: tonen van een lijst aan de gebruiker

Het tonen van een lijst gaat met behulp van een `ListView`. Het aanmaken daarvan gebeurt in `OnCreate` op precies dezelfde manier als we dat gewend zijn voor andere componenten, zoals `TextView`, `Button`, `SeekBar` enzovoorts. Boven de eigenlijke `ListView` zetten we ook nog een `Button` waarmee de gebruiker uiteindelijk zijn keuze kan delen. De twee componenten worden zoals gebruikelijk gestapeld in een `LinearLayout`. Tot zover weinig nieuws. Maar hoe krijgen we de namen van de kleuren in beeld?

Adapter: bewaarplaats voor de te tonen strings

Je zou misschien verwachten dat een `ListView` een array of een list als property heeft waarin de te tonen strings zitten. Zo is het echter niet geregeld: het gebeurt via een tussenstap. Een `ListView` heeft een zogeheten *adapter*, en die bewaart de strings die in de listview getoond worden.
We hebben in ieder geval een array met de te tonen kleurnamen nodig. Met de notatie uit sectie 9.3 kunnen we (in een declaratie bovenin de klasse) de elementen direct opsommen:

```
string[] kleurNamen = { "AliceBlue", "AntiqueWhite", en nog veel meer
, "YellowGreen" };
```

Vervolgens declareren we in de methode `OnCreate` een adapter, meer precies een `ArrayAdapter<string>`:

```
ArrayAdapter<string> kleurAdapter;
```

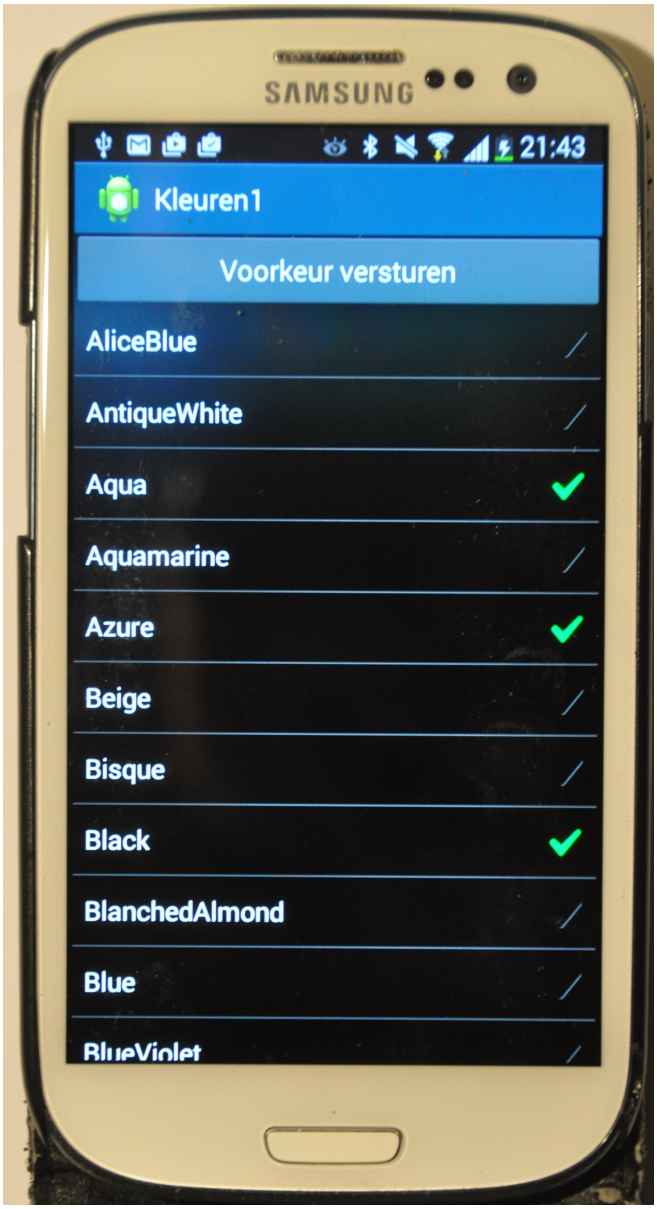
Zoals elk object moet deze objectverwijzing inderdaad naar een nieuw object gaan verwijzen:

```
kleurAdapter = new ArrayAdapter<string>( nog twee parameters , kleurNamen );
```

De derde parameter van de constructormethode is de array van strings die gebruikt gaat worden. Het adapter-object als geheel wordt aan een property van de `ListView` in beheer gegeven. Die weet daardoor wanneer hij daar behoefte aan heeft de kleurnamen wel te vinden.

```
kleurLijst.Adapter = kleurAdapter;
```

Op deze manier hebben we een listview gemaakt waarin de kleurnamen zichtbaar worden.



Figuur 26: De app KleurenApp1 in werking

```
using System;           // vanwege EventArgs
using Android.App;      // vanwege Activity
using Android.Content;  // vanwege Intent
using Android.Widget;   // vanwege ListView, ArrayAdapter, ChoiceMode, Button enz.
using Android.OS;       // vanwege Bundle
using Android.Util;     // vanwege SparseBooleanArray

namespace KleurLijst1
{
    [ActivityAttribute(Label = "Kleuren1", MainLauncher = true)]
    public class KleurenApp1 : Activity
    {
        ListView kleurLijst;
        string[] kleurNamen = { "AliceBlue", "AntiqueWhite", "Aqua", "Aquamarine", "Azure", "Beige",
                              , "Wheat", "White", "WhiteSmoke", "Yellow", "YellowGreen" };

        protected override void OnCreate(Bundle bundle)
        {   base.OnCreate(bundle);

            ArrayAdapter<string> kleurAdapter
                = new ArrayAdapter<string>
                  (this, Android.Resource.Layout.SimpleListItemChecked, kleurNamen);
            // probeer ook eens: SimpleListItemMultipleChoice en SimpleListItemActivated1
            kleurLijst = new ListView(this);
            kleurLijst.Adapter = kleurAdapter;
            kleurLijst.ChoiceMode = ChoiceMode.Multiple;
            kleurLijst.ItemClick += itemklik;
            kleurLijst.SetItemChecked(2, true);
            kleurLijst.SetItemChecked(4, true);
            kleurLijst.SetItemChecked(7, true);

            Button versturen = new Button(this);
            versturen.Text = "Voorkeur versturen";
            versturen.Click += verstuurKlik;

            LinearLayout stapel = new LinearLayout(this);
            stapel.Orientation = Orientation.Vertical;
            stapel.AddView(versturen);
            stapel.AddView(kleurLijst);
            this.SetContentView(stapel);
        }
        private void verstuurKlik(object sender, EventArgs e)
        {   string bericht = "Dit vind ik mooie kleuren:\n";
            SparseBooleanArray a = kleurLijst.CheckedItemPositions;
            for (int k = 0; k < a.Size(); k++)
                if (a.ValueAt(k))
                    bericht += $"{kleurNamen[a.KeyAt(k)]}\n";
            Intent i = new Intent(Intent.ActionSend);
            i.SetType("text/plain");
            i.PutExtra(Intent.ExtraText, bericht);
            this.StartActivity(i);
        }
        private void itemklik(object sender, AdapterView.ItemClickEventArgs e)
        {   string t = kleurNamen[e.Position];
            Toast.MakeText(this, t, Android.Widget.ToastLength.Short).Show();
        }
    }
}
```

Listing 29: KleurLijst1/KleurenApp1.cs

ChoiceMode: selecteerbare lijstelementen
Om de gebruiker de kleurnamen ook te kunnen laten aanvinken is nog maar één extra opdracht nodig:

```
kleurLijst.ChoiceMode = ChoiceMode.Multiple;
```

Er zijn drie mogelijke `ChoiceMode`-waarden: `None` (als de lijstelementen niet aanvinkbaar zijn), `Single` (als de gebruiker één element mag aanvinken) of `Multiple` (als de gebruiker meerdere elementen mag aanvinken).
We gebruiken in dit programma `Multiple`. Nu wordt ook van belang wat we als tweede parameter hebben meegegeven bij de constructie van de adapter. Daar wordt namelijk bepaald hoe het aanvinken gevisualiseerd wordt. We gebruiken klassieke groene vinkjes:

```
kleurAdapter = new ArrayAdapter<string>(... , Android.Resource.Layout.SimpleListItemChecked, ...);
```

Alternatieven zijn: `SimpleListItemMultipleChoice` voor een meer checkbox-achtige layout, en `SimpleListItemActivated1` als de hele tekstregel moet oplichten. Voor `Single`-keuzelijsten is `SimpleListItemSingleChoice` geschikter: dat geeft een radiobutton-achtige layout. En voor niet-selecteerbare lijsten gebruik je bijvoorbeeld `SimpleListItem1`.

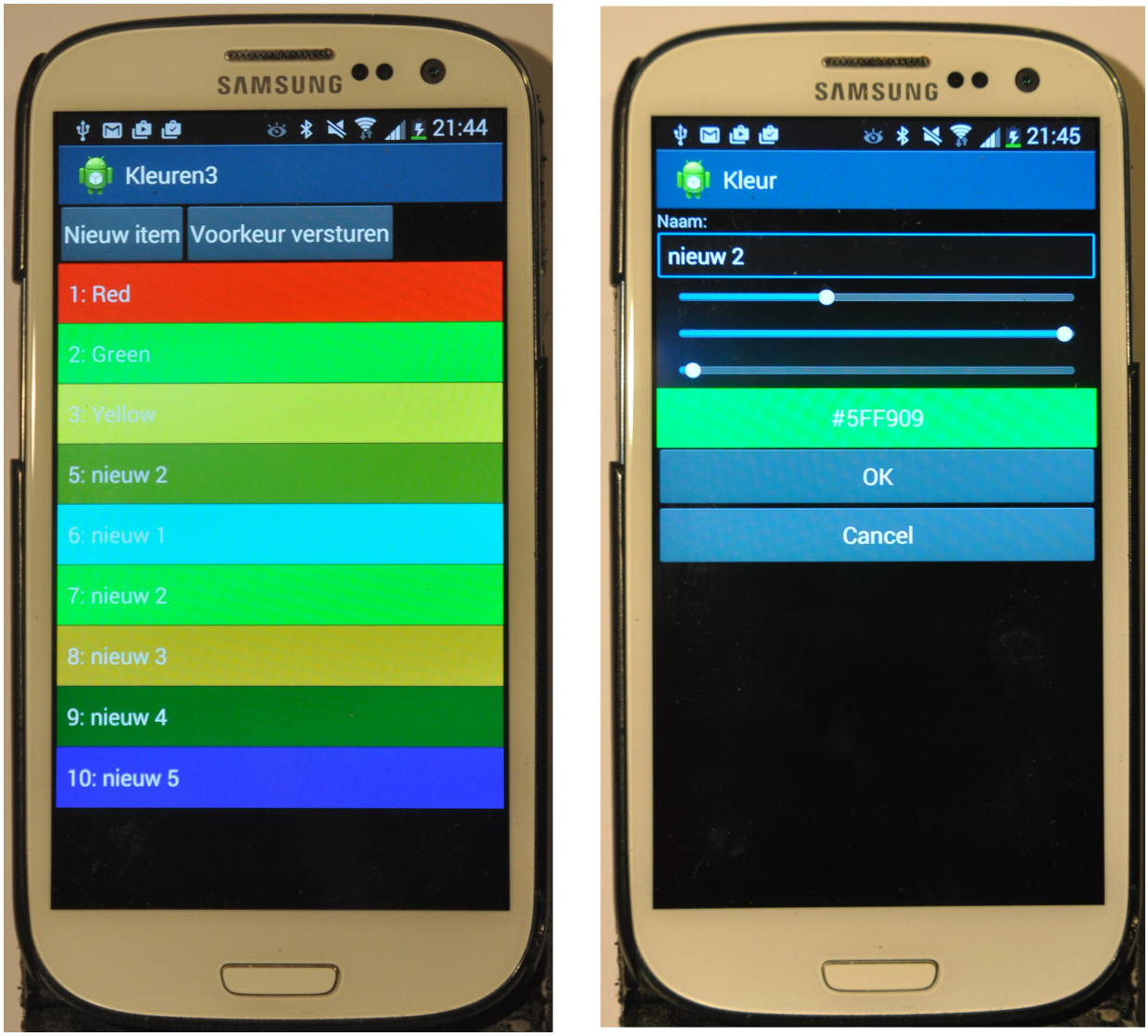
Aanklikken van lijstelementen
Bij het aanklikken van de lijstelementen in een listview waarvan de `ChoiceMode` niet `None` is worden de vinkjes automatisch geplaatst. Wil je (ook nog) iets anders doen, dan kun je een event-handler
blz. 137 aan `ItemClicked` koppelen. De parameter van de event-handler (zie listing 29) is een object waarin het nummer van de aangeklikte regel beschikbaar is in de property `Position`. Die gebruiken we hier om de naam uit onze eigen array met kleurnamen op te halen. Die naam wordt vervolgens twee seconden lang aan de gebruiker getoond in een `Toast`. Doordat zo'n toast-berichtje na een paar seconden automatisch verdwijnt is dit een voor de gebruiker wat minder hinderlijke manier van berichten dan de `AlertDialog` die we in sectie 8.2 gebruikten.

Opvragen van gekozen lijstelementen
De methode `verstuurKlik` tenslotte, is de event-handler van de knop waarmee we de keuze gaan versturen naar een van onze sociale media. Het laat zien hoe je kunt opvragen welke lijst-elementen zijn aangevinkt. Je inspecteert daartoe de property `CheckedItemPositions` van de listview. Dat levert een object op, waarvan je voor alle getallen tot zijn `Count` kunt opvragen wat de *key* is (dat is het volgnummer van een lijstelement) en wat daarbij de `value` is (een bool die true is als het element is aangevinkt).
blz. 137 Als voorbeeld bouwen we in regel 45–47 van listing 29 een berichtje op waarin de namen van uitsluitend de aangevinkte elementen een rol spelen, dat vervolgens met behulp van een `Intent` de wereld in gestuurd wordt (zoals we al bespraken in sectie 8.3).

10.2 Een eigen Adapter

In deze sectie passen we het voorbeeldprogramma aan, zo dat nu niet alleen de namen van de kleuren getoond worden, maar ook de kleur zelf zichtbaar gemaakt wordt. De Activity-subklasse
blz. 142 staat in listing 30 en listing 31. In dit geval is er een eigen klasse voor de adapter nodig. Die staat
blz. 143 in listing 32. En tenslotte hebben we een klasse (in listing 33) waarin een object wordt beschreven
blz. 144 die de relevante informatie van elke kleur bevat: de naam, en de bijbehorende RGB-waarde. In
blz. 145 figuur 27 zijn het hoofdscherm en het kleuren-editscherm van dit programma in werking te zien.

