**Aantekeningen bij Objective C cursus**

**Hoofdstuk 0 Introduction**

Objective C is de taal om apps te bouwen voor MacBook (OSX), iPhone en iPad (iOS). De cursus gaat o.a. in op het Foundation framework, een verzameling van code snippets die in elk programma kan worden gebruikt. En op XCode4, de meest recente versie (eind 2011) van de geintegreerde ontwikkelomgeving van Apple.

Benodigde voorkennis voor deze cursus bestaat uit de weg weten op de Mac en wat programmeerervaring. Ervaring met Objective C is niet noodzakelijk. Objective C is echter afwijkend van talen als Java en C++, dat gaat verder dan enkel de syntax; Objective C heeft zich in de loop van de jaren op een eigen manier ontwikkeld en er zijn afwijkende concepten in de taal geintroduceerd.

Bij de cursus zijn oefenbestanden met code te vinden, deze zijn per hoofdstuk onderverdeeld. De bestanden kunnen tekstfiles zijn die 1 op 1 in XCode gekopieerd kunnen worden. Er zijn ook projectfiles met als extensie *.xcodeproj*. Door hierop te dubbelklikken wordt XCode geopend en alle bij het project behorende bestanden worden automatisch geopend.

**Hoofdstuk 1 Getting started**

Ontwikkelen van programma’s in Objective C doe je met XCode, in feite de enige ontwikkelomgeving hiervoor. Om XCode te verkrijgen moet je je als ontwikkelaar (gratis) aanmelden bij Apple, XCode kan dan gratis gedownload worden. Wil je een app in de Apple App Store gaan verkopen (of uitrollen op een device tbv testen), dan moet je je aanmelden voor het Apple Developer program, dat kost 99$ per jaar.

Ga hiervoor naar <developer.apple.com>. Hier is heel veel technische technische informatie te vinden. De site betaat uit 3 onderdelen:

* iOS Dev Center, voor iPhone en iPad
* Mac Dev Center, voor Mac desktop programma’s
* Safari, voor ontwikkeling van Safari extensions

Als je bent aangemeld als Developer, log dan in en navigeer naar de pagina waar je XCode4 kunt downloaden (ruim 4 GB!). Installeer vervolgens XCode, dit wordt niet in de Applications (Programma’s) folder geinstalleerd maar in de folder Developer\Applications. Laat dit zo! Wanneer je het XCode programma ergens anders heen sleept (bijv desktop) is de kans groot dat je problemen daarvan gaat ondervinden. NB In deze folder staat veel meer dan XCode alleen.

We starten met de standaard Hello World applicatie. Start XCode en selecteer ‘Create a new project’. XCode organiseert code in projecten, en slaat deze in aparte folders op. XCode creeert ook de projectfile (met extensie .xcodeproj) waarin allerhande projectinformatie wordt opgeslagen. Wordt in de Finder op een dergelijke file dubbel geklikt dan wordt het betreffende proejct in XCode geopend.

XCode kent diverse soorten projecten, zoals het XCode ‘Create new project’ scherm laat zien. We starten met het eenvoudigste project type van de Command Line application. Selecteer hiertoe Application (onder Mac OSX) en in het scherm rechts Command Line Tool. Selecteer Next en vul de naam van het nieuwe project in (bijv. FirstApp), in het Product veld. Selecteer by ‘type’ Foundation. Klik op Next en selecteer de folder waar het project moet worden opgeslagen. Uncheck de box ‘create local git repository for this project’, we gaan nl nog geen gebruik maken van source control. Klik op Create en het XCode project scherm verschijnt.

Links in dit scherm is de Navigator afgebeeld, dit laat alles van het project zien: files, settings, configuratie opties etc in een folder structuur. Selecteer nu de ‘main.m’ in de Navigator, deze source file staat juist onder de naam van het project. Rechts verschijnt nu het editor scherm.

Bovenin het scherm zijn diverse run buttons opgenomen, waarmee het programma op diverse manieren gerund kan worden. In het midden van dit scherm is een soort console schermpje waarin XCode runtime messages laat zien van de draaiende applicatie. Rechts bovenaan het scherm zijn 2x3 buttons opgenomen, waarmee het scherm snel een andere indeling kan worden gegeven en bijv andere aan dit project gerelateerde windows opgevraagd kunnen worden.

We concentreren ons nu op de main.m sourcefile. Dit bevat ons programma. Druk nu op de Run knop links boven aan het scherm. Rechts onderin het scherm verschijnen nu run-time meldingen van het gestarte programma (Hello world in dit geval). In de code is dit in het NSLog statement terug te vinden (a la printf in C maar dan veel intelligenter). De betekenis van de statements zal in een volgend hoofdstuk worden behandeld.

In de XCode folder Developer\Applications staan een aantal applicaties waarvan XCode veruit de belangrijkste is. Dashcode is voor het ontwikkelen van web based applicaties en dashboard widgets. Quartz Composer is voor het ontwikkelen van grafische applicaties. Beide worden in deze cursus niet behandeld. Instruments is een app die diagnostische informatie geeft over programma’s, zoals CPU en geheugengebruik.

In de cursus wordt een nieuw project SecondApp gecreeerd. Er worden een aantal algemene XCode settings aangepast, selecteer hiertoe Preferences in het XCode menu. In de tab Fonts&Colors wordt ‘Presentation’ geselecteerd (wat het font o.a. een stuk groter maakt). In de tab Tekst&Editing wordt Line Numbers aangevinkt, zodat elke code regel door een regelnummer wordt voorafgegaan.

XCode biedt velerlei ondersteuning bij het coderen, zoals automatic indentation, syntax afhankelijke kleuren (zodat bijv. makkelijk onderscheid kan worden gemaakt tussen code, commentaar en variabelen). XCode biedt ook Code Sense, de auto completion feature bij het intikken van code. Hiermee verschijnt gedurende het typen bijv. automatisch een lijst van bibliotheek functies, waaruit vervolgens eenvoudig de juiste geselecteerd kan worden. XCode geeft ook direct gedurende het typen al hints ten aanzien van de syntactische correctheid van wat wordt ingetypt.