Een klasse voor een eigen type object
In de vorige versie van het programma ging het eigenlijk alleen maar om de namen van de kleuren. De listview visualiseerde simpelweg een array van strings. Ook in deze versie zijn de namen van belang, maar ook de bijbehorende kleur. De gegevens die in de listview worden getoond, zijn dus steeds een combinatie van een naam en een `Color`. We gaan een lijst van objecten maken, waarvan elk element zo'n naam-kleur-combinatie bevat. Omdat er geen standaardtype is waarin een naam-kleur-combinatie opgeslagen kan worden, moeten we dat type zelf maken.
Dit doen we op de manier die beschreven staat in sectie 9.1: een klasse is het type van een object, dus als we een nieuw type object willen beschrijven, moeten we een klasse definiëren. In dit geval is dat de klasse `KleurItem`.
Het belangrijkste van deze klasse zijn de twee declaraties van de onderdelen van het object: een string die de naam bevat, en de bijbehorende kleur:



Figuur 27: De app KleurenApp2 in werking

```
class KleurItem
{
    public string Naam;
    public Color Kleur;
```

Om het aanmaken van `KleurItem`-objecten gemakkelijker te maken, doen we er ook een constructormethode bij. Het is de fantasieloze variant, die net zo veel parameters heeft als er declaraties in de klasse staan:

```
    public KleurItem(string n, Color k)
    {
        this.Naam = n;
        this.Kleur = k;
    }
}
```

blz. 145 De complete klasse staat in listing 33. Deze is nog iets uitgebreider, en ook iets anders dan hier beschreven is, maar dat is vooral vanwege het gebruik dat we er in sectie 10.4 nog van gaan maken.

KleurItem-objecten in plaats van strings

In de vorige versie van het programma gebruikten we een array van strings met namen van kleuren:

```
string[] kleurNamen = { "AliceBlue", "AntiqueWhite", "Aqua", "Aquamarine", "Azure", "Beige", "Bisque", "B
```

In deze versie vervangen we die door een array van naam-kleur-combinaties, oftewel van `KleurItem`-objecten:

```
KleurItem[] defaultKleuren = { new KleurItem("Rood",  Color.Red)
                             , new KleurItem("Groen", Color.Green)
                             , new KleurItem("Geel",  Color.Yellow)
                             };
```

De spreiding over meerdere regels is alleen om de source code een beetje overzichtelijk te houden; strikt noodzakelijk is dit niet.

List in plaats van array

In de vorige versie van het programma gaven we de array met strings in beheer aan een *adapter*. We zouden dat nu ook kunnen doen met onze array van `KleurItem`s. Dat gaat goed zolang we de items in de listview alleen willen bekijken. Maar in sectie 10.3 gaan we ook items tussenvoegen en verwijderen. In een array is het niet mogelijk om later nog elementen tussen te voegen. In een `List` kan dat wel. Daarom gaan we nu een `List<KleurItem>` gebruiken in plaats van een `KleurItem[]`. Dat kan heel gemakkelijk, want een van de constructormethodes van `List` kan worden aangeroepen met een array als parameter, mits die array hetzelfde type elementen heeft als de lijst:

```
List<KleurItem> kleuren = new List<KleurItem>(defaultKleuren);
```

Het blijft zinvol om ook de array te declareren, want die kun je gemakkelijk initialiseren door de elementen op te sommen. Dat kan met een `List` dan weer niet.

blz. 145 In listing 33 staat de declaratie van `kleuren` boven in de klasse. De initialisatie staat in de éénregelige methode `BeginKleuren`, die op zijn beurt wordt aangeroepen in de constructormethode van `KleurenApp2`. Dat heeft uiteindelijk hetzelfde effect als de declaratie-met-initialisatie hierboven. De reden dat we declaratie en initialisatie uit elkaar hebben getrokken blijkt in sectie 10.4.

Een eigen adapter-klasse

In de vorige versie van het programma gebruikten we een `ArrayAdapter<string>`. Dat is een standaardklasse, die de visualisatie van een array van strings voor zijn rekening neemt. Door een object van dit type te gebruiken als waarde van de `Adapter`-eigenschap van de `ListView` gebeurt dat helemaal automatisch.

In deze versie willen we de visualisatie echter zelf doen. We moeten daarom een eigen adapterklasse maken: `KleurenAdapter`. Als die er eenmaal is, is het niet moeilijk meer: we vervangen de regels

```
ArrayAdapter<string> kleurAdapter;
kleurAdapter = new ArrayAdapter<string>(this, Android.Resource.Layout.SimpleListItemChecked, kleurNamen);
kleurLijst.Adapter = kleurAdapter;
```

door

```
KleurenAdapter kleurAdapter;
kleurAdapter = new KleurenAdapter(this, kleuren);
kleurLijst.Adapter = kleurAdapter;
```

De twee toenenningsopdrachten staan niet direct in de constructormethode, maar in een methode `LeesKleuren`, die vanuit de constructormethode van `KleurenApp2` wordt aangeroepen. De reden voor deze aparte methode blijkt in de volgende sectie.
Maar eerst moeten we dus wel zo'n klasse `KleurenAdapter` hebben. Deze staat in listing 32. blz. 144
Het is een subklasse van `BaseAdapter`, een standaardklasse die bedoeld is om als uitgangspunt te gebruiken bij het definiëren van adapters. Als variabelen heeft de klasse de lijst met items, en ook de activity waar hij deel van uitmaakt (die is namelijk nodig als je nieuwe `View`-objecten wilt maken, en dat is precies de taak van een adapter). We maken een constructormethode die de gewenste waarden van beide variabelen meekrijgt, en deze opslaat voor later gebruik:

```
public class KleurenAdapter : BaseAdapter<KleurItem>
{
    IList<KleurItem> items;
    Activity context;
    public KleurenAdapter(Activity context, IList<KleurItem> items)
    {
        this.items = items;
        this.context = context;
    }
```

Het belangrijkste onderdeel van onze adapter-klasse is de methode `GetView`. Het is de taak van deze methode om, gegeven een positie in de lijst, een `View` op te leveren waarin dit element gevisualiseerd wordt.
In essentie krijgt deze methode een `int` als parameter waarmee de positie wordt aangeduid. In de body maken we een `TextView` aan, waarvan we de attributen naar smaak aanpassen. Met behulp van de positie kunnen we het betreffende `KleurItem`-object uit de lijst ophalen om de tekst en de kleur van de nieuwe `TextView` te bepalen.

```
    public override View GetView(int position)
    {
        TextView view = new TextView(context);
        view.TextSize = 30;
        view.SetHeight(100);
        KleurItem item = items[position];
        view.Text = $"{item.Id}: {item.Naam}";
        view.SetBackgroundColor(item.Kleur);
        return view;
    }
```

Deze methode wordt door de `ListView` steeds opnieuw aangeroepen als er een lijstelement in beeld komt. Als de gebruiker heen en weer scrollt door een lange lijst kan dat heel vaak gebeuren, en steeds opnieuw worden er dan `TextView`-objecten aangemaakt. Omdat de geheugenruimte daarmee snel zou opraken, is de methode in werkelijkheid iets ingewikkelder. Hij krijgt als parameter ook nog een `View`-object, waarin een reeds eerder aangemaakt object beschikbaar kan zijn dat niet meer nodig is. Als deze parameter niet `null` is, kunnen we die opnieuw gebruiken, en daarmee het aanmaken van een nieuw `TextView` object uitsparen. Zie listing 32 voor de details. blz. 144

```
using System;           // vanwege EventArgs
using System.Collections.Generic; // vanwege List
using Android.App;      // vanwege Activity
using Android.Content;  // vanwege Intent
using Android.Widget;   // vanwege ListView, ArrayAdapter, ChoiceMode, Button enz.
using Android.OS;       // vanwege Bundle
using Android.Graphics; // vanwege Color;
//using System.IO;      // vanwege File (nu nog niet nodig, maar in KleurenApp3 wel)
//using SQLite;         // vanwege SQLiteConnection (nu nog niet nodig, maar in KleurenApp3 wel)

namespace KleurLijst2
{
    [ActivityAttribute(Label = "Kleuren2", MainLauncher = false)]
    public class KleurenApp2 : Activity
    {
        protected ListView kleurLijst;
        protected List<KleurItem> kleuren;
        protected KleurItem[] defaultKleuren = { new KleurItem("Rood", Color.Red), new KleurItem("Gr
        protected KleurenAdapter kleurAdapter;

        protected override void OnCreate(Bundle bundle)
        {   base.OnCreate(bundle);
            this.BeginKleuren();
            kleurLijst = new ListView(this);
            kleurLijst.ChoiceMode = ChoiceMode.None;
            kleurLijst.ItemClick += itemKlik;
            this.LeesKleuren();

            Button nieuw = new Button(this);
            nieuw.Text = "Nieuw item";
            nieuw.Click += nieuwItem;
            Button versturen = new Button(this);
            versturen.Text = "Voorkeur versturen";
            versturen.Click += verstuurKlik;

            LinearLayout rij = new LinearLayout(this);
            rij.Orientation = Orientation.Horizontal;
            rij.AddView(nieuw);
            rij.AddView(versturen);

            LinearLayout stapel = new LinearLayout(this);
            stapel.Orientation = Orientation.Vertical;
            stapel.AddView(rij);
            stapel.AddView(kleurLijst);
            this.SetContentView(stapel);
        }
        private void verstuurKlik(object sender, EventArgs e)
        {   string bericht = "<html><body><table>\n";
            foreach (KleurItem item in kleuren)
            {   bericht += $"<tr bgcolor=#{item.R:X2}{item.G:X2}{item.B:X2}>";
                bericht += $"<td>{item.Naam}</td></tr>\n";
            }
            bericht += "</table></body></html>\n";
            Intent i = new Intent(Intent.ActionSend);
            i.SetType("text/html");
            i.PutExtra(Intent.ExtraText, bericht);
            this.StartActivity(i);
        }
```

Listing 30: KleurLijst2/KleurenApp2.cs, deel 1 van 2

```
        private void nieuwItem(object sender, EventArgs e)
        {
            Intent i = new Intent(this, typeof(KleurEdit));
            this.StartActivityForResult(i, 1000000);
        }

        private void itemKlik(object sender, AdapterView.ItemClickEventArgs e)
        {
            int pos = e.Position;
            Intent i = new Intent(this, typeof(KleurEdit));
            KleurItem item = kleurAdapter[pos];
            i.PutExtra("naam", item.Naam);
            i.PutExtra("kleur", item.Kleur.ToArgb());
            this.StartActivityForResult(i, pos);
        }

        protected virtual void BeginKleuren()
        {
            kleuren = new List<KleurItem>(defaultKleuren);
        }

        protected virtual void LeesKleuren()
        {
            kleurAdapter = new KleurenAdapter(this, kleuren);
            kleurLijst.Adapter = kleurAdapter;
        }

        protected override void OnActivityResult(int pos, Result res, Intent data)
        {
            if (res==Result.Ok)
            {
                string naam = data.GetStringExtra("naam");
                Color kleur = new Color(data.GetIntExtra("kleur", 0));
                if (pos == 1000000)
                    kleuren.Add(new KleurItem(naam, kleur));
                else
                    kleuren[pos] = new KleurItem(naam, kleur);
            }
            else
            {
                if (pos < 1000000)
                    kleuren.RemoveAt(pos);
            }
            this.LeesKleuren();
        }
    }
}
```

Listing 31: KleurLijst2/KleurenApp2.cs, deel 2 van 2

```
using System.Collections.Generic;
using Android.App;
using Android.Views;
using Android.Widget;

namespace KleurLijst2
{
    public class KleurenAdapter : BaseAdapter<KleurItem>
    {
        Activity context;
        IList<KleurItem> items;

        public KleurenAdapter(Activity context, IList<KleurItem> items)
        {
            this.context = context;
            this.items = items;
        }

        public override long GetItemId(int position)
        {
            return position;
        }

        public override KleurItem this[int position]
        {
            get { return items[position]; }
        }

        public override int Count
        {
            get { return items.Count; }
        }

        public override View GetView(int position, View hergebruik, ViewGroup root)
        {
            TextView view = (TextView)hergebruik;
            if (view==null)
                view = new TextView(context);
            view.TextSize = 30;
            view.SetHeight(100);
            KleurItem item = items[position];
            view.Text = $"{item.Id}: {item.Naam}";
            view.SetBackgroundColor(item.Kleur);

            return view;
        }
        /*  alternatieve versie:
        public override View GetView(int position, View hergebruik, ViewGroup root)
        {   TextView view = (TextView)(hergebruik
                                    ?? context.LayoutInflater.Inflate
                                         (Android.Resource.Layout.SimpleListItem1, null)
                                    );
            KleurItem item = items[position];
            view.Text = $"{item.Id}: {item.Naam}";
            view.SetBackgroundColor(item.Kleur);
            return view;
        }
        */
    }
}
```

```
using Android.Graphics;
using SQLite;

namespace KleurLijst2
{
    public class KleurItem
    {
        [PrimaryKey, AutoIncrement]
        public int Id { get; set; }

        public string Naam { get; set; }
        public int R { get; set; }
        public int G { get; set; }
        public int B { get; set; }

        public KleurItem()
        {
        }

        public KleurItem(string naam, Color kleur)
        {
            this.Naam = naam;
            this.R = kleur.R;
            this.G = kleur.G;
            this.B = kleur.B;
        }
        public Color Kleur
        {
            get { return new Color(R, G, B); }
        }
    }
}
```

Listing 33: KleurLijst2/KleurItem.cs

10.3 Interactief toevoegen en verwijderen

Als de gebruiker in deze versie op een lijstelement klikt, wordt er een apart scherm gelanceerd waarin de gebruiker de naam en/of de kleur kan aanpassen. De activity die dit mogelijk maakt (in listing 34 en listing 35) is grotendeels geïnspireerd door de kleurenmixer die we in sectie 3.2 al hadden geschreven.

blz. 147
blz. 148

Lanceren van een nieuwe activiteit

Het scherm waarmee een kleur (èn de naam daarvan!) kan worden aangepast wordt op twee plaatsen in dit programma gelanceerd:

- als de gebruiker op een element van de listview klikt, om een bestaande kleur te veranderen (in de event-handler `itemKlik`)
- als de gebruiker op de knop 'nieuw item' klikt, om een nieuwe kleur toe te voegen (in de event-handler `nieuwItem`)

In beide gevallen maken we een `Intent` aan met de benodigde gegevens, die we daarna meegeven aan `StartActivityForResult`.

De gewijzigde kleur en/of naam worden weer teruggestuurd als het kleuren-editscherm klaar is. Daarom gebruiken we `StartActivityForResult` in plaats van de gewone `StartActivity`. Om het antwoord op te vangen is er ook een methode `OnActivityResult`, precies zoals we in sectie 8.3 al hebben gezien. Daar zagen we dat je bij het lanceren een code kunt meegeven, die je bij `OnActivityResult` weer terugkrijgt. Die zetten we hier slim in: we gebruiken het volgnummer van het lijstitem als code, zodat in `OnActivityResult` ook weer duidelijk is welk item het betrof. Alleen in het geval van een nieuwe kleur gebruiken we een speciale code: 1 miljoen (zo lang zal de lijst hopelijk nooit worden, anders gaat dit fout. . .).

Verwerken van het resultaat van een activiteit

In de methode `OnActivityResult` onderscheiden we de verschillende situaties: als de kleuren-editor werd afgesloten met 'Ok', vervangen we een bestaande kleur, of voegen een nieuwe kleur toe. Anders werd de kleuren-editor blijkbaar afgesloten met 'cancel', en dan verwijderen we een bestaande kleur (en een nieuwe kleur hoeft toch maar niet te worden toegevoegd). Tenslotte doen we een aanroep van `LeesKleuren`, zodat de mogelijk gewijzigde lijst ook weer wordt gebruikt in de adapter van de listview. Nu komt het dus goed uit dat de betreffende opdrachten in een aparte methode staan!