Een project kan worden getest door de Run knop links boven aan het XCode scherm in te drukken. De code wordt dan ge-build en ge-rund. Dit kan ook door in het Product menu de opties Build en vervolgens Run te selecteren. In het dashboard-achtige schermpje boven in het XCode window verschijnen diagnostische meldingen, bijv. over de afloop van de build.

**Hoofdstuk 2 Objective C Basics**

**2.1 De Objective C taal**

C stond aan de oorsprong van Objective C. In de 80-er jaren werd SmallTalk ontwikkeld, de eerste object georienteerde programmeertaal. Objective C ontstond in 1986, is volledig gebaseerd op C en bevat de nodige object georienteerde toevoegingen (zoals classes, objects, messaging etc). Andere object georienteerde talen als C++, Java en C# zijn weliswaar gebaseerd op C, maar elk vanaf de grond af ontwikkeld en hebben ieder hun eigen identiteit. Dit betekent dat jaren ‘70 C code zonder meer correct compileert in Objective C, terwijl dat bij Java en C# niet zal lukken.

De uitbreidingen die Objective C biedt tov. C zijn door in de syntax terug te vinden, zoals de blokhaken [ en ], de compiler directives voorafgegaan door @ (bijv @interface, @implementation, @property, @synthesize) en diverse typenamen voorafgegaan door NS (van NextStep). Mac OS X is gebouwd op Objective C, evenals iOS.

**2.2 De structuur van een Objective C programma**

Aan de hand van het default Hello World programma dat XCode creeert bij aanmaken van een project wordt de structuur van een Objective C programma behandeld.

Commentaar in Objective C wordt voorafgegaan door 2 slashes //.

De regel #import <Foundation/Foundation.h> geeft het programma de beschikking over het Foundation framework. Dit bestaat uit een 100-tal files met code die het werken met strings, arrays, etc etc eenvoudiger maakt. In vrijwel elk Objective C programma kom je deze regel tegen.

Het startpunt van elk Objective C programma is de main() routine, vrijwel identiek aan C.

Codeblokken beginnen met een { en eindigen met een }.

Objective C is een case sensitive programmeertaal. Dus de variabelen iCount en icount zijn in Objective C verschillende variabelen!

Het eerste statement in vrijwel elk Objective C programma is:

NSAutoreleasePool \* pool = [[NSAutoreleasePool alloc] init];

Hiermee alloceert de applicatie geheugen. NB In de versie 4.2.1 van XCode die ik gebruik (maart 2012) is deze regel vervangen door @autoreleasepool, met kennelijk hetzelfde effect.

De functie NSLog is vergelijkbaar met printf in C.

Het statement [pool drain] geeft het door het programma gealloceerde geheugen weer vrij.

Het laatste statement (return 0) geeft de returncode van het programma, in dit geval is het programma succesvol afgesloten.

De hier genoemde NS… functies maken deel uit van het Foundation framework, dat technisch gezien geen onderdeel is van de taal Objective C maar een uitbreiding daar op.

**2.3 Compileren en runnen van code**

De Build knop doet zowel compileren als runnen. Voor de compilatie wordt de LLVM compiler gebruikt (is echter instelbaar in XCode settings). De ‘Build success’ melding verdwijnt even snel als hij verscheen. Door op de Log Navigator knop te drukken (7e knop van links boven aan het scherm, lijkt op een tekstballon) kan de informatie teruggehaald worden. De linker knop in dit rijtje zal het meest gebruikt worden, die geeft de projectstructuur weer. De middelste Issue Navigator knop geeft informatie over problemen met code in source files.

Fouten in de source code worden na het builden weergegeven in de code zelf (regel kleurt lichtrood) en in het dashboard schermpje bovenin het XCode window. In de Issue Navigator kunnen de problemen een voor een opgezocht worden. XCode zal zelf waar mogelijk voorstellen doen om de fouten te corrigeren, dmv de FixIt functie. Die werkt al tijdens het intypen van de code. Ook warnings worden direct tijdens het typen weergegeven, bijv. voor een variabele die is gedefinieerd maar niet in de code wordt gebruikt.

In Xcode Preferences menu kan het gedrag (Behaviour tab) van XCode bij compileren, builden, testen etc. ingesteld worden. Een voorbeeld is het tonen van de Issue Navigator als er issues bij het builden optreden.

Links boven in het XCode scherm wordt het ‘scheme’ getoond dat gebruikt wordt voor de code generatie. In ons voorbeeld staat hier ‘ My Mac 64-bit’ . Afhankelijk van het soort project kunnen hier andere keuzemogelijkheden getoond worden, bijv de het aangesloten apparaat (iPhone, iPad) of de simulator in geval van een iOS project.

Wordt in de Project Navigator (icon links boven in XCode scherm) het project zelf geselecteerd (eerste regel), dan verschijnen allerlei instellingen mbt compileren, builden en de project-applicatie zelf. Voorbeelden van Build Settings zijn de Base SDK (bij mij: OS X 10.7) en de Deployment Target (bij mij: OS X 10.7), die niet gelijk hoeven te zijn. De belangrijkste info is ook in het Info tabblad te zien.

**Hoofdstuk 3 Program Flow**

**3.1 Het loggen van berichten op de commandline**

Variabele declaraties in de vorm van

int a = 3;

int b = 4;

int c = a \* b;

Logging naar de console mbv NSLog:

NSLog(@”a = %i, b = %i en a\*b = %i”, a, b, c);

Het @ teken is voor de compiler een indicatie dat er een ‘Objective C type’ string moet worden afgedrukt. %i wordt een ‘placeholder’ genoemd. Andere veel gebruikte zijn %f voor float en %c voor character.