Een activity om de kleur aan te passen

Voor elk data-invoerscherm in een app moet je een aparte activity maken, dus ook voor het kleuren-editscherm. Deze activity (in listing 34 en listing 35) is vrijwel hetzelfde als het programma uit sectie 3.2, compleet met knop om een random kleur te genereren.

blz. 147
blz. 148

Hier is er toegevoegd dat in `OnCreate` de start-kleur en -naam worden uitgepakt uit de `Intent`, en in de event-handlers van de 'ok'- en 'cancel'-knop weer worden ingepakt. Als er geen naam is gespecificeerd gaat het blijkbaar om de situatie dat er een nieuw item wordt toegevoegd. In dit geval roepen we de event-handler van de random-kleur-knop alvast aan, zodat er een random kleur klaar staat. Omdat een event-handler nu eenmaal twee parameters vraagt, geven we `null` en `null` mee; we gebruiken deze parameters toch niet in de body van de event-handler. Iets soortgelijks gebeurt bij de aanroep van de event-handler `veranderd` die er voor zorgt dat de RGB-waarde van de kleur als opschrift op de knop komt te staan.

```
using System;
using Android.App;
using Android.Content;
using Android.OS;
using Android.Widget;
using Android.Graphics;

namespace KleurLijst2
{
    [ActivityAttribute(Label = "Kleur", MainLauncher = false)]
    class KleurEdit : Activity
    {
        Color kleur;
        EditText naamVeld;
        SeekBar rood, groen, blauw;
        Button huidig;
        static int volgnummer = 1;

        protected override void OnCreate(Bundle b)
        {
            base.OnCreate(b);

            string naam = this.Intent.GetStringExtra("naam");
            kleur = new Color(this.Intent.GetIntExtra("kleur", 0));

            rood  = new SeekBar(this); rood. Max = 255; rood. Progress = kleur.R; rood. SetBackgroun
            groen = new SeekBar(this); groen.Max = 255; groen.Progress = kleur.G; groen.SetBackgroun
            blauw = new SeekBar(this); blauw.Max = 255; blauw.Progress = kleur.B; blauw.SetBackgroun

            LinearLayout stapel = new LinearLayout(this);
            TextView naamLabel = new TextView(this); naamLabel.Text = "Naam:";
            Button okButton = new Button(this); okButton.Text = "OK";
            Button cancelButton = new Button(this); cancelButton.Text = "Cancel";

            if (naam == null)
            {
                naam = $"nieuw {volgnummer}"; volgnummer++;
                this.willekeurig(null, null);
            }
            this.naamVeld = new EditText(this); naamVeld.Text = naam;
            this.huidig = new Button(this);
            this.veranderd(null, null);

            rood .ProgressChanged += veranderd;
            groen.ProgressChanged += veranderd;
            blauw.ProgressChanged += veranderd;
            huidig.Click       += willekeurig;
            okButton.Click     += ok;
            cancelButton.Click += cancel;

            stapel.Orientation = Orientation.Vertical;
            stapel.AddView(naamLabel);
            stapel.AddView(naamVeld);
            stapel.AddView(rood);
            stapel.AddView(groen);
            stapel.AddView(blauw);
            stapel.AddView(huidig);
            stapel.AddView(okButton);
            stapel.AddView(cancelButton);
            this.SetContentView(stapel);
        }
```

```
        private void willekeurig(object sender, EventArgs e)
        {
            Random generator = new Random();
            rood. Progress = generator.Next(255);
            groen.Progress = generator.Next(255);
            blauw.Progress = generator.Next(255);
        }

        private void veranderd(object sender, SeekBar.ProgressChangedEventArgs e)
        {
            int r = rood. Progress;
            int g = groen.Progress;
            int b = blauw.Progress;
            huidig.Text = $"#{r:X2}{g:X2}{b:X2}";
            this.kleur = new Color(r, g, b);
            huidig.SetBackgroundColor(kleur);
        }

        private void cancel(object sender, EventArgs e)
        {
            Intent i = new Intent();
            this.SetResult(Result.Canceled, i);
            this.Finish();
        }

        private void ok(object sender, EventArgs e)
        {
            Intent i = new Intent();
            i.PutExtra("naam", naamVeld.Text);
            i.PutExtra("kleur", kleur.ToArgb());
            this.SetResult(Result.Ok, i);
            this.Finish();
        }
    }
}
```

Listing 35: KleurLijst2/KleurEdit.cs, deel 2 van 2

10.4 Data bewaren in een database

Een laatste variant is waarin de wijzigingen die de gebruiker aanbrengt in een database worden bewaard, zodat als de app later nog eens wordt gebruikt, de gewijzigde kleuren nog behouden zijn. De eerste helft (met de methoden `OnCreate` en `verstuurKlik`) van deze activity is (op de naam van de klasse na) precies hetzelfde als in listing 30. Het tweede deel (met de methoden `BeginKleuren`, `LeesKleuren`, `itemKlik` en `OnActivityResult`) is anders, en staat in listing 36. blz. 142 blz. 150

SQLite

Een gemakkelijke manier om een database te maken is met behulp van de library `SQLite`. SQL (*Structured Query Language*) is een veel gebruikte taal om vragen (*queries*) te kunnen stellen aan een database. Het achtervoegsel 'lite' in de naam van de library geeft aan dat niet de hele taal beschikbaar is. Toch is de library goed genoeg voor het soort van databases dat je in een app typisch zult gebruiken. Je kunt queries formuleren in de taal SQL, en die als string meegeven aan bepaalde methoden in de library. Maar er zijn ook methoden aanwezig waarmee je queries kunt doen, zonder dat je zelf noemenswaardig SQL hoeft te kennen.

Installeren van SQLite

Om de SQLite library te gebruiken moet je in je programma de regel

```
using SQLite
```

opnemen. Maar omdat dit geen standaard-library is, is dit nog niet genoeg. Je moet ook de library zelf downloaden op github. Je krijgt dan een bestand van 3500 regels source code, die je gewoon in je project er bij kunt zetten. Het bestand zit ook in de zip-file met alle voorbeeldprogramma's uit dit diktaat.

Aanmaken van een database

Een SQLite-database bevindt zich in zijn geheel in één bestand. Je kunt de database gebruiken door middel van een object van het type `SQLiteConnection`. Bij het aanmaken van dit object geef je de naam van het bestand mee, en daarna kun je de database meteen gebruiken. Dat kan door het object onder handen te nemen met de volgende methoden:

- `CreateTable`, om een nieuwe tabel aan te maken
- `Table`, om een bestaande tabel uit te lezen
- `Insert`, om een record toe te voegen aan een tabel
- `Update`, om een record in een tabel te wijzigen
- `Delete`, om een record uit een tabel te verwijderen

Het bedenken waar het bestand moet staan is misschien nog wel het lastigste aspect van het aanmaken van de database. In regel 82 en 83 van de listing 36 staat hoe je in Android aan een geschikte locatie voor data-bestanden kunt komen. Bestaat dit bestand nog niet, dan maken we een nieuwe aan, en zetten daar meteen alvast een paar standaardkleuren in. Dit zal dus alleen de eerste keer dat we de app runnen gebeuren. Bij een volgende gelegenheid bestaat de database al, en hoeft deze niet opnieuw met standaardkleuren gevuld te worden. blz. 150

Tabellen in een SQLite-database

In een SQLite-database kunnen, zoals gebruikelijk in een SQL-database, verschillende tabellen worden gebruikt. In ons programma hebben we echter maar één tabel nodig, gevuld met `KleurItem`-objecten. De tabellen worden onderscheiden door hun type. De vijf hierboven genoemde methoden zijn *generiek*, dat wil zeggen het type waarop ze betrekking hebben moet er tussen punthaken achter worden geschreven. Aan dat type heeft SQLite genoeg om te weten welke tabel uit de databae aangesproken wordt.

De parameters van de methoden hangen ook af van dat type. Zo heeft de methode `Insert<KleurItem>` een parameter van het type `KleurItem`. Voluit luidt de header van deze methode dus:

```
public void Insert<KleurItem> (KleurItem k)
```

Deze methode gebruiken we om de database te vullen met de kleuritems uit de array met default-kleuren. Voor die tijd hebben we, als de database nog niet bestond, de parameterloze methode `CreateTable<KleurItem>()` aangeroepen om de tabel aan ta maken.

```
        // Dit is de klasse KleurenApp3
        // Tot hier was het hetzelfde als KleurenApp2. Maar de rest is anders:

        SQLiteConnection database;

        protected void BeginKleuren()
        {   string docsFolder = System.Environment.GetFolderPath
                                    (System.Environment.SpecialFolder.MyDocuments);
            string pad = System.IO.Path.Combine(docsFolder, "kleuren.db");
            bool bestaat = File.Exists(pad);
            database = new SQLiteConnection(pad);
            if (!bestaat)
            {   database.CreateTable<KleurItem>();
                foreach (KleurItem k in defaultKleuren)
                    database.Insert(k);
            }
        }

        protected void LeesKleuren()
        {   kleuren = new List<KleurItem>();
            TableQuery<KleurItem> query = database.Table<KleurItem>();
            foreach (KleurItem k in query)
                kleuren.Add(k);
            kleurAdapter = new KleurenAdapter(this, kleuren);
            kleurLijst.Adapter = kleurAdapter;
        }

        protected override void OnActivityResult(int pos, Result res, Intent data)
        {
            if (res == Result.Ok)
            {   string naam = data.GetStringExtra("naam");
                Color kleur = new Color(data.GetIntExtra("kleur", 0));
                if (pos == 1000000)
                    database.Insert(new KleurItem(naam, kleur));
                else
                {   KleurItem k = new KleurItem(naam, kleur);
                    k.Id = kleuren[pos].Id;
                    database.Update(k);
                }
            }
            else
            {   if (pos < 1000000)
                {   KleurItem k = new KleurItem();
                    k.Id = kleuren[pos].Id;
                    database.Delete(k);
                }
            }
            this.LeesKleuren();
        }
    }
}
```

Listing 36: KleurLijst2/KleurenApp3.cs, deel 2 van 2

```
using System.Collections.Generic; // vanwege List
using Android.App;      // vanwege Activity
using Android.Content;  // vanwege Intent
using Android.Graphics; // vanwege Color;
using System.IO;        // vanwege File
using SQLite;           // vanwege SQLiteConnection
using System;

namespace KleurLijst2
{
    [ActivityAttribute(Label = "Kleuren4", MainLauncher = true)]
    public class KleurenApp4 : KleurenApp2
    {
        SQLiteConnection database;

        protected override void BeginKleuren()
        {   string docsFolder = System.Environment.GetFolderPath
                                  (System.Environment.SpecialFolder.MyDocuments);
            string pad = System.IO.Path.Combine(docsFolder, "kleuren.db");
            bool bestaat = File.Exists(pad);
            database = new SQLiteConnection(pad);
            if (!bestaat)
            {   database.CreateTable<KleurItem>();
                foreach (KleurItem k in defaultKleuren)
                    database.Insert(k);
            }
        }

        protected override void LeesKleuren()
        {   kleuren = new List<KleurItem>();
            TableQuery<KleurItem> query = database.Table<KleurItem>();
            foreach (KleurItem k in query)
                kleuren.Add(k);
            base.LeesKleuren();
        }

        protected override void OnActivityResult(int id, Result res, Intent data)
        {   if (res == Result.Ok)
            {   string naam = data.GetStringExtra("naam");
                Color kleur = new Color(data.GetIntExtra("kleur", 0));
                if (id == 1000000)
                    database.Insert(new KleurItem(naam, kleur));
                else
                {   KleurItem k = new KleurItem(naam, kleur);
                    k.Id = id;
                    database.Update(k);
                }
            }
            else
            {   if (id < 1000000)
                {   KleurItem k = new KleurItem();
                    k.Id = id;
                    database.Delete(k);
                }
            }
            this.LeesKleuren();
        }
    }
}
```

Listing 37. KleurLijst2/KleurenApp4.cs

Types voor SQLite-tabellen

Niet zomaar elk type kan worden gebruikt als type van een tabel in een SQLite-database. Ons type `KleurItem` in listing 33 voldoet aan de vereisten: Die zijn:

blz. 145

- Bij elke member-variabele moet expliciet worden aangegeven dat deze opgevraagd en veranderd kan worden, met `{get; set;}` in de declaratie.
- Er moet een member-variabele zijn die gebruikt kan worden als *key* in de database, en die moet gemarkeerd zijn met het attribuut `[PrimaryKey]`.
- Er moet een constructorfunctie zonder parameters zijn.
- Alle member-variabelen moeten van basistypes zijn (`int`, `double`, `string` enz.), of van enkele andere types die worden herkend door SQLite (bijvoorbeeld `DateTime`). Het type `Color` wordt niet herkend, dus dat is taboe. Daarom hebben we aparte variabelen `R`, `G` en `B` gedeclareerd.

Queries op een SQLite-tabel

De belangrijkste query die je kunt doen op een database is het opvragen van de inhoud. Dat gebeurt door een aanroep van `Table`:

```
TableQuery<KleurItem> query = database.Table<KleurItem>();
```

Als je wilt kun je de resulterende waarde (van het type `TableQuery`) nog nader onder handen nemen met de methode `Where`, om alleen de records die aan een bepaalde voorwaarde voldoen te selecteren. In ons programma doen we dat niet, en vraagt de query dus om alle records.
Het leuke is dat je een `TableQuery` kunt gebruiken in een `foreach` opdracht. Hiermee worden alle resultaten van de query achtereenvolgens verwerkt. In ons programma voegen we ze toe aan een `List`:

```
foreach (KleurItem k in query)
    kleuren.Add(k);
```

zodat we die net zoals in de vorige versie van het programma in een adapter kunnen gebruiken:

```
kleurAdapter = new KleurenAdapter(this, kleuren);
kleurLijst.Adapter = kleurAdapter;
```

Aanpassen van de kleuren in de database

De methode `itemKlik`, die wordt aangeroepen bij het aanklikken van een lijst-item, doet vrijwel hetzelfde als in de vorige versie. Het enige verschil is dat ditmaal niet de *positie* in de lijst wordt gebruikt, maar de `Id` die in het `KleurItem` is opgeslagen.
Dit maakt het gemakkelijker om in de methode `OnActivityResult` het gewenste record in de database terug te vinden. De methoden `Update` en `Delete` werken namelijk op een record met de gegeven `Id` als key. De werking spreekt voor zich: `Update` verandert de overige velden van het record in die van het meegegeven `KleurItem`, en `Delete` gooit het record met de gegeven key weg, ongeacht de waardes van de overige velden.

10.5 Een eigen subklasse

In de vorige sectie hebben we `KleurenApp3` geschreven: een aangepaste versie van `KleurenApp2` met daaraan toegevoegd code om de gegevens in een database op te slaan. Een groot deel van de code bleef hetzelfde, en slechts in enkele methoden hoefden veranderingen te worden aangebracht.

Een programma kopiëren en aanpassen

Dit was de aanpak bij het schrijven van `KleurenApp3`: we begonnen met een kopie van `KleurenApp2`. Als eerste veranderden we de naam van de klasse. Alle declaraties van variabelen (zoals de `ListView`, de adapter, de lijst met kleuren, en de array met standaardkleuren) zijn nog steeds nodig. Sommige methoden konden in hun geheel worden overgenomen: de methode `OnCreate` waarin de userinterface werd opgebouwd, en de drie event-handlers `verstuurKlik`, `nieuwItem` en `itemKlik`.
Drie andere methoden hebben een nieuwe invulling gekregen (zie figuur 28)

- De methode `BeginKleuren` kopieerde in versie 2 simpelweg de array met standaardkleuren naar de kleurenlijst. In versie 3 legt deze methode de verbinding met de database. Als de database nog niet bestaat, wordt die aangemaakt, en worden *daarin* de standaardkleuren neergezet.

- Aan de methode `LeesKleuren` wordt toegevoegd dat de kleurenlijst wordt ingelezen uit de database. Daarna doet de methode ook wat hij in versie 2 al deed: de kleurenlijst gebruiken in de adapter bij de listview.
- De methode `OnActivityResult` wordt aangepast: in plaats van dat de veranderde kleur in de lijst wordt teruggezet, wordt deze in de database aangepast. Door de aanroep van `LeesKleuren` aan het eind van de methode wordt de database daarna opnieuw ingelezen, en worden de veranderde kleuren dus inderdaad gebruikt.

In deze drie methoden wordt ook een nieuwe variabele gebruikt: de `SQLiteConnection` database.

Nadelen van copy/paste/edit

Op deze manier kun je wel snel een programma schrijven, maar de aanpak heeft een groot nadeel. Dat merk je niet meteen, want in eerste instantie ben je wel blij met het snel geschreven programma. Maar na verloop van tijd zou het kunnen gebeuren dat er een bug wordt ontdekt in het gekopieerde programma. Of er wordt een verbeterde versie uitgebracht van het oorspronkelijke programma, en je wilt die verbetering ook graag in de database-versie van het programma opnemen.

In beide gevallen moet je de bugfix/de verbetering niet alleen in versie 2 aan (laten) brengen, maar apart ook nog eens in versie 3. Niet alleen geeft dat extra werk, maar er is ook het risico dat je het verkeerd doet, of helemaal vergeet. Dit alles wordt nog erger als er inmiddels alweer is voortgebouwd op versie 3, misschien wel meerdere malen, en er een hele waaier van versies bestaat die *allemaal* aangepast moeten worden om de bugfix of verbetering ook door te voeren.

Subklasse: hergebruik zonder kopiëren

Dit kan handiger! Er is in C# (en in elke object-georiënteerde taal) een mechanisme waarmee je kunt profiteren van een eerder geschreven klasse, zonder die klasse in zijn geheel te kopiëren. Dit mechanisme hebben we al lang gebruikt: het is het schrijven van een *subklasse*.

In listing 37 ontwikkelen we nogmaals de database-variant van het kleurenprogramma. Ditmaal blz. 151 niet door een kopie van de oorspronkelijke klasse te maken, maar door in de header aan te geven dat het er een uitbreiding van is:

```
class KleurenApp4 : KleurenApp2
```

Want dat is wat een subklasse is: een uitbreiding van de klasse die achter de dubbele punt wordt genoemd. In de subklasse zijn alle oorspronkelijke methoden te gebruiken, en kun je methoden toevoegen en/of aanpassen.

Als je listing 37 vergelijkt met listing 36 zie je dat de code vrijwel hetzelfde is. Maar de nieuwe blz. 151 listing is het complete bestand! De oude listing begon op regel 79, en er stond in commentaar dat blz. 150 je er het overeenkomstige stuk uit listing 30 ook nog voor moest denken. Met een subklasse kun blz. 142 je precies dit effect bereiken, maar dan zonder trucjes als 'zet dit er nog even voor' en 'verander dit nog even'. De aanpak staat samengevat in figuur 29.

Override: methode krijgt een nieuwe invulling

Je kunt ook methoden die in de oorspronkelijke klasse al bestonden een nieuwe invulling geven. Je schrijft dan `override` in de header. In dit voorbeeld is dat voor alle drie de methoden het geval: `BeginKleuren`, `LeesKleuren` en `OnActivityResult` hadden al een invulling in de oorspronkelijke klasse, maar in de database-versie krijgen ze een nieuwe invulling.

Virtual: methode die overriden mag worden

De auteur van de oorspronkelijke klasse moet bij een methode aangeven dat het toegestaan is om hem in een subklasse een nieuwe invulling te geven. In de header van zo'n methode staat erbij dat de methode `virtual` is. Met een vooruitziende blik hadden we in `KleurenApp2` de methoden `BeginKleuren` en `LeesKleuren` inderdaad als `virtual` methode gedefinieerd. Daarom mogen ze in `KleurenApp3` met `override` opnieuw gedefinieerd worden.

Base: roep de oorspronkelijke methode aan

In het voorbeeld krijgt de methode `BeginKleuren` een geheel nieuwe invulling. Bij de methode `LeesKleuren` is er iets anders aan de hand. In de oorspronkelijke versie gebeurde er dit:

```
protected virtual void LeesKleuren()
{   kleurAdapter = new KleurenAdapter(this, kleuren);
    kleurLijst.Adapter = kleurAdapter;
}
```

```
public class KleurApp2 : Activity
{
  ListView kleurLijst;
  List<KleurItem> kleuren;
  KleurItem[] defaults = ...;
  KleurenAdapter kleurAdapter;

  void OnCreate(Bundle b) {...}

  void verstuur(object o, EventArgs ea) {...}

  void nieuwItem(object, EventArgs)  {...}

  void itemKlik(object o, EventArgs ea) {...}



  void BeginKleuren( ) {...}

  void LeesKleuren( ) {...}

  void OnActivityResult
        (int pos, Result res, Intent data) {...}
}
```

```
public class KleurApp3 : Activity
{
  ListView kleurLijst;
  List<KleurItem> kleuren;
  KleurItem[] defaults = ...;
  KleurenAdapter kleurAdapter;

  void OnCreate(Bundle b) {...}

  void verstuur(object o, EventArgs ea) {...}

  void nieuwItem(object, EventArgs)  {...}

  void itemKlik(object o, EventArgs ea) {...}

  SQLiteConnection database;

  void BeginKleuren( ) {NIEUW}

  void LeesKleuren( ) {NIEUW}

  void OnActivityResult
        (int p, Result r, Intent d) {NIEUW}
}
```

Figuur 28: Klasse `KleurenApp3` is een aangepaste versie van `KleurenApp2`

```
public class KleurApp2 : Activity
{ protected
  ListView kleurLijst;
  List<KleurItem> kleuren;
  KleurItem[] defaults = ...;
  KleurenAdapter kleurAdapter;
  protected override
  void OnCreate(Bundle b) {...}
  private
  void verstuur(object o, EventArgs ea) {...}
  private
  void nieuwItem(object, EventArgs)  {...}
  private
  void itemKlik(object o, EventArgs ea) {...}



  protected virtual
  void BeginKleuren( ) {...}
  protected virtual
  void LeesKleuren( ) {...}
  protected override
  void OnActivityResult
        (int pos, Result res, Intent data) {...}
}
```

```
public class KleurApp4 : KleurApp2
{
  private
  SQLiteConnection database;
  protected override
  void BeginKleuren( ) {NIEUW}
  protected override
  void LeesKleuren( ) {NIEUW}
  protected override
  void OnActivityResult
        (int p, Result r, Intent d) {NIEUW}
}
```

Figuur 29: Klasse `KleurenApp4` is een subklasse van `KleurenApp2`

In de nieuwe versie moet er dit gebeuren:

```
protected override void LeesKleuren()
{   kleuren = new List<KleurItem>();
    TableQuery<KleurItem> query = database.Table<KleurItem>();
    foreach (KleurItem k in query)
        kleuren.Add(k);
    kleurAdapter = new KleurenAdapter(this, kleuren);
    kleurLijst.Adapter = kleurAdapter;
}
```

De laatste twee regels zijn hetzelfde als in de oorspronkelijke versie. De methode krijgt dus niet een helemaal nieuwe invulling, maar is een *uitbreiding* van de oorspronkelijke methode. We zitten nu toch weer stukken code te kopiëren, met alle nadelen in het versiebeheer die daar het gevolg van zijn.
Beter kunnen we daarom de oorspronkelijke methode aanroepen. Je kunt dat doen zoals je altijd methoden aanroept: door de naam op te schrijven. Voor de punt staat er echter niet `this` (want dan zou de methode zichzelf aanroepen, en zich daarmee oneindig lang in de staart blijven bijten) maar `base`. Het keyword `base` duidt hetzelfde object aan als `this`, maar dan met *de oorspronkelijke klasse als type*. We schrijven dus:

```
protected override void LeesKleuren()
{   kleuren = new List<KleurItem>();
    TableQuery<KleurItem> query = database.Table<KleurItem>();
    foreach (KleurItem k in query)
        kleuren.Add(k);
    base.LeesKleuren();
}
```

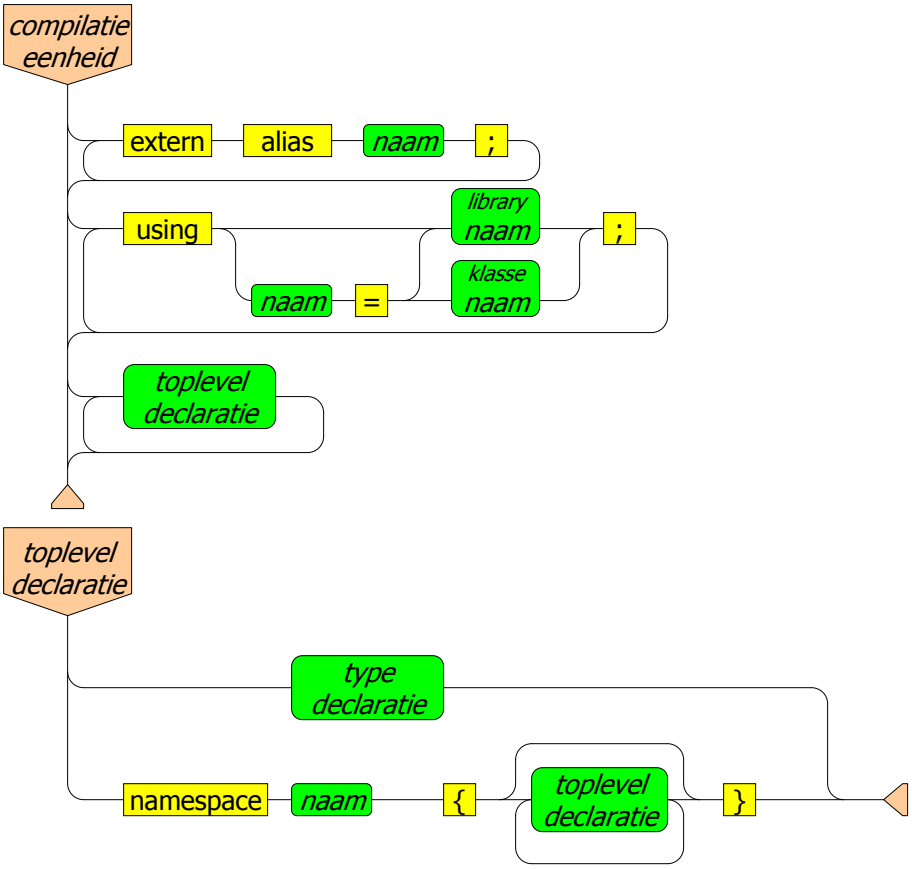
Op de laatste regel wordt de oorspronkelijke versie van `LeesKleuren` aangeroepen.

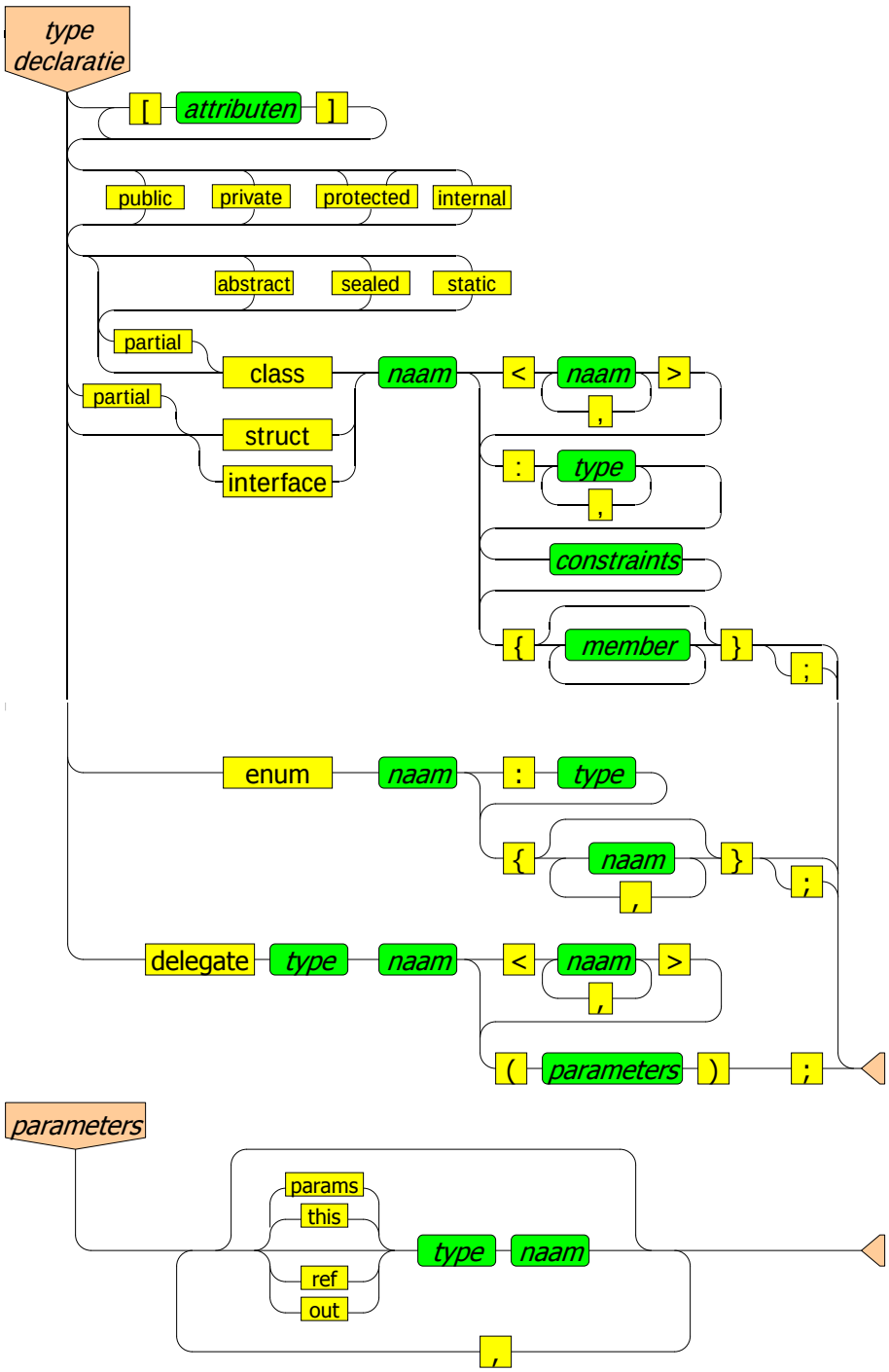
Virtuele methoden in de Android-library

We hebben het principe van virtuele methoden al lang gebruikt. In de klasse `Activity` zitten twee belangrijke virtuele methoden: `OnCreate` en `OnActivityResult`. Het is toegestaan, en vaak zelfs de bedoeling, dat je deze methoden in een subklasse met `override` een nieuwe invulling geeft. We hebben dat in alle activities in dit diktaat steeds braaf gedaan.
In de eerste hoofdstukken leek het misschien dat de noodzaak om een methode als `OnCreate` te schrijven iets is dat fundamenteel is ingebakken in C#. Dat is niet het geval: het is gewoon een ideetje dat de programmeur van `Activity` heeft bedacht, en waaraan iedereen die een subklasse van `Activity` schrijft zich heeft te houden. In andere klassen zitten weer andere virtuele methoden. Zo zit er in de klasse `View` de methode `OnDraw`, waaraan je geacht wordt een invulling te geven in een subklasse.
Het principe van virtuele methoden die in een subklasse overriden worden is niet een voorrecht dat alleen in libraries gebruikt wordt. Je kunt het ook in je eigen programma's gebruiken. De casus in dit hoofdstuk is daar een voorbeeld van. Natuurlijk is dat pas de moeite waard als je programma's wat groter worden, en er sprake is van varianten of uitbreidingen. En de straf op de verleidelijke aanpak van copy/paste/edit is er pas als zo'n programma in de toekomst ook onderhouden moet worden. Dat geldt echter voor veel van de programma's die je nog zult gaan schrijven of laten schrijven, vaak in teamverband. Het gebruik van klassen met virtuele methoden die in subklassen overriden worden zul je daarom nog vaak tegenkomen.