**3.2 Het schrijven van conditionele code**

Testen op gelijkheid in if statement:

if (a == 5) {

// statements hier

}

Let op het dubbele = teken, een enkel = teken is een assignment waarmee de conditie altijd waar is!

Testen op ongelijkheid: if (a != 5) // code etc.

Testen op meerdere condities:

AND: if (a!=5 && b==100) // code etc

OR: if (a!=5 || b==100) // code etc

Het else blok:

If (a != 5 && b==100) {

// statements indien conditie waar is

} else {

// statements indien conditie niet waar is

}

Binnen de ‘waar’ en ‘niet waar’ blokken kunnen wederom geneste if statements opgenomen worden.

**3.3 Het switch statement**

Wanneer een variabele op diverse, verschillende waarden getest moet worden wordt dat onoverzichtelijk wanneer met if-else statements wordt gewerkt. Voor deze situatie is het switch statement ontworpen. Syntax:

switch (a) {

case 1: // statement block

break;

case 2: // statement block

break;

default: // statement block

break; // break is hier optioneel omdat dit de laatste keuze van de switch is

}

Het break statement zorg ervoor dat het switch block verlaten wordt. Zonder break zou het eerstvolgende statement in de code, dwz. de volgende case, worden uitgevoerd, **zonder dat op de waarde wordt getest**. Dit is omdat Objective C lijdt aan ‘fall-through’, de break is nl niet verplicht (in tegenstelling tot bijv C#). In feite een onhandigheidje van Objective C, het kan nl leiden tot onvoorspelbaar gedrag, waarbij de fout moeilijk opspoorbaar is.

**3.4 Code snippets**

Code snippets kunnen opgezocht worden in de Utilities sectie, die rechts op het scherm verschijnt na aanklikken van het ‘Hide or Show Utilities’ icon rechts boven aan het XCode scherm. Klik onder aan het Utilities window op het {} icon. Kies de gewenste code snippet en sleep het naar de gewenste plek in de editor. Met de Tab toets kan worden gesprongen naar de invoervelden in de code snippet, bijv om de verschillende elementen van een if-else statement in te vullen.

De opmaak moet dan nog aangepast worden. Doe een rechterklik (eventueel met CTRL) en selecteer Structure en vervolgens Re-Indent uit het verschenen menu. Hetzelfde effect kan met CTRL-I bereikt worden.

Eigen code snippets kunnen ook aan de library toegevoegd worden. Selecteer hiervoor een blok code en sleep het naar de code snippet library. Een user defined code snippet kan weer worden verwijderd door deze te selecteren en op de Backspace key te drukken.

**3.5 Operatoren en expressies**

Standaard arithmetische operators: + - \* /

Assignment operator: =

Shorthands: a += 5 voor a = a+5, ook voor de andere operators - \* /

Operator volgorde: \* en / gaan voor + en –

Vergelijking: == != < > <= >=

Logische AND en OR: && ||

Modulus: % geeft het restant na een deling van 2 integers: 13%5 = 3

Increment en decrement: ++ --, kunnen voor en achter variabele gezet worden, de waarde wordt dan voor resp. na het gebruik verhoogd of verlaagd. Bij voorkeur niet in expressies gebruiken maar alleen als enkel statement, vanwege buggevoeligheid.

NB Dit is een unaire operator, itt de andere binaire operators.

Ternary operator: condition ? true : false, is een shortcut voor if-else statement. Voorbeeld: a = (b>c) ? b : c

Lekker compact programmeren zo dus.

Verder nog bitwise operators, die hier niet behandeld worden vanwege het geringe belang ervan.

**3.6 Loops**

While loop:

a = 1;

while (a < 10) {

// statements

a++;

}

Do while loop:

a = 1;

do {

// statements

a++;

} while (a < 10);

NB Do while loop wordt minimaal een keer doorlopen. Omdat de conditie bij do while loops aan het eind van de loop staat, is die vorm daarmee wat onoverzichtelijker. Voor een beter begrip is het ook handiger de conditie aan het begin van de loop te testen, omdat de code in de loop lang niet altijd minimaal een keer moet worden uitgevoerd. In de praktijk wordt de while do loop daarom veel meer gebruik dan de do while loop.

De for loop is in feite een syntactisch aangepaste while do loop:

for (i=1; i<=100; i++) {

// statements

}

Het **break** statement zorgt ervoor dat de loop direct wordt verlaten zodra dit statement wordt bereikt. Voorbeeld:

for (i=1; i<=100; i++) {

// statements

if (i == 23) {

break;

}

}

Zodra i gelijk is aan 23 wordt de for loop verlaten, net zoals wanneer i de waarde 100 heeft aangenomen.

Het **continue** statement zorgt ervoor dat de resterende statements van een iteratie niet meer worden uitgevoerd, er wordt direct teruggesprongen naar het begin van de loop. Voorbeeld:

for (i=1; i<=100; i++) {

// statements

if (i == 23) {

continue;

}

// nog meer statements

}

De statements na de test (i == 23) worden normaal gesproken altijd uitgevoerd, behalve wanneer i gelijk is aan 23, in welk geval direct wordt teruggesprongen naar het begin van de loop.

Break en continue zijn feitelijk overbodig en worden daarom niet zeer veel gebruikt.

De while do, do while en for loop zijn de drie essentiele loops. Er zijn nog andere die later aan bod komen.

**3.7 Functies**

Syntax voor een functie met 2 int parameters die een waarde teruggeeft:

int myFunction( int a, int b) {

// statements

result = a\*b;

return result;

}

Functie zonder parameters die geen waarde teruggeeft ziet er als volgt uit:

Void myFunction( void ) { OF void myFunction() {

// statements

return; // return mag igv void functie ook worden weggelaten

}

Zodra het return statement wordt uitgevoerd wordt de functie verlaten, ook als de return midden in de functie voorkomt.