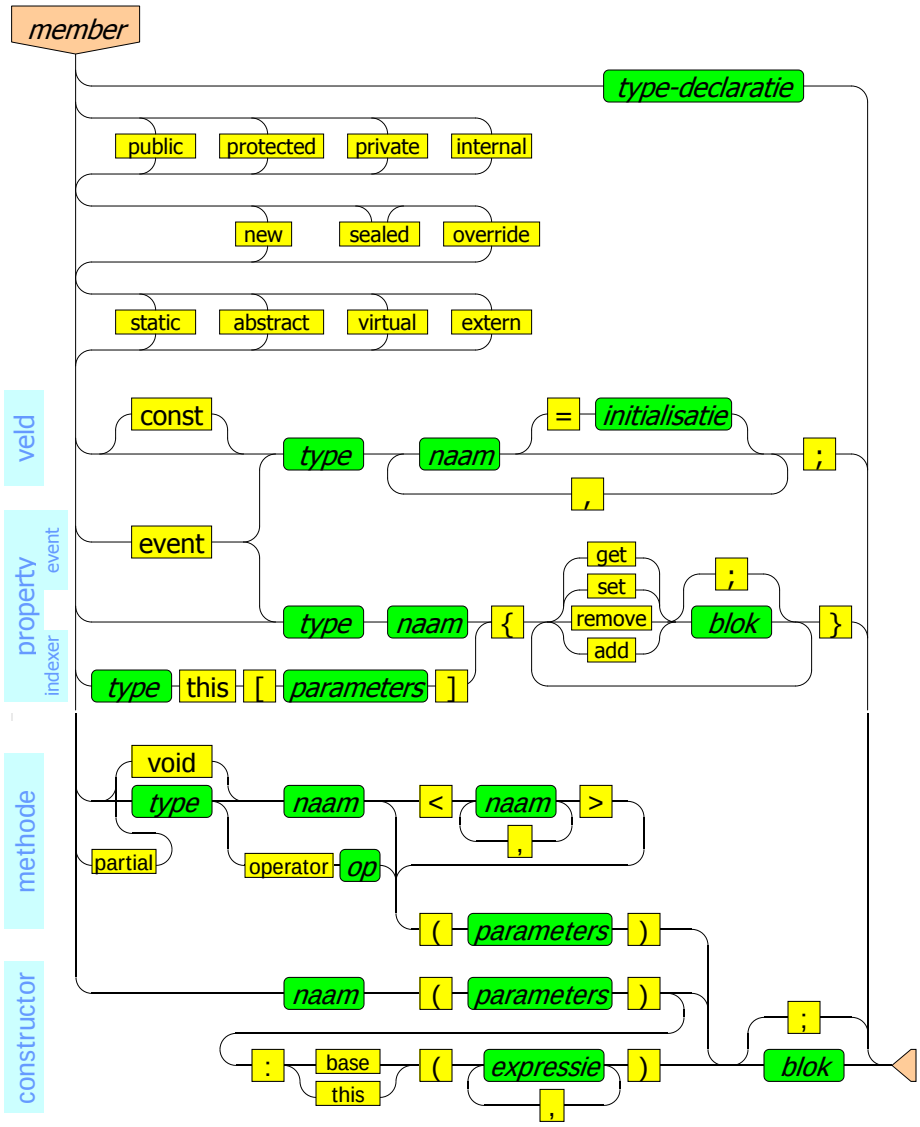
Bijlage A

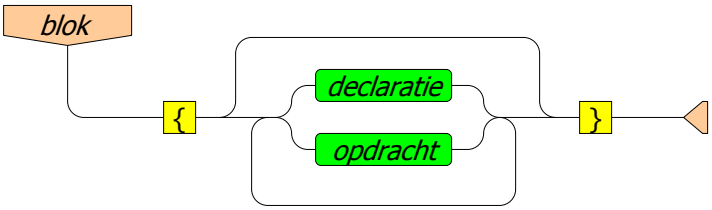
Syntax

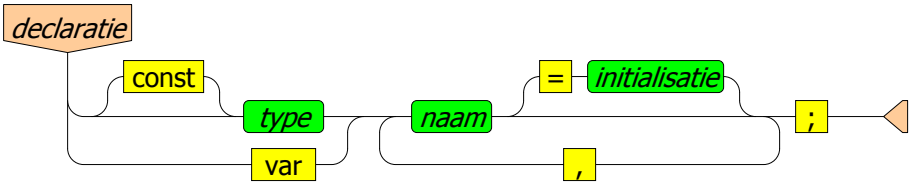


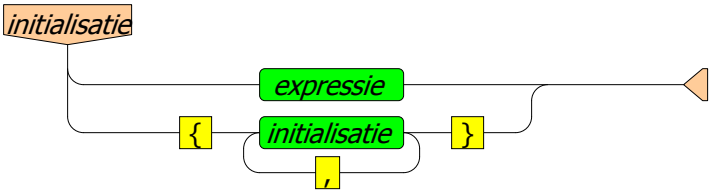

type declaratie
[
attributen
]
public
private
protected
internal
abstract
sealed
static
partial
class
partial
struct
interface
naam
<
naam
>
,
:
type
,
constraints
{
member
}
;
enum
naam
:
type
{
naam
}
,
;
delegate
type
naam
<
naam
>
,
(
parameters
)
;

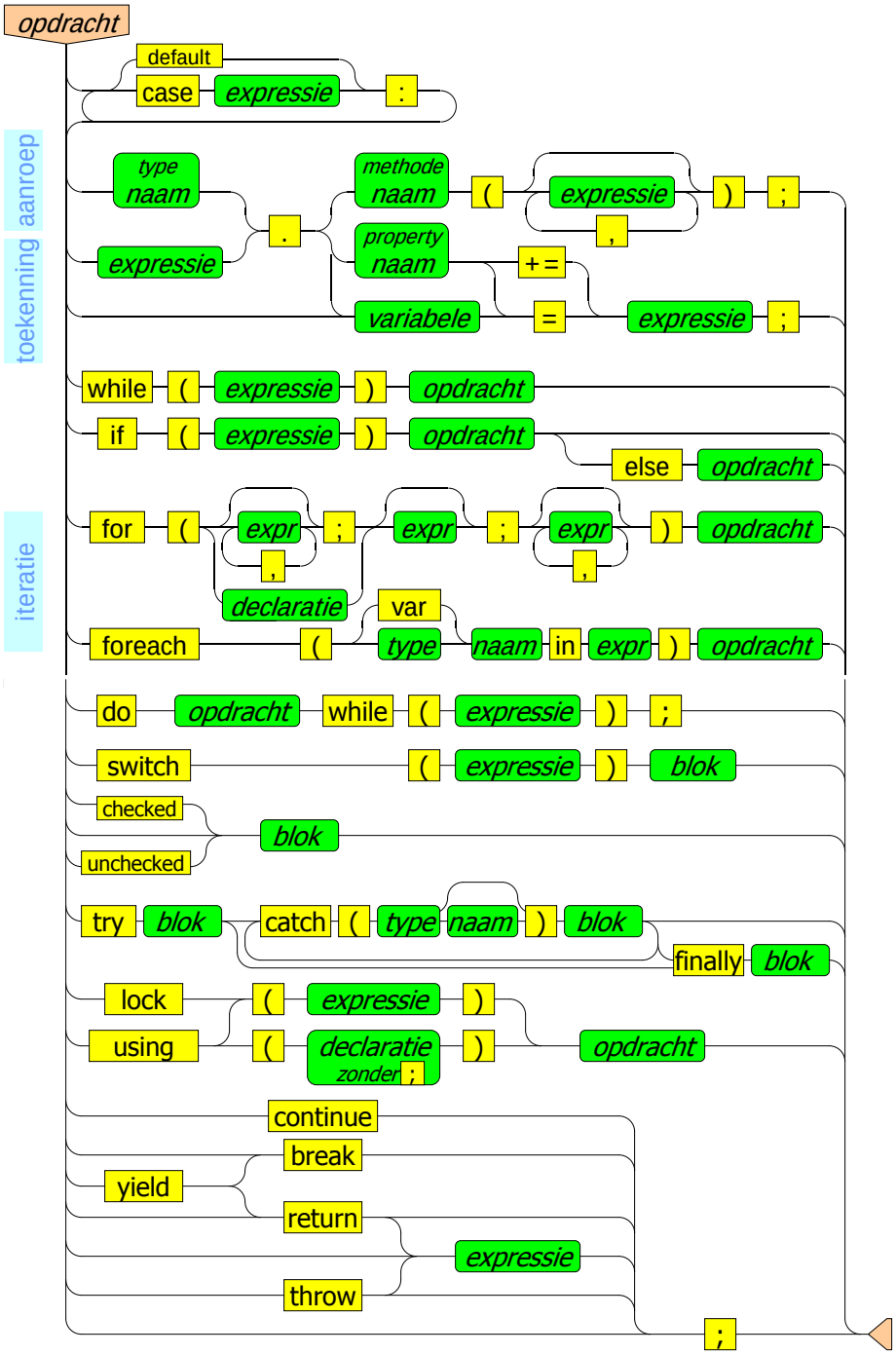
parameters
params
this
ref
out
type
naam
,

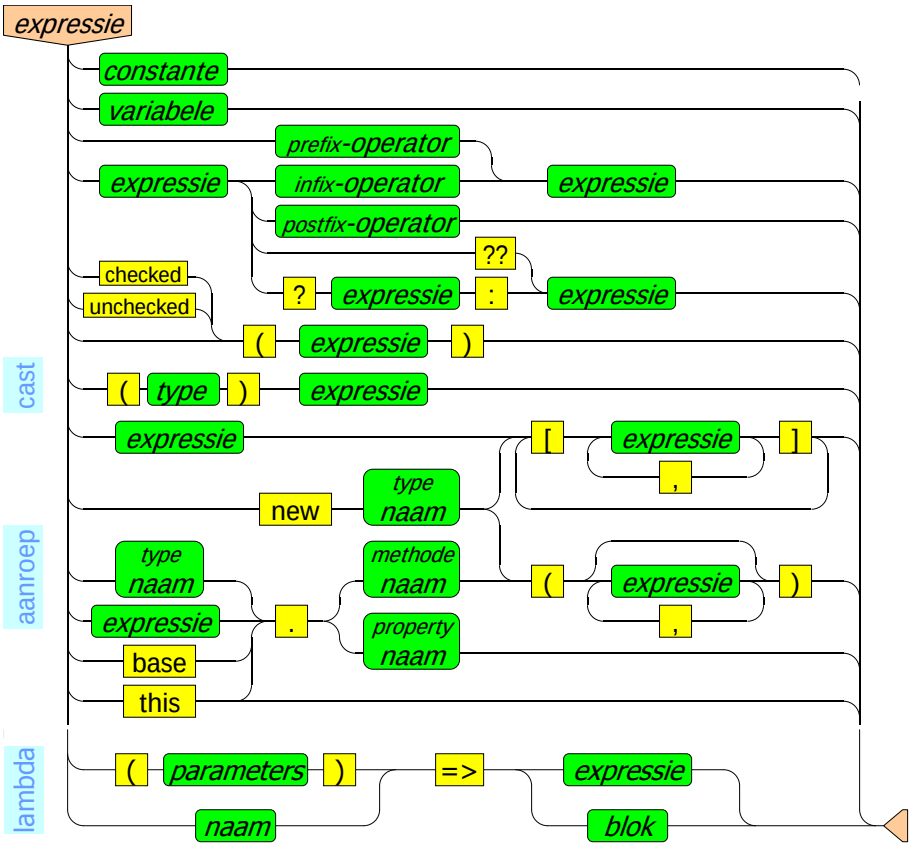

member
type-declaratie
public
protected
private
internal
new
sealed
override
static
abstract
virtual
extern
veld
const
type
naam
=
initialisatie
;
,
property
event
event
type
naam
{
get
set
remove
add
;
blok
}
indexer
type
this
[
parameters
]
methode
void
type
naam
<
naam
>
,
partial
operator
op
(
parameters
)
constructor
naam
(
parameters
)
:
base
this
(
expressie
)
,
;
blok