In Objective C moeten de functies bekend zijn voordat main() wordt aangeroepen, anders faalt de build (i.c. de compilatie) van het programma. Dit kan worden gedaan door de volledige functie (incl body) voor main in de code op te nemen. Handiger en gebruikelijker is om de beschrijving (het **prototype** genoemd) van de functie (dus zonder body) voor main op te nemen:

void myFunction( void );

Int main( int argc, const char \* argv[]) {

// statements

return 0;

}

void myFunction( void ) {

//statements

return;

}

Vaak wordt een functie gecreeerd als een stuk reeds ingetikte code vaker moet worden gebruikt. XCode kan hier bij helpen dmv de **refactor** functie. Selecteer hiervoor een blok code, doe een rechtsklik en selecteer refactor en vervolgens Extract… uit het menu dat verschijnt. Er wordt daarna om een functienaam gevraagd etc. De situatie voor en na het refactor-en wordt getoond en indien gewenst kan een snapshot worden opgeslagen.

**Hoofdstuk 4 Variabelen**

**4.1 Data types**

In Objective C is iedere variabele van een bepaald type dat niet verandert gedurende het bestaan van de variabele. Dit in tegenstelling tot bijv. Javascript, waar een variabele tijdens zijn bestaan van type kan veranderen.

Variabele namen zijn case sensitive in Objective C: ab en aB zijn dus verschillende variabelen. Vwb. de naamgeving is het gewoonte om variabele namen te starten met een lowercase letter, en ieder nieuw woord binnen de variabele naam te starten met een hoofdletter, zoals in ditIsEenVoorbeeldVariabeleNaam. Underscores en cijfers worden niet of nauwelijks gebruikt. Deze naamgeving wordt door Apple zelf gebruikt, dus is handig om aan te leren wanneer bijv. technische documentatie (van Apple bijv) wordt doorgenomen.

Basis data typen (primitive types):

int: 4 bytes lang

float

double: 2x grootte van een float

char: enkel karakter

BOOL: dit is in Objective C toegevoegd, is onder de oppervlakte een char

Dit is de basis. Andere data typen, zoals string en date, zijn op deze basis data typen gebaseerd en zijn dus geen in Objective C ingebouwde data typen. Apple heeft deze en nog vele andere (bijv NSString, NSDate) in het Foundation framework ingebouwd (classes, zie hoofdstuk 5).

**4.2 Werken met getallen**

Numerieke data typen:

integer: 4 bytes, gehele getallen range van -2.147.483.648 t/m 2.147.483.647

unsigned int: idem, range van 0 t/m 4.294.967.295

long int: bij 32 bits omgeving ook 4 bytes (= integer), op 64 bits omgeving 8 bytes

long long int: 8 bytes integer op 32- en 64-bits omgeving

short int: 2 bytes

NB Wanneer de type modifier wordt gebruikt, mag ‘int’ in het type worden weggelaten. Dit is echter niet erg duidelijk.

float: 4 bytes floating point

double: 8 bytes floating point

Bij initialisatie van een float is het handig om een ‘f’ achter de initialisatiewaarde te zetten, omdat Objective floating point constanten standaard als double worden aangemerkt door de compiler. Voorbeeld:

float myFloat = 7.2f

double myDouble = 7.2

De placeholder in de NSLog voor floats is %f (6 decimale posities) en %e (exponentiele notatie)

NSLog(@”De waarde van myFloat is %f”, myFloat) geeft 7.200000

NSLog(@”De waarde van myDouble is %f”, myDouble) geeft 7.200000e+00

Bij berekeningen kan gebruik worden gemaakt van variabelen van verschillende typen:

int a = 25;

int b = 2;

float result = a / b;

NSLog(@”Het resultaat is %f”, result);

Dit geeft als resultaat 12.000000. Dit komt omdat in de expressie 2 integers op elkaar worden gedeeld, waardoor Objective C het resultaat ook een integer maakt. Dit kan worden gecorrigeerd dmv. de cast operator:

float result = (float) a / b;

Dit geeft als resultaat 12.500000. Het is voldoende om 1 van beide variabelen te type casten.

Voor grotere getallen en bijv. berekeningen met munteenheden, waar de precisie erg belangrijk is, heeft Apple classes ontwikkeld waar dat mee kan. Dit volgt later in deze cursus.

**4.3 Werken met karakters**

Karakter initialisatie:

char aChar = ‘A’; // let op: single quote gebruiken

Gebruik placeholder %c in NSLog:

NSLog(@”De waarde van aChar = %c“, aChar);

Het type BOOL is een uitbreiding in Objective C (tov van C):

BOOL aBool = YES; // BOOL, YES en NO altijd uppercase

Dus geen True of False voor booleans in Objective C gebruiken.

In een conditionele expressie hoeft niet de vergelijking met YES of NO gemaakt te worden:

if (aBool) { }

of

if (!aBool) {}

volstaat om de waarde van boolean variabele aBool te testen.

De C taal kende oorspronkelijk geen boolean type. Daar is zo’n 10 jaar geleden verandering in gekomen. Het type boolean maakt nu onderdeel uit van C, een boolean variabele kan de waarden true en false aannemen. Omdat Objective C een uitbreiding is op C, kan het type boolean (kleine letters) dus ook gebruikt worden. Dit wordt echter afgeraden, aangezien alle Foundation classes gebruik maken van het BOOLEAN type.

Er is geen specifieke placeholder voor BOOLEAN variabelen, gebruik %i hiervoor bij gebrek aan beter. Een NSLog message zal dan de waarden 1 (YES) of 0 (NO) weergeven.

**4.4 Scope van variabelen**

Standaard bestaan variabelen alleen binnen het statement block waarbinnen ze zijn gedefinieerd.

Standaard worden variabelen by value doorgegeven aan functies. Verandert de functie de parameterwaarde, dan houdt de variabele die als parameter was doorgegeven zijn oorspronkelijke waarde.

Moeten variabelen worden gedeeld tussen verschillende functies, dan moeten deze globaal worden gedefinieerd, dus ook buiten main().

**4.5 Enumeraties**

Wanneer het gewenst is dat een variabele een beperkt aantal waarden kan aannemen, kan deze als enum gedeclareerd worden. Eerst moet het enum type gedefinieerd worden met de enumeratie waarden:

enum stoelVoorkeur { gang, midden, raam };

De waarden die gebruikt kunnen worden kunnen geheel naar eigen inzicht ingegeven worden (Objective C vertaald die naar een volgnummer).

Is dit gebeurd dan kan een variabele worden gedefinieerd:

enum stoelVoorkeur marcellaStoelVoorkeur = raam;

Er is geen placeholder voor enums. Bij weergeven van de waarden in een NSLog message %i gebruiken. De eerste enum waarde krijgt standaard de waarde 0, de tweede is 1 etc. Het is ook mogelijk om getallen naar keuze te koppelen aan de enum waarden:

enum stoelVoorkeur { gang = 10, midden = 20, raam = 30 };

**4.6 Het gebruik van typedef**

Met typedef kunnen eigen typen gecreeerd en benoemd worden. Voorbeeld:

typedef enum { gang = 10, midden = 20, raam = 30} stoelVoorkeur;

Enumeratie variabelen kunnen nu eenvoudiger gedefinieerd worden:

stoelVoorkeur marcellaStoelVoorkeur = raam;

**4.7 Preprocessor directives**

Drie veel (meest) gebruikte preprocessor directives:

* #import <Foundation/Foundation.h>

importeert de code in de file tussen de haken een op een in de code

* #define MAXINT 32767

definieert de constante MAXINT en stelt deze gelijk aan 32767. Er zijn in de Foundation library vele constanten op deze manier gedefinieerd, ook voor MAXINT (heet INT\_MAX) etc.

* #if PREP\_DIRECTIVE

// programmacode

#endif

Dit zorgt er voor dat code conditioneel wordt gecompileerd, afhankelijk van de waarde van PREP\_DIRECTIVE. Een bekend vorbeeld is de directive DEBUG. Deze wordt afhankelijk van het gekozen ‘scheme’ al dan niet gezet. Het scheme kan worden benaderd door links boven in het XCode scherm op de programmanaam te klikken en Edit scheme… te selecteren. In het window dat opent kan de build configuratie worden aangepast: debug of release.

**4.8 Werken met strings**

Objective C heeft geen ingebouwd type voor strings. Belangrijkste reden is dat dit ook niet in C zit, waar Objective C op gebaseerd is. In C is een string een character array en daar is niet zo eenvoudig mee te werken. Daarom is in de Foundation library een NSString klasse gedefinieerd:

NSString \*msg = @”Hello”;

Het Foundation framework kan in XCode worden bekeken door in de Project Navigator links op het XCode scherm Frameworks en vervolgens Foundation.framework te selecteren. Hier staan alle files die mbt #import <Foundation/Foundation.h> geimporteerd worden. Ook de NSString klasse staat er gedefinieerd (in NSString.h).

Het @ teken zegt de compiler dat de string die volgt van het type NSString is. Wordt deze vergeten, dan ontstaat een C type string, wat in Objective C in deze context niet gaat werken.

De variabele msg is een pointer naar een object van het type NSString. In Objective C worden alle objecten gerefereerd mbv pointer variabelen.

De placeholder voor alle objecten is %@, dus om de waarde van msg af te drukken wordt het volgende NSLog statement gebruikt:

NSLog(@”De waarde van msg is %@”, msg);

Bij het toekennen van een waarde aan msg moet het pointer teken worden weggelaten:

NSString \*msg;

msg = @”Hello”;

De compiler weet immers al dat msg een pointer is van het type NSString.

**Hoofdstuk 5 Classes**

**5.1 Introductie in object orientatie**

Een klasse is een soort blueprint van iets (het idee), een object is een instantie (representatie) van de klasse (het ding zelf). Bijv. de klasse NSStrings geeft eigenschappen van strings en methoden die erop kunnen worden toegepast.

NSString \*msg = @”Hello world”;

msg is een object van de klasse string.

Aspecten van klassen (APIE):

* **A**bstraction is het definieren van gemeenschappelijke eigenschappen/methoden voor verschillende objecten die tot dezelfde klasse behoren
* **P**olymorfisme
* **I**nheritance
* **E**ncapsulation is het inkapselen van methoden en eigenschappen in een klasse

Aandacht voor abstraction en encapsulation. Polymorfisme en inheritance worden niet behandeld.

**5.2 Objecten en pointers**

Alle objecten worden via pointers benaderd, omdat objecten complexer in elkaar zitten dan primitive types. De grootte is daarom nooit gelijk en kan bovendien veranderen tijdens gebruik van objecten. Pointers zijn daarom handiger in gebruik.

NB de \* kan aan de klasse, variabele of los worden gepresenteerd:

* NSString\* myString
* NSString \*myString
* NSString \* myString

Syntactisch zit hier geen verschil tussen.

NB De \* alleen gebruiken bij de declaratie van een variabele, daarna niet meer nodig omdat Objective C weet dat het een pointer is. Verder alleen nog nodig als parameter definitie in functie definities.

**5.3 Messages en methoden**

Aanroep van methoden in Objective C dmv vierkante haken:

[myObject someMethod];

…en dus niet in de vorm van myObject.someMethod() zoals in andere talen als java.

Dit heeft tot gevolg dat het bericht ‘someMethod’ naar object ‘myObject’ wordt gestuurd.