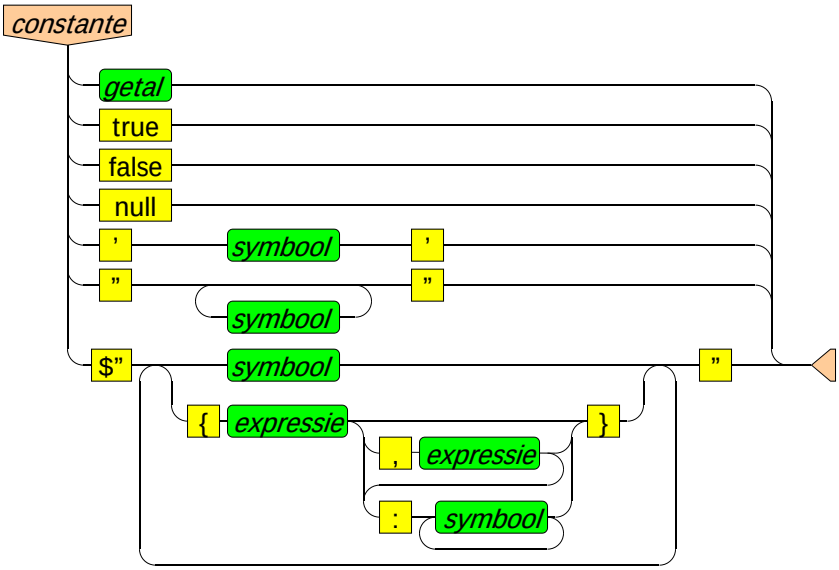

blok
{
declaratie
opdracht
}

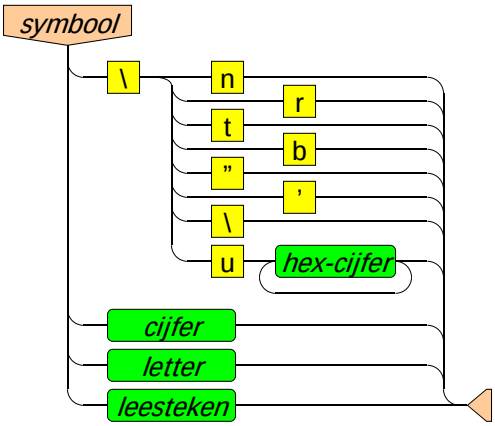

declaratie
const
type
var
naam
=
initialisatie
;
,

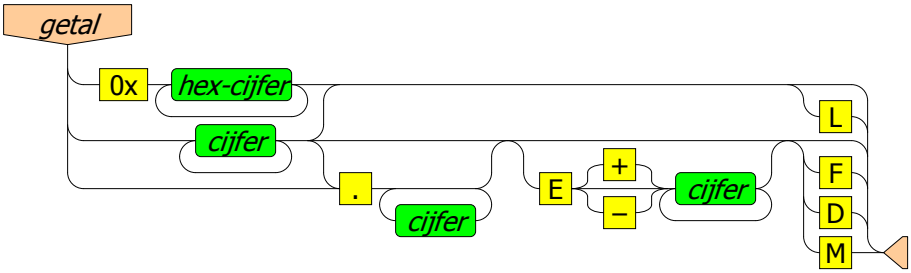

initialisatie
expressie
{
initialisatie
}
,

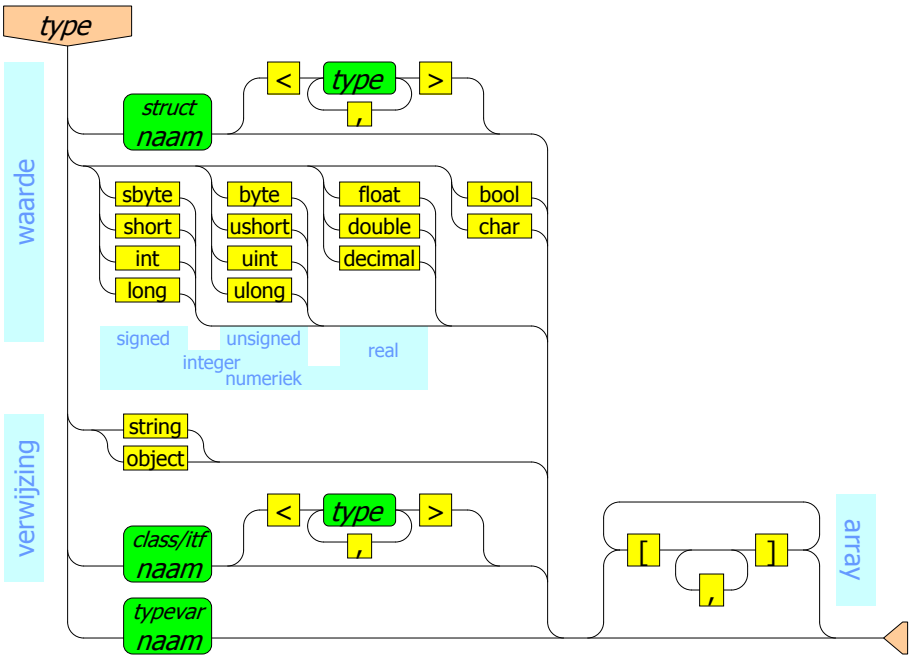

opdracht
default
case
expressie
:
aanroep
toekenning
type
naam
expressie
.
methode
naam
(
expressie
,
)
;
property
naam
+=
variabele
=
expressie
;
while
(
expressie
)
opdracht
if
(
expressie
)
opdracht
else
opdracht
iteratie
for
(
expr
,
;
expr
;
expr
,
)
opdracht
declaratie
var
foreach
(
type
naam
in
expr
)
opdracht
do
opdracht
while
(
expressie
)
;
switch
(
expressie
)
blok
checked
unchecked
blok
try
blok
catch
(
type
naam
)
blok
finally
blok
lock
(
expressie
)
using
(
declaratie
zonder ;
)
opdracht
continue
break
yield
return
expressie
throw
;


expressie
constante
variabele
prefix-operator
expressie
infix-operator
expressie
postfix-operator
??
checked
unchecked
?
expressie
:
expressie
(
expressie
)
cast
(
type
)
expressie
expressie
[
expressie
,
]
new
type naam
aanroep
type naam
methode naam
(
expressie
,
)
expressie
.
property naam
base
this
lambda
(
parameters
)
=>
expressie
naam
blok


constante
getal
true
false
null
'
symbool
'
"
symbool
"
$"
symbool
"
{
expressie
,
expressie
:
symbool
}


symbool
\
n
r
t
b
"
'
\
u
hex-cijfer
cijfer
letter
leesteken


getal
0x
hex-cijfer
cijfer
.
cijfer
E
+
−
cijfer
L
F
D
M


type
waarde
struct
naam
<
type
,
>
sbyte
short
int
long
byte
ushort
uint
ulong
float
double
decimal
bool
char
signed
unsigned
integer
numeriek
real
verwijzing
string
object
class/itf
naam
<
type
,
>
typevar
naam
[
,
]
array

Bijlage B

Werkcollege-opgaven

B.1 Serie 1

1.1 *Syntax en semantiek*
Geef een beschrijving (liefst zo kernachtig mogelijk) van de syntax en de semantiek van de volgende drie soorten opdrachten:

- toekenningsopdracht
- methode-aanroep
- return-opdracht

Probeer de syntax en de semantiek zo veel mogelijk te scheiden: beschrijf eerst puur de syntax, en daarna apart de semantiek.
Vergelijk daarna de beschrijving met die van iemand anders. Kon het compacter? Kon het duidelijker?

1.2 *Declaratie, opdracht, expressie*
Wat is het verschil tussen een *declaratie*, een *opdracht* en een *expressie*?

1.3 *Vermenigvuldigen en delen*
Hieronder staan drie alternatieve formuleringen van het bepalen van het top-punt van het dak van de huizen uit het voorbeeldprogramma `Huizen`. Is er verschil in het effect van deze drie opdrachten?

```
topy = y - 3*br / 2;
topy = y - 3/2 * br;
topy = y - br/2 * 3;
```

1.4 *Android klassen*
Geef in een schema weer op welke manier onderstaande klassen (uit de libraries en uit de voorbeeldprogramma's) subklassen van elkaar zijn. (Sommige zijn duidelijk, maar er zitten ook wat subtiele gevallen tussen!).

- Activity
- Button
- Context
- HalloApp
- LinearLayout
- MondriaanView
- object
- TextView
- View

1.5 *Methoden met een resultaat*
Schrijf een methode die, gegeven een rechthoek-object die als parameter wordt meegegeven, de *omtrek* van die rechthoek berekent (dus wat de totale lengte is van een draad die rondom de rechthoek gespannen zou worden).
Schrijf ook een methode die de lengte van de *diagonaal* van een gegeven rechthoek berekent.

1.6 *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?

1.7 *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                x=y+1;
ax02              2xa0                  0x2a
Button b=ok;      Button                new Button(this)
ll.AddView(b);    String ok(Button b)   class OK : View
```

1.8 *Methode schrijven en gebruiken*
Maak een schetsje van de `View` met onderstaande `OnDraw`-methode.
Herschrijf nu de methode `OnDraw`, waarbij je de regelmaat duidelijker naar voren laat komen met gebruikmaking van een zelfgeschreven extra methode.

```
public void OnDraw(Canvas canvas)
{
    Paint verf = new Paint();
    verf.Color = Color.Black;
    canvas.DrawLine(10, 10, 20, 20, verf);
    canvas.DrawLine(20, 10, 10, 20, verf);
    verf.Color = Color.Red;
    canvas.DrawLine(30, 10, 50, 30, verf);
    canvas.DrawLine(50, 10, 30, 30, verf);
    verf.Color = Color.Blue;
    canvas.DrawLine(60, 10, 90, 40, verf);
    canvas.DrawLine(90, 10, 60, 40, verf);
}
```

B.2 Serie 2

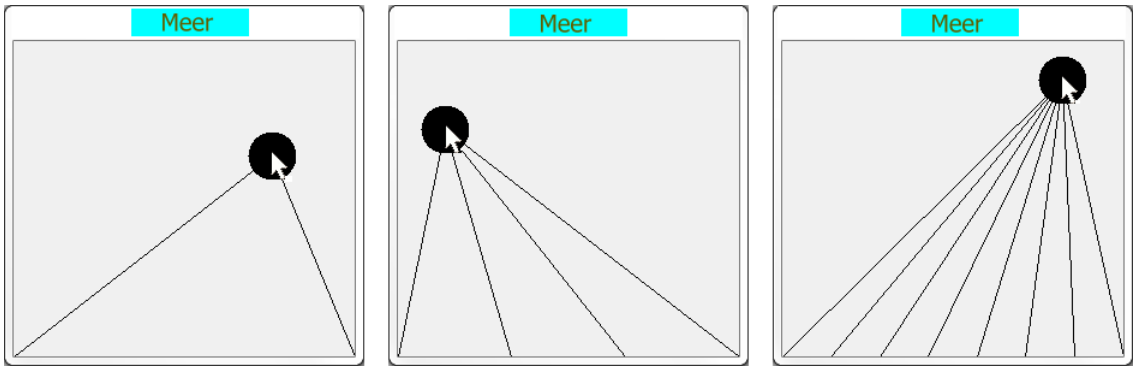
2.1 *Keywords*
Wat betekent het Engelse woord `void`, en in welke situatie is dit keyword nodig?
Wat betekent de Engelse afkorting `int`, en in welke situatie is dit keyword nodig?
Wat betekent in een C#-opdracht het woord `return`, en in welke situatie is dit keyword nodig?
Wat wordt in een C#-methode aangeduid door `this`, en in welke situatie is dit keyword nodig?

2.2 *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 `"-|-||-|||-||||-|||||-"`
`this.spijker(7)` geeft `"-|||||||-"`
`this.spijker(203)` geeft `"-||--|||-"`
Hint: verwerk eerst het laatste cijfer, en herhaal dan voor de rest van de cijfers. Het laatste cijfer kun je te pakken krijgen door slim gebruik te maken van de 'rest bij deling' operator.

2.3 *Stralen*
Gegeven is de volgende klasse:

```
class StralenApp : Activity
{   public override void OnCreate(Bundle b)
    {   base.OnCreate(b);
        Button knop = new Button(this);
        StralenView sv = new StralenView(this);
        LinearLayout lay = new LinearLayout(this);
        lay.Orientation = Orientation.Vertical;
        knop.Text = "Meer";
        knop.Click += sv.klik;
        lay.AddView(knop);
        lay.AddView(sv);
        this.SetContentView(lay);
    }
}
```

Schrijf de klasse `StralenView`, 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 aan het begin in het midden van het scherm. Als de gebruiker het scherm aanraakt, komt het punt op die plek te staan. En als de gebruiker met zijn vinger over het scherm beweegt, beweegt de cirkel ook mee.
Twee lijnen verbinden het midden van de cirkel met de twee onderhoeken van het window. Elke keer als de gebruiker op de knop drukt komt er een lijn bij. De lijnen monden op gelijke afstanden uit op de onderrand van het window.
Zie onderstaande figuur, met daarin: de beginstituatie, de situatie na 2 keer klikken, en de situatie na nog 4 keer klikken. (De pijl geft de vinger-positie aan, deze hoef je niet te tekenen).



2.4 Beschrijf de syntax van de foreach-opdracht.
Beschrijf daarna ook de semantiek van de foreach-opdracht.

2.5 Variabelen kunnen onder andere worden gedeclareerd in de *header van een methode*, in de *body van een methode*, en in de *body van een klasse*.
Hoe krijgt een variabele die is gedeclareerd in de *header van een methode* zijn waarde, en waar wordt zo'n variabele voor gebruikt?
Hoe krijgt een variabele die is gedeclareerd in de *body van een methode* zijn waarde, en waar wordt zo'n variabele voor gebruikt?
En hoe krijgt een variabele die is gedeclareerd in de *body van een klasse* zijn waarde, en waar wordt zo'n variabele voor gebruikt?

2.6 In de library worden veel `class`-typen gedefiniëerd, maar ook `struct`-typen. Daar zijn veel overeenkomsten tussen, maar wat is het verschil?

2.7 In de library worden naast `class`-typen ook `interface`-typen gedefiniëerd. Stel dat er in een library een interface `IA` is gedefinieerd, en je gebruikt die in een programma. Waar zul je die naam `IA` in het programma typisch gebruiken, en welke verlichting brengt dat met zich mee?

2.8 Hieronder staat 16 fragmenten uit een programma. Schrijf bij elk fragment een letter passend bij het overeenkomstige fragment:

- **T** als het programmafragment een **type** is
- **E** als het programmafragment een **expressie**
- **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

```
{ }             int a(Color c)          override void()      while(true);
Color           return Color.Red;       const bool b=1==0;   for (x=0;x++) t=0;
"\""+2          new Color(1,2,3)        bool b(bool c)       true==false
t=t%t;          Color.Red               bool                 Color c;
```

B.3 Serie 3

3.1 *Android-klassen*
Voor welk doel maak je een subklasse van de volgende klassen? Welke methoden zitten daar dan in ieder geval in, en wat is de taak van die methoden?

- `Activity`
- `View`
- `BaseAdapter`

3.2 *Bijzondere methoden*
Hoe onderscheiden de volgende soorten methoden zich van andere methoden? Op welke manier worden ze aangeroepen?

- Constructor-methoden
- Static methoden
- Event-handlers

3.3 *Een eigen klasse*
In een bepaald programma worden locatie-gegevens verwerkt, volgens het kilometerstelesel van de Nederlandse topografische dienst. Maar van de locaties is ook de hoogte van belang.
Schrijf een klasse waarmee objecten worden beschreven waarin deze gegevens worden opgeslagen. Geef de klasse een paar verschillende constructor-functies die in de praktijk handig zouden kunnen zijn. Maak ook een methode die zo'n object omzet naar een string, en twee methodes die het omgekeerde doen: een statische, en een extra constructor. Schrijf ook een zinvolle methode die de afstand berekent tussen twee objecten van dit type. En maak tenslotte een paar constanten van belangrijke locaties.

3.4 *Array-methoden*
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.

3.5 *Array-methoden*
Schrijf een methode `Gemiddelde` die als parameter een array van getallen meekrijgt, en daarvan het gemiddelde uitrekent.

3.6 *Vergelijken van strings*
In de klasse `String` zit onder andere de methode

```
static int Compare(string, string)
```

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:

`String.Compare("aap", "noot")`	geeft een negatief getal, want `'a'<'n'`
`String.Compare("noot", "nieten")`	geeft een positief getal, want `'o'>'i'`
`String.Compare("niet", "nietmachine")`	geeft een negatief getal vanwege de lengte
`String.Compare("noot", "noot")`	geeft 0, want precies gelijk
`String.Compare("noot", "NOOT")`	geeft een positief getal, want `'n'>'N'`

Schrijf deze methode, *zonder* gebruik te maken van de bestaande `Compare` en `CompareTo` methoden.

3.7 *Analyse van zinnen*
Schrijf een methode met een string als parameter, die uit woorden bestaat (bijvoorbeeld `"dit is een zin met losse woorden gescheiden door spaties"`). De methode moet bepalen hoe lang het langste woord is (in het voorbeeld is dat: 10). Schrijf een andere methode die bepaalt welk woord het langste is (in het voorbeeld is dat: `"gescheiden"`). (Als er meerdere kandidaten zijn mag je kiezen welke daarvan je oplevert).

B.4 Serie 4

4.1 *Plaatje schalen*
Gegeven is een bitmap `b`. We willen deze bitmap tekenen in de `OnDraw`-methode van een subklasse van `View`.

a. Hoe kun je een bitmap vergroot op het scherm tekenen?
b. We kunnen de schaalfactor berekenen met deze opdracht:

```
Schaal = Math.Min( ((float)this.Width)  / this.Plaatje.Width
                 , ((float)this.Height) / this.Plaatje.Height );
```

Hoe ziet het scherm er uit als we de bitmap met deze schaalfactor tekenen?
c. En hoe als we in plaats van `Min` de methode `Max` gebruiken?
d. Wat gebeurt er als we de cast `(float)` niet zouden gebruiken?

4.2 *Sensoren*
Om de sensoren van een device (kompas, thermometer, enz.) uit te lezen gebruik je een `SensorManager`-object. Welke methode roep je aan, en wat is er verder nodig om de sensorwaarden te verkrijgen?

4.3 *Aanraakscherm*
Schrijf een `View`-subklasse die (wanneer een object ervan als contentview in een app wordt gebruikt) alle punten op het scherm die de gebruiker met de vinger aantikt met lijnen verbindt.

4.4 *Operatoren*
Welke bewerking voeren de volgende operatoren uit: `%`, `&&`, `+=`, `??`, `<=`

4.5 *Analyse van zinnen (vervolg)*
Vorige week schreven we een methode die bepaalt hoe lang het langste woord van een zin is. Schrijf een methode met een string als parameter, die weer uit woorden bestaat (bijvoorbeeld `"dit is een zin met losse woorden gescheiden door spaties"`). Ditmaal berekenen we *welke woordlengte het vaakste voorkomt*. In het voorbeeld is dat 3, omdat het de drieletterwoorden zijn die het vaakste voorkomen (namelijk 4 keer: `"dit"`, `"een"`, `"zin"`, `"met"`). De andere woordlengtes komen minder vaak voor.
Je mag aannemen dat geen enkel woord langer is dan 29 letters. Hint: turf eerst van alle woordlengtes hoe vaak ze voorkomen, en bepaal daarna welk daarvan het vaakste is.

4.6 *String-methodes (vervolg)*
In de klasse `String` zitten onder andere de volgende methodes:

```
string Substring(int start, int lengte)
bool   Contains (string s)
```

De hier bedoelde versie van `Substring` levert het deel van de string op vanaf het meegegeven startpunt, met de meegegeven lengte. Bij te grote lengte geeft de methode zo veel letters als hij heeft. Voorbeelden:

`"Utrecht".Substring(2,4)` geeft `"rech"`
`"Utrecht".Substring(3,10)` geeft `"echt"`

De methode `Contains` geeft aan of de parameter ergens voorkomt in het totaal. Voorbeelden:

`"Utrecht".Contains("ech")` geeft true
`"Utrecht".Contains("trh")` geeft false
`"Utrecht".Contains("Utrecht")` geeft true

Stel dat je de auteur van de klasse `String` bent. Enkele methoden en properties zijn al geschreven: de indexerings-property om een losse letter te pakken, de `Length`-property, de operator `+`, `==`, of als je wilt de methodes `Concat` en `Equals` zijn al beschikbaar (die mag je dus gebruiken). De andere methoden ontbreken nog en mag je dus niet aanroepen.
De opgave: Schrijf de twee hierboven beschreven methoden. (De zelf-geschreven methodes mogen wel elkaar aanroepen).

4.7 *Zelfgemaakte klasse*
In de library `Android.Graphics` zitten klassen `Rect` en `RectF`. Objecten van die klasse beschrijven de plaats en grootte van een rechthoek. We gaan een klasse `BlokF` schrijven die ook een rechthoek beschrijft, maar dan net een beetje anders dan `RectF`. Je mag bij het schrijven ervan niet de

bestaande `Rect`-klassen gebruiken, maar wel `PointF`.
Een object van de klasse `BlokF` legt de eigenschappen van een rechthoek vast door de *linker bovenhoek*, en de *breedte* en de *hoogte* op te slaan. De klasse heeft de volgende methoden:

- een constructormethode waaraan de gewenste linker bovenhoek, breedte, en hoogte als parameter worden meegegeven
- een constructormethode waaraan twee tegenoverliggende hoekpunten van de gewenste rechthoek worden meegegeven
- een methode om de oppervlakte van de rechthoek te berekenen
- een methode om de rechter onderhoek van de rechthoek te bepalen
- een methode om de gegevens van de rechthoek in een string weer te geven
- een methode om de string van de vorige methode weer om te zetten in een rechthoek (naar keuze een extra constructormethode, of de andere gangbare manier om dit te doen)
- een methode die bepaalt of de rechthoek het punt, dat als parameter wordt meegegeven, bevat

4.8 *Arrays*

In onderstaand raamwerk van een klasse wordt een array gedeclareerd met daarin de uitslagen van een tentamen. Als een object van deze klasse wordt gebruikt als contentview van een app, dan wordt er een staafdiagram van deze uitslagen op het scherm getekend, zoals in de afbeelding.
De uitslagen worden daarbij naar beneden afgerond, dus bijvoorbeeld een 7.5 wordt meegeteld in de staaf behorend bij 7. Het programma moet ook gebruikt kunnen worden als, later, de array wordt aangepast (waarna het programma opnieuw wordt gecompileerd). De hoogte van de balkjes is 20 beeldpunten, de breedte van de balkjes is 10 beeldpunten voor elke corresponderende uitslag. Als de gebruiker een balkje aanraakt, moet korte tijd een toelichting voor dat balkje op het scherm te zien zijn, in een pop-up kadertje dat na een paar seconden weer verdwijnt.

```
public class Diagram : View
{   double [] cijfers
    = { 10, 8, 9, 7, 10, 6.5, 8, 7.5, 9, 7.5, 8, 8.5, 4, 6.5, 8.5, 8, 8, 8, 7, 8.5, 9, 8.5, 4, 5.5, 9, 9,
      , 8, 8, 6, 7.5, 9, 8, 9, 6, 8.5, 7, 9, 6, 8.5, 8.5, 9, 8, 6, 8, 7, 8.5, 7.5, 8, 7.5, 8, 8, 9, 9, 8.
    Context ctx;
    // opgave a
    public Diagram(Context c) : base(c)
    {   ctx = c;
        // opgave b
    }
    protected override void OnDraw(Canvas c)
    {   base.OnDraw(c);
        // opgave c
    }
    public void RaakAan(object o, TouchEventArgs tea)
    {  // opgave d
    }
}
```

Opgaven (schrijf de letter er duidelijk bij, dan hoef je de gegeven headers niet over te schrijven)

1 Schrijf de nog benodigde declaratie van een variabele in de klasse
2 Schrijf de body van de constructormethode (hint: doe hier o.a. het benodigde turf-werk)
3 Schrijf de body van de `OnDraw`-methode
4 Schrijf de body van de event-handler (hint: let op dat het niet crasht als de gebruiker het scherm onder het 10-balkje aanraakt)

Bijlage C

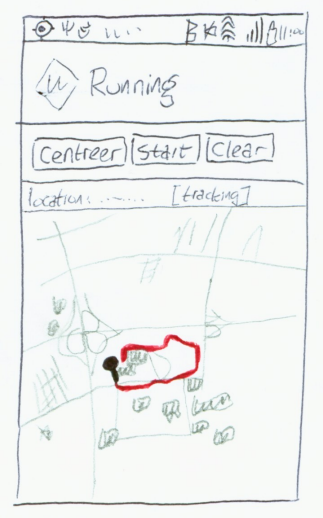
Practicum

In het practicum gaan we een 'Running app' ontwikkelen. Dit is een app waarmee de gebruiker zijn/haar prestaties kan meten en analyseren tijdens een dagelijks rondje hardlopen.
In de eerste opgave maken we een basis-app om de gelopen route op een kaart vast te leggen. In de tweede en derde opgave gaan we de routes vastleggen, analyseren en misschien ook nog uitwisselen met anderen.

C.1 Practicum 1

Het idee
Een brainstorm met de beoogde gebruikers heeft een schets opgeleverd hoe de app er ongeveer uit moet komen te zien. Zie figuur 30 voor deze impressie. Het is nog niet zeker dat het precies zo gaat worden: tijdens de ontwikkeling kun je op ideeën komen hoe het gebruiksgemak verbeterd kan worden. Maar de hoofdzaak is:

- Het grootste deel van het scherm wordt gebruikt om een kaart te tonen
- Boven de kaart kan eventueel enige status-informatie worden getoond
- Er zijn enkele knoppen waarmee de gebruiker de app kan bedienen



Figuur 30: Schets van de userinterface van de Running app

De kaart
De kaart laat een gedetailleerd beeld van Utrecht-Oost/De Uithof zien. Met een pinch-beweging kan de gebruiker in- en uitzoomen op de kaart, tot een zeker maximum: de kaart moet niet onzinnig groot of klein kunnen worden. Met een drag-beweging kan de gebruiker de kaart verplaatsen. Ook dit moet worden begrensd tot het zinnige: de gebruiker mag de kaart niet helemaal buiten beeld slepen.

De huidige locatie van de gebruiker moet duidelijk zijn gemarkeerd: niet alleen waar hij zich bevindt, maar ook in welke richting hij kijkt. En tenslotte moet het afgelegde trainingsparcours (het 'track') in een duidelijke kleur worden getoond.

De knoppen
Er zijn in ieder geval knoppen om de volgende akties van de gebruiker mogelijk te maken:

- Het centreren van de kaart op de huidige locatie.
- Het starten en stoppen van de training. Alleen tijdens de training wordt het track opgebouwd, daarbuiten niet. Na het stoppen kan de training later weer worden hervat; het track kan dus uit meerdere losse segmenten bestaan.
- Het wissen van het track. Liefst wel met een 'are you sure?' bevestiging, anders kun je met een onhandige klik je hele training kwijtraken...

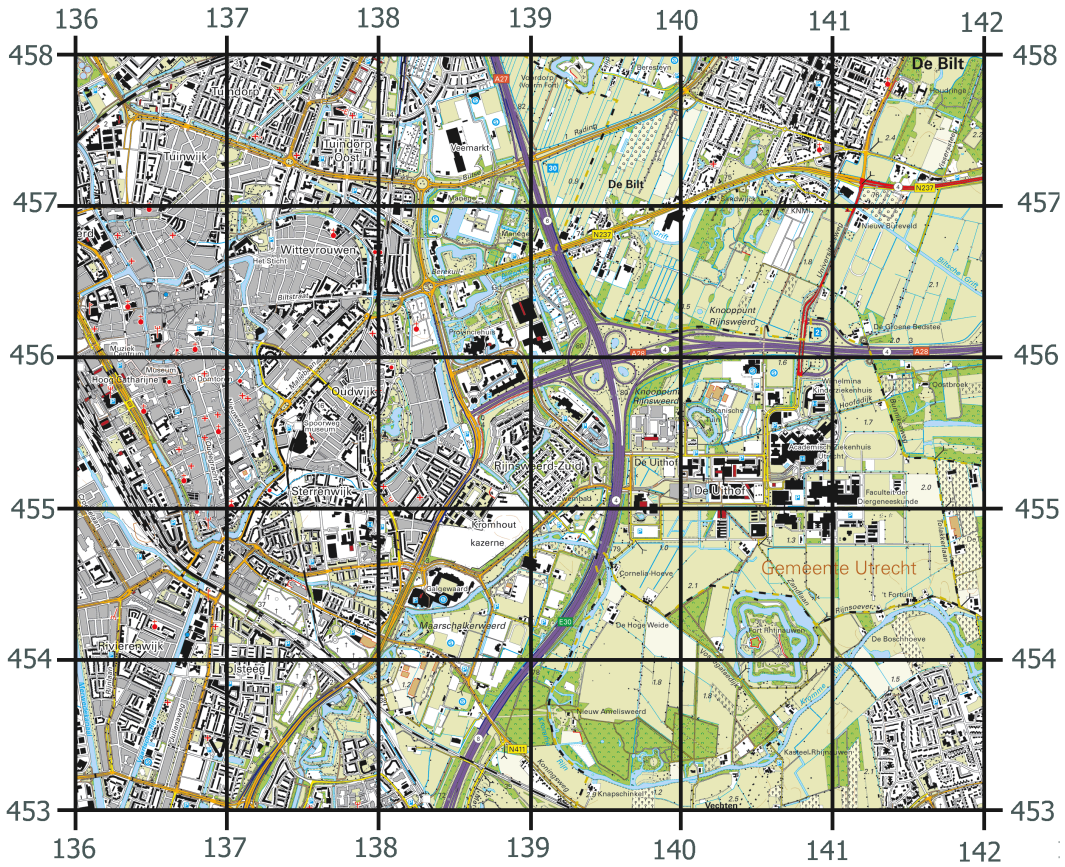
Het track
De GPS van het Android-device geeft ongeveer elke seconde de positie-informatie door. Als je dat allemaal zou loggen wordt het wel een erg lange track. Zorg er daarom voor dat een locatie alleen maar wordt opgeslagen als dat de moeite waard is (bijvoorbeeld: een redelijke afstand tot het vorige track-punt of een significante verandering van richting).

Gegeven: het kaartmateriaal
Topografisch kaartmateriaal van heel Nederland kun je downloaden op `www.pdok.nl`, als je vervolgens klikt op Producten > PDOK Downloads > Bases Registratie Topografie > TOPraster > TOPraster Actueel > TOP25raster. Je hoeft de gegevens niet zelf te downloaden: het relevante kaartfragment (samengesteld uit de bladen 31H en 32C) is beschikbaar op de practicum-webpagina. Zorg wel dat je je aan de licentievoorwaarden (CC-BY-4.0) van PDOK houdt.
De kaarten zijn gebaseerd op de topografische kaarten 1:25.000 van het Kadaster. Op papier komt daarin 4 centimeter op de kaart overeen met 1 km in het terrein. De kaarten zijn gescand met een resolutie van 100 pixels per centimeter, dus 400 pixels in de bitmap komt overeen met 1 km in het terrein.

Gegeven: de projectieformule
De GPS van het Android-device geeft de locatie in geografische coördinaten: een 'latitude' (breedtegraad, in Nederland ongeveer 52 graden) en een 'longitude' (lengtegraad, in Nederland ongeveer 5 graden). Om de bolvorm van de aarde op een kaart af te beelden worden de coördinaten *geprojecteerd* op een plat vlak.
De Nederlandse topografische kaarten gebruiken de projectie van de 'Rijksdriehoekmeting', kortweg de RD-projectie. Na projectie hebben de punten een x- en een y-coördinaat in meters. Voor het in deze opgave relevante stuk rond Utrecht loopt de x-coördinaat van 136000 tot 142000 en de y-coördinaat van 453000 tot 458000. Zie figuur 31 om je te oriënteren.
Onze bitmap van het gebied rond Utrecht beslaat een terrein van 6km breed en 4km hoog. Omdat elke kilometer in 400 pixels wordt weergegeven, is de bitmap $6 \times 400 = 2400$ pixels breed en $5 \times 400 = 2000$ pixels hoog. Let op dat de y-coördinaat van beneden naar boven loopt, dat is dus andersom dan het stelsel op een `Canvas`!
De projectieformules zijn ingewikkeld, en worden beschreven in F.H. Schreutelkamp en G.L. Strang van Hees: 'Benaderingsformules voor de transformatie tussen RD- en WGS84-kaartcoördinaten', in *Geodesia* **43** (2001), pp. 64–69. Volledige tekst van dit artikel is gemakkelijk te vinden op het internet. Je hoeft dit echter niet zelf te programmeren, want de benodigde conversieformules zijn al beschikbaar in twee methodes