Als someMethod een waarde teruggeeft dan wordt deze als volgt opgevraagd:

result = [myObject someMethod];

Als één parameter moet worden meegegeven dan ziet dat er als volgt uit:

result = [myObject someMethod:arg1];

Moeten er meerdere parameters worden meegegeven dan gaat het er anders uitzien, zoals in dit praktijkvoorbeeld:

[myObject insertString: @”Hello” atIndex:11];

Zeker met meer parameters gaat het er wat onoverzichtelijk uitzien, maar alle parameters worden bij naam aangeroepen, waardoor het leesbaarder wordt (betekenis parameterwaarden is duidelijk) en de kans op het maken van fouten in de parameteraanroep verkleind wordt. Voorbeeld:

[NSFileManager replaceItemAtURL: withItemAtURL: backupItemName:

options: resultingItemURL: error: ];

Een nog extremer voorbeeld vormt de klasse NSBitmapImageRep, die 11 parameters kent…

Tot slot de aanroep van geneste methode calls:

[myObject someMethod:[anotherObject anotherMethod]];

Meer dan 2 niveaus diep nesten uiteraard ook mogelijk op soortgelijke wijze.

**5.4 Bestaande klassen uit de Foundation Framework gebruiken**

Project aanmaken in XCode, selecteer type Foundation (niet Core Foundation). Doordat dan alle bestaande standaard classes meegelinkt worden met de project code, krijgt het project de beschikking over deze standaard klassen zoals NSString, NSLog, NSCalendar, etc…

Foundation library zit vol met standaard klassen. Hoe vind je wat je nodig hebt?

Dmv **codesense** (automatisch aanvullen van woorden in XCode) kom je al een eind. Nog meer info is op te vragen in de Utilities section (2e button van rechts rechtsboven in XCode scherm). Klik vervolgens op de Quick Help button met de golvende lijntjes, de Quick Help verschijnt dan. In het voorbeeld wordt de NSString Class Reference opgevraagd. Hierin staat veel informatie, waaronder alle methoden die op strings kunnen worden losgelaten. In de les wordt in de sectie Changing Case de methode uppercaseString gebruikt. Dit is een methode die een pointer naar een NSString object teruggeeft. Het ‘-‘ teken voor de naam uppercaseString geeft aan dat het een **instance** methode is, die op een instantie van een NSString object werkt (in tegenstelling tot een class method die geen object nodig heeft). De voorbeeldcode is als volgt:

NSString \*msg1 = @”Hello”;

NSString \*msg2 = [msg1 uppercaseString];

NSLog(@”Het resultaat is **%@**”, msg2); 🡺 HELLO

De placeholder **%@** wordt gebruikt om objecten weer te geven met NSLog.

Het volgende voorbeeld betreft de NSDate class. Hier wordt een object variabele myDate van het type NSDate gecreeerd. Het opvragen van hoe deze variabele moet worden geinitialiseerd (datum format) kan worden opgevraagd door de Quick Help op te roepen. Dit kan snel door met de muis op NSDate te gaan staan, en vervolgens de Options (Alt toets in VMWare) in te drukken, waarna een window verschijnt van waaruit de Class Reference kan worden opgevraagd.

In het voorbeeld wordt myDate met de huidige datum geinitialiseerd:

NSDate \*myDate = [NSDate date];

Er is een principieel verschil tussen de initialisatie van msg2 en myDate. Bij msg2 wordt een **instance method** toegepast (uppercaseString op het object msg1). Bij myDate wordt de **class method** date toegepast op klasse NSDate. In de methode definitie is te herkennen of een instance danwel een class method wordt gedefinieerd wanneer deze wordt voorafgegaan door een **‘-‘** resp. een **‘+’**.

**Hoofdstuk 6**

**6.1 Memory management**

Basisregel: degene die objecten creeert, reserveert geheugen daarvoor en geeft dit weer vrij wanneer het object niet meer gebruikt wordt. Objective C heeft een garbage collector die automatisch geheugen van niet gebruikte objecten vrijgeeft. Deze is enkele jaren geleden aan Objective C toegevoegd. In deze cursus wordt die niet gebruikt:

* Memory management is in de eerste plaats de verantwoordelijkheid van de programmeur, een ‘basic skill’;
* Garbage collection niet mogelijk op iPad en iPhone en op oudere OSX 10.x versies van vóór Leopard, hier moet het geheugen manueel beheerd worden;
* Garbage collection is niet de default instelling in Objective C, maar moet expliciet afgedwongen worden dmv project settings (in XCode) en eigen code;
* Om garbage collection te kunnen gebruiken is kennis van memory mgt nodig.

Memory mgt in Objective C is gebaseerd op reference counting. Wanneer een object gecreeerd wordt krijgt deze de **retain count** 1. Het object wordt gebruikt etc en released wanneer het niet meer nodig is. Bij een release wordt de retain count met 1 verlaagd waardoor het 0 wordt. Dat is het sein voor Objective C om het geheugen in gebruik door het object vrij te geven.

Het wordt wat ingewikkelder als objecten worden meegegeven aan een functies. Die functie kan een retain uitvoeren op het meegegeven object waardoor de retain count met 1 wordt verhoogd. Is de functie klaar dan doet die, vlak voor de return, een release. Dit werkt allemaal goed als deze regels strikt worden gevolgd, anders kan er een of een ‘dangling pointer’ ontstaan (verwijzing naar een niet bestaand object) of een memory leak (object retain count > 1 terwijl object niet meer gebruikt wordt). Memory leaks kunnen de reden zijn van iPad en iPhone apps die crashen, wanneer dat maar vaak genoeg optreedt.

Basis regel: de eigenaar van een object moet deze vrijgeven. Eigenaar is degene die één van de volgende object operaties uitvoert:

* Alloc
* New
* Copy
* Retain

Tegen deze operatie moet een release zitten.

Aan de andere kant: doe je deze operaties niet, dan geen release uitvoeren.

**6.2 Object Creation**

Voorbeeldcode voor creeren object:

NSDate \*myDate = [NSDate new]; // new is een class method

Dit creeert én initialiseert het object myDate, dat het adres van het gecreeerde object bevat. Het is gebruikelijk in Objective C om deze 2 stappen afzonderlijk te coderen:

NSDate \*myDate = [NSDate alloc]; // allloc is een class method

…alloceert het geheugen voor het NSDate object en geeft het adres terug aan myDate.

myDate = [myDate init]; // init is een instance method

…initialiseert myDate vervolgens, de wijze waarop objecten worden geinitialiseerd hangt af van de klasse definitie (waar de init methode is gedefinieerd). Meestal worden deze 2 stappen in 1 regel code geschreven:

NSDate \*mydate = [[NSDate alloc] init];

De reden om allocatie en initialisatie in de code te scheiden is dat er meestal diverse initialisatie methoden ter beschikking bestaan. De toepassing bepaalt welke init methode gebruikt moet worden.

De aanroep van new of alloc maakt de aanroeper de eigenaar van het object 🡺 deze moet dan ook de **release** doen. Het voor het object gereserveerde geheugen wordt dan vrijgegeven. De pointer variabele (myDate hier) verwijst dan echter nog steeds naar dit adres en is daarmee een ‘dangling pointer’, wat bij onrechtmatig gebruik problemen geeft. Door

myDate = nil;

op te nemen wordt het adres gewist. Wordt vervolgens een methode op myDate losgelaten, dan wordt dat in dit geval (myDate = nil) door Objective C genegeerd. Anders ontstaat een exception error.

**6.3 Using autorelease pools**

De aanroeper van new, alloc, retain en copy moet ook een release doen op het betreffende object. Objective C biedt een (soort van) alternatief voor manuele release: de release pool. De aanroep:

[myObject autorelease];

Voegt myObject toe aan de release pool van Objective C. Dit betekent dat het object ‘later’ automatisch wordt released, wanneer de code:

[pool drain];

Wordt aangeroepen, op alle objecten in de release pool wordt dan automatisch een release uitgevoerd. Een manuele release heeft daarentegen direct effect: de retain count wordt direct met 1 verlaagd.

In iPad en iPhone programma’s wordt de [pool drain] aan het eind van elke event loop uitgevoerd. Dus bijv. na het indrukken van een button wordt de bijbehorende handeling uitgevoerd en direct daarna wordt de pool drain uitgevoerd, waarna het programma weer gaat wachten op een input event. Zo’n event loop kan milliseconden duren. De reden om de pool drain in een event loop te stoppen is dat iPhone en iPad programma’s heel lang kunnen draaien, waardoor veel geheugen gebruikt kan worden, dat tussentijds moet worden vrijgegeven en niet vlak voor beeindiging van het programma.

Bij voorkeur zo weinig mogelijk gebruik maken van autorelease, dit kan tot slordigheden leiden bij het memory management. Autorelease moet alleen toegepast worden in een functie die een object creeert en dit object teruggeeft aan de aanroeper, binnen de functie vlak voor het return statement

[someObject autorelease];

opnemen. Een release van het object nog binnen de functie die het creeert is niet mogelijk omdat de retain count dan op 0 zou komen en het voor het object gereserveerde geheugen zou worden vrijgegeven.

Overigens moet in dit geval de aanroeper van deze functie een retain op het object uitvoeren!

**6.4 Apple autoreleasing objects**

Release alleen doen bij de ‘NARC’ aanroepen: new, alloc, retain en copy, welke je eigenaar maken van het object. Objective C biedt vele initialisatie methoden die een object teruggeven, deze standaard methoden doen zelf een autorelease, zodat in dat geval in de eigen code geen release nodig is en zelfs vermeden moet worden, omdat dat het memory management in de war kan schoppen.

**Hoofdstuk 7**

**7.1 Je eigen classes creeren**

In Objective C worden classes gedefinieerd in 2 secties:

* interface
* implementation

De interface beschrijft de buitenkant: de attributen en methoden die ter beschikking staan (het wat). De implementatie is de code van dat alles (het hoe).

De interface sectie ziet er als volgt uit:

**/\* Interface section**

@interface <class name> : <super class>{

@private

<instance variables>

}

/\* methoden… bijv:

- (void) myMethod; /\* NBmyMethodis een instance method (**-**) itt een class method (**+**)

@end

**/\* Implementation section**

@implementation <class name>

- (void) myMethod{

NSLog(@”In methode myMethod”);

}

@end

Opmerkingen:

* Interface sectie start met **@interface** en eindigt met **@end**
* Class names beginnen standaard met een hoofdletter;
* Super class is de parent class waarvan attributen en methoden geerfd worden. Deze moet altijd aangegeven worden. Is er geen eigen parent class, dan is de default parent NSObject, de opperklasse van Objective C waar alle klassen van erven;
* Instance variabelen geven de attributen van de objecten van deze klasse;
* @private: instance variabelen van een klasse zijn private, dwz niet te benaderen van buiten de klasse;
* Implementation sectie start met **@implementation** en eindigt met **@end**

In bijv de main function kan nu een object van <class name> (bijv Employee) gecreeerd worden:

Employee \*empl = [[Employee alloc] init];

De methoden alloc en init zijn niet in <class name> gedefinieerd, deze worden geerfd vanuit de klasse NSObject.

Het empl object kan nu gebruikt worden en moet na gebruik released worden, bijv:

[empl myMethod]; /\* de enige methode die beschikbaar is

[empl release];

Naarmate het programma groeit is het overzichtelijker om interfaces en implementaties van klassen te scheiden. Class interfaces worden in .h files opgeslagen, class implementations in .m files.