```
public static PointF Geo2RD(PointF geo)
public static PointF RD2Geo(PointF rd)
```

die je kunt vinden op de practicum-webpagina.



Figuur 31: RD-coördinatenstelsel rond Utrecht

Bijlage D

Class library Mobiel Programmeren

namespace System

class **String** // alias voor string

		int	Length	
		char	this	[int]
s	operator	string	+	(string, string)
s	operator	bool	==	(string, string)
s	operator	bool	!=	(string, string)
s	static	string	Empty	
s	static	int	Compare	(string, string)
s	static	int	Compare	(string, string, StringComparison)
s	static	string	Concat	(string, string)
s	static	string	Copy	(string)
s	static	bool	Equals	(string, string)
s	static	bool	Equals	(string, string, StringComparison)
s	static	bool	IsNullOrEmpty	(string)
s	static	bool	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	()

enum **StringComparison** Ordinal, OrdinalIgnoreCase, CurrentCulture, CurrentCultureIgnoreCase

class **Object** // alias voor object

	virtual	bool	Equals	(object)
s	static	bool	Equals	(object, object)
	virtual	string	ToString	()

struct **Int32** // alias voor int

s	static	int	Parse	(string)

struct **Int64** // alias voor long

s	static	long	Parse	(string)

struct **Double** // alias voor double

s	static	double	Parse	(string)
s	static	double	Parse	(string, CultureInfo)

struct **Single** // alias voor float

s	static	float	Parse	(string)
s	static	float	Parse	(string, CultureInfo)

abstract class **Array** // methodes werken op arrays

		int	Length	
		object	Clone	()
		int	GetUpperBound	(int)

struct **DateTime**

		ⓒ	DateTime	(int y,int mo,int d)
		ⓒ	DateTime	(int y,int mo,int d,int h,int mi,int s)
		int	Year	
		int	Month	
		int	Day	
		int	DayOfWeek	
		int	DayOfYear	
		int	Hour	
		int	Minute	
		int	Second	
s	static	DateTime	Now	
s	static	DateTime	Today	
s	operator	DateTime	+	(DateTime, TimeSpan)
s	operator	DateTime	−	(DateTime, TimeSpan)
s	operator	TimeSpan	−	(DateTime, DateTime)

struct **TimeSpan**

		ⓒ	TimeSpan	(int h,int mi,int s)
		ⓒ	TimeSpan	(int d,int h,int mi,int s)
		int	Days	
		int	Hours	
		int	Minutes	
		int	Seconds	
		double	TotalDays	
s	operator	TimeSpan	+	(TimeSpan, TimeSpan)
s	operator	TimeSpan	−	(TimeSpan, TimeSpan)

static class **Math**

s	static	double	E	
s	static	double	PI	
s	static	double	Abs	(int)
s	static	double	Abs	(double)
s	static	double	Sin, Cos, Tan	(double)
s	static	double	Exp, Log, Log10	(double)
s	static	double	Pow, Atan2	(double, double)
s	static	double	Sqrt	(double)
s	static	double	Floor, Ceiling	(double)
s	static	double	Truncate, Round	(double)
s	static	double	Min, Max	(double, double)
s	static	double	Min, Max	(int, int)

delegate	void	**EventHandler**	(object, EventArgs)
delegate	void	**EventHandler⟨T⟩**	(object, T)
class **EventArgs**			
class **Random**			
	ⓒ	Random	()
	int	Next(int)	

namespace System.Globalization

class **CultureInfo**			
	ⓒ	CultureInfo	(string)
s static	CultureInfo	InvariantCulture	

namespace 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)
class **List⟨T⟩** : IList⟨T⟩			
	ⓒ	List⟨T⟩	()
	ⓒ	List⟨T⟩	(IEnumerable⟨T⟩)
	int	Capacity	

namespace Android.Content

class **Context**			
	Resources	Resources	
	object	GetSystemService	(string)
	void	StartActivity	(Intent)
s static	string	LocationService	
s static	string	SensorService	
class **Intent**			
	ⓒ	Intent	()
	ⓒ	Intent	(Context, Type)
	ⓒ	Intent	(String, Uri)
	Intent	PutExtra	(string, string)
	Intent	PutExtra	(string, int)
	Intent	PutExtra	(string, double)
	Intent	PutExtra	(string, *en nog 21 andere types*)
	string	GetStringExtra	(string)
	int	GetIntExtra	(string, int)
	double	GetDoubleExtra	(string, double)
s static	string	ActionView, ActionSend, ActionSearch, ActionCall, ...	
s static	string	ExtraText, ExtraSubject, ExtraEmail, ExtraCC, ...	

namespace Android.App

class **Activity** : Context

virtual	void	OnCreate	(Bundle)
virtual	void	OnDestroy	()
virtual	void	OnPause	()
virtual	void	OnResume	()
virtual	void	Finish	()
virtual	void	OnActivityResult	(int, Result, Intent)
	Intent	Intent	
	string	Title	
	void	SetContentView	(View)
	View	FindViewById	(int)
	T	FindViewById⟨T⟩	(int)
	void	SetResult	(Result, Intent)
	void	StartActivityForResult	(Intent, int)

enum **Result** Canceled, FirstUser, Ok

class **Dialog**

	void	Show	()

class **DatePickerDialog** : Dialog

	ⓒ	DatePickerDialog	(Context , EventHandler⟨DatePickerDialog.DateSetEventArgs⟩ ,int,int,int)

class **DatePickerDialog.DateSetEventArgs** : EventArgs

	DateTime	Date

class **AlertDialog.Builder** : Dialog

	ⓒ	AlertDialog.Builder	(Context)
	AlertDialog.Builder	SetTitle	(string)
	AlertDialog.Builder	SetNegativeButton	(string, EventHandler)
	AlertDialog.Builder	SetPositiveButton	(string, EventHandler)

namespace Android.Views

class **View**

	int	Width	
	int	Height	
	void	SetBackgroundColor	(Color)
virtual	void	OnDraw	(Canvas)
event	EventHandler⟨View.TouchEventArgs⟩	Touch	
	void	Invalidate	()

class **ViewGroup** : View

	ⓒ	ViewGroup	(Context)
	void	AddView	(View)

class **View.TouchEventArgs**

	MotionEvent	Event

class **MotionEvent**

	int	PointerCount	
	float	GetX, GetY	()
	float	GetX, GetY	(int d)

class **ScaleGestureDetector**

	ⓒ	ScaleGestureDetector	(Context, ScaleGestureDetector.IOnScaleGestureListener)
	void	OnTouchEvent	(MotionEvent)

interface **ScaleGestureDetector.IOnScaleGestureListener**

	bool	OnScale	(ScaleGestureDetector d)
	bool	OnScaleBegin	(ScaleGestureDetector d)
	void	OnScaleEnd	(ScaleGestureDetector d)

namespace Android.Widget

class **TextView** : View

	ⓒ	TextView	(Context)
	string	Text	
	float	TextSize	
	void	SetTextColor	(Color)
event	EventHandler⟨TextChangedEventArgs⟩	TextChanged	

class **Button** : TextView

	ⓒ	Button	(Context)
event	EventListener	Click	

class **CompoundButton** : Button

	bool	Checked	
	void	Toggle	()

class **RadioButton** : CompoundButton

	ⓒ	RadioButton	Context

class **CheckBox** : CompoundButton

	ⓒ	CheckBox	Context

class **SeekBar** : View

	ⓒ	SeekBar	(Context)
	int	Max	
	int	Progress	
event	EventListener	ProgressChanged	

class **EditText** : TextView

	ⓒ	EditText	(Context)
event	EventHandler⟨AfterTextChangedEventArgs⟩	AfterTextChanged	

class **LinearLayout** : ViewGroup

	ⓒ	LinearLayout	(Context)
	Orientation	Orientation	
	void	AddView	(View, LinearLayout.LayoutParams)

class **RadioGroup** : LinearLayout

	ⓒ	RadioGroup	(Context)

class **LinearLayout.LayoutParams**

	int	LeftMargin, RightMargin, TopMargin, BottomMargin	

enum **Orientation** Horizontal, Vertical

class **AnalogClock** : View

	ⓒ	AnalogClock	(Context)

class **TextClock** : TextView

	ⓒ	TextClock	(Context)
	string	Format24Hour	

class **ListView** : View

	void	SetItemChecked	(int, bool)
event	EventHandler⟨ItemClickEventArgs⟩	ItemClick	
	ChoiceMode	ChoiceMode	
	Adapter	Adapter	
	SparseBooleanArray	CheckedItemPositions	

enum **ChoiceMode** None, Single, Multiple

class **ArrayAdapter⟨T⟩** : BaseAdapter

	ⓒ	ArrayAdapter⟨T⟩	(Context, Layout, IList⟨T⟩)
	ⓒ	ArrayAdapter⟨T⟩	(Context, Layout, T[ ])

enum **Layout** SimpleListItemChecked, SimpleListItemMultipleChoice, SimpleListItemActivated1

class **BaseAdapter**

virtual	long	GetItemId	(int)
virtual	object	this	[int]
virtual	int	Count	
virtual	View	GetView	(int, View, ViewGroup)

class **Toast**

	©	Toast	(Context)
	ToastLength	Duration	
s static	Toast	MakeText	(Context, string, ToastLength)
	void	SetText	(string)
	void	Show	()

enum **ToastLength** Short, Long

namespace Android.Hardware

class **SensorManager**

	bool	RegisterListener	(ISensorEventListener,Sensor,SensorDelay)
	Sensor	GetDefaultSensor	(SensorType st)

class **Sensor**

	SensorType	Type

class **SensorEvent**

	Sensor	Sensor
	float[ ]	Values

enum **SensorType** Orientation, AmbientTemperature, Pressure, Light, Proximity, Heartrate, StepCounter, Temperature, ...

enum **SensorDelay** Fastest, Game, Normal, Ui

interface **ISensorEventListener**

	void	OnSensorChanged	(SensorEvent se)
	void	OnAccuracyChanged	Sensor s, SensorStatus st

namespace Android.Locations

class **LocationManager**

	string	GetBestProvider	(Criteria, bool)
	IList⟨string⟩	GetProviders	(Criteria, bool)
	void	RequestLocationUpdates	(string, long millisec , float meters, ILocationListener)
s static	string	GpsProvider, NetworkProvider	

interface **ILocationListener**

	void	OnLocationChanged	(Location loc)
	void	OnProviderDisabled	(string)
	void	OnProviderEnabled	(string)
	void	OnStatusChanged	(string, Availability, Bundle)

class **Critera**

	©	Criteria	()
	Accuracy	Accuracy	

enum **Accuracy** Coarse, Fine, High, Medium, Low

enum **Availability** OutOfService, TemporarilyUnvailable, Available

namespace Android.Net

class **Uri**

s static	Uri	Parse	(string)

namespace Android.OS

class **Bundle**

namespace Android.Graphics

struct **Color**

	ⓒ	Color	(int argb)
	ⓒ	Color	(int r, int g, int b)
	ⓒ	Color	(int r, int g, int b, int a)
s static	Color	ParseColor	(string)
s static	Color	Argb	(int a, int r, int g, int b)
s static	Color	HSVToColor	(float[ ])
	byte	R	
	byte	G	
	byte	B	
	byte	A	
	float	GetHue	()
	float	GetSaturation	()
	float	GetValue	()
s static	Color	AliceBlue, AntiqueWhite, Aqua, ...	
s static	Color	White, LightGray, Gray, DarkGray, Black, ...	
s static	Color	Red, Green, Blue, Yellow, Magenta, Cyan, ...	

class **Canvas**

	ⓒ	Canvas	(Bitmap)
	void	DrawRect	(float x1,float y1,float x2,float y2,Paint)
	void	DrawOval	(float x1,float y1,float x2,float y2,Paint)
	void	DrawArc	(float x1,float y1,float x2,float y2 , float starthoek, float tekenhoek, Paint)
	void	DrawLine	(float x1,float y1,float x2,float y2,Paint)
	void	DrawCircle	(float x, float y, float r, Paint p)
	void	DrawText	(string s, float x, float y, Paint p)
	void	DrawBitmap	(Bitmap b, float x, float y, Paint p)
	void	DrawBitmap	(Bitmap b, Matrix m, Paint p)
	void	DrawBitmap	(Bitmap b, Rect src, Rect dst, Paint p)
	void	DrawColor	(Color c)
	void	DrawRect	(Rect r, Paint p)
	void	DrawRect	(RectF r, Paint p)

class **Paint**

	ⓒ	Paint	()
	Color	Color	
	float	StrokeWidth	
	float	TextSize	
	void	SetStyle	(Paint.Style)
	void	SetTypeface	(Typeface)

class **Typeface**

	ⓒ	Typeface	(string, TypefaceStyle)

enum **TypefaceStyle** Bold, BoldItalic, Italic, Normal

enum **Paint.Style** Stroke, Fill

class **Rect**

	ⓒ	Rect	(int left, int top, int right, int bottom)
	int	Left, Top, Right, Bottom	
	bool	IsEmpty	
	bool	Contains	(Rect)
	bool	Contains	(int x, int y)
	int	Width	()
	int	Height	()
	bool	Sort	()
	void	Union	(Rect)
	void	Union	(int x, int y)
	void	Inset	(int x, int y)

class **Point**

	ⓒ	Point	(int x, int y)
	int	X, Y	
	void	Offset	(int dx, int dy)

class **RectF** // als Rect, maar dan met float in plaats van int

class **PointF** // als Point, maar dan met float in plaats van int

class **Bitmap**

	int	Width	
	int	Height	
	int	GetPixel	(int x, int y)
	void	SetPixel	(int x,int y, Color c)

class **BitmapFactory**

s	static	Bitmap	DecodeResource	(Resources, int)
s	static	Bitmap	DecodeResource	(Resources,int,BitmapFactory.Options)

class **BitmapFactory.Options**

	ⓒ	BitmapFactory.Options	()
	bool	InScaled	
	int	InSampleSize	

class **Matrix**

	void	PostTranslate	(float dx, float dy)
	void	PostScale	(float sx, float sy)
	void	PostRotate	(float d)

namespace Android.Util

class **SparseBooleanArray**

	int	Size	()
	int	KeyAt	(int)
	bool	ValueAt	(int)

namespace SQLite

class **SQLiteConnection**

	ⓒ	SQLiteConnection	(string)
	void	CreateTable⟨T⟩	()
	TableQuery⟨T⟩	Table⟨T⟩	()
	void	Insert⟨T⟩	(T)
	void	Update⟨T⟩	(T)
	void	Delete⟨T⟩	(T)

class **TableQuery⟨T⟩** : IEnumerable⟨T⟩

	TableQuery⟨T⟩	Where	(Expression)