In XCode wordt een class interface en implementation file gecreeerd mbv optie New File in het File menu. In het cursusvoorbeeld wordt een desktop applicatie ontwikkeld, selecteer in het scherm dat verschijnt Cocoa (onder Mac OS X) en vervolgens Objective C Class. Daarna wordt gevraagd wat de super class is (subclass of), selecteer hier NSObject. Dan verschijnt een schermpje waarin de naam van de te creeren klasse moet worden ingevoerd (bij Save as:). In de cursus wordt hier Player ingevuld. De checkbox Add to targets moet aangevinkt zijn, omdat de klasse onderdeel moet worden van het uiteindelijke gecompileerde project. Klik vervolgens op Save.

Er zijn nu 2 files gecreeerd: Player.h en Player.m. Player.h is de interface file, die door XCode al met het interface template is gevuld. Player.m is de implementation file, die door XCode met een implementation template is gevuld.

**7.2 Methoden definieren**

Let bij het definieren van methoden op dat instance variabelen voor de gekrulde haak worden gedefinieerd en methoden daarna maar voor de @end:

@interface Player : NSObject {

@private

// definieer instance variabelen hier: voor de gekrulde haak definieren

int exampleVar;

}

// definieer methode declaraties hier: buiten de gekrulde haak, maar voor de @end

- (void) exampleMethod;

@end

Een methode declaratie bestaat uit de volgende elementen:

* Een – teken voor een instance method, en een + voor een class method. Een instance method moet in combinatie met een object aangeroepen worden, een class method met een klasse (zoals NSString),
* Dan volgt het return type tussen haakjes,
* En tot slot de naam van de methode gevolgd door “;”, indien de methode geen parameters heeft.

Heeft de methode 1 parameter, dan volgt een “:” achter de methode naam, gevolgd door de parameterlijst. Iedere parameter definitie bestaat uit het type tussen aakjes en de parameter naam:

- (void) exampleMethod: (int) exampleParam;

Kent de methode meerdere parameters dan wordt de methode naam uitgebreid met evenveel delen als parameters. Voorbeeld:

- (int) addNumber: (int) a toNumber: (int) b;

De methode naam wordt nu addNumber toNumber.

Dit wordt toegelicht aan de hand van een voorbeeld in XCode. Hiervoor is mbv. New File command (zie 7.1) een nieuwe klasse MyClass gecreeerd. XCode maakt hiervoor de .h headerfile en de .m implementatiefile aan met het default template (NB in mijn geval is het .m template vrijwel leeg). Vaak wordt zijn bij het coderen van een klasse zowel in de .h als de .m file nodig. Hiervoor heeft XCode de Assistant Editor voorhanden, klik hiervoor op het 2e icon van links rechtsboven in het XCode scherm (het gezicht). Zowel de .h als de .m file worden vervolgens getoond. De Assistant Editor probeert altijd de tegenpool van de file die getoond wordt te tonen. Van een .h file is de tegenpool de .m file, en omgekeerd.

In de MyClass .h interface file wordtMyClass als volgt gedefinieerd:

@interface MyClass : NSObject {

@private

}

-(void) performAcion;

-(int) addNumber:(int) a toNumber:(int) b;

@end

In de MyClass.m implementatiefile worden deze methoden uitgewerkt. Voor het uitschrijven van de methoden heeft XCode nog een handigheidje. Wanneer onder de @implementation regel de eerste letter van de klasse wordt ingevuld, bijv. De “p”, dan laat XCode een lijst van methoden zien die met een p beginnen. Bovenaan de lijst staat performAction, kies deze en XCode zal de volledige definitie van deze methode uit de interface file halen en kopieren naar de implementation file. Die komt er voor MyClass.m als volgt uit te zien:

@implementation MyClass

-(void) performAction (void) {

NSLog(@”Hier is ie dan”);

}

-(int) addNumber: (int) a toNumber: (int) b {

return a+b;

}

**7.3 Eigenschappen definieren**

In het voorbeeld wordt een klasse MyClass gedefinieerd met 2 instance variabelen score en privateNumber:

@interface MyClass: NSObject {

@private

int score;

int privateNumber;

}

-(void) logMessage;

@end

Er is geen methode beschikbaar om de private variabelen score en privateNumber te veranderen. Dit zou kunnen door @private in @public te veranderen, echter wordt dit afgeraden. De aanbevolen manier is om zg. accessor methoden te schrijven voor private variabelen. Die worden ook wel ‘getters’ en ‘setters’ genoemd. In Objective C is het gebruikelijk om de ‘getter’ methode dezelfde naam te geven als de instance variabele (andere talen zetten er bij voorkeur ‘get’ voor) en de ‘setter’ methode wordt set + de naam van de instance variabele:

-(int) score;

-(void) setScore: (int) s;

Deze worden juist voor of juist na de definitie van logMessage in de .h interface file toegevoegd. De implementatie van deze methoden in de .m implementatie file is als volgt:

@implementation MyClass // methoden na @implementation opnemen

-(int) score {

return score;

}

-(void) setScore: (int) s {

score = s;

}

In main.m kan nu de volgende code worden opgenomen voor het aanpassen en opvragen van de instance variabele score:

MyClass \*demo = [[MyClass alloc] init];

[demo setScore:1000]; // pas score aan en zet op 1000

NSLog(@”De waarde van score is %@”, [demo score]);

Dit werkt op zich prima maar is een nogal omslachtige manier om private instance variabelen te kunnen benaderen, zeker wanneer een klasse er diverse heeft. Daartoe is in Objective C (vanaf versie 2.0) een handigheidje ingebouwd, die werkt op basis van de compiler directives @property en @synthesize. Vervang in de interface file de definities van de methoden score en setScore door de regel:

@property int score;

Vervang de implementatie van de beide methoden score en setScore in de implementation file door de regel:

@synthesize score;

Door deze aanpassingen te doen worden de ‘getter’ en ‘setter’ automatisch aangemaakt, met de standaard Objective C namen zoals eerder beschreven. In het voorbeeld dus score en setScore. Dat betekent dat de aanroepen in main.m niet behoeven te worden aangepast.

Voor synthesized properties is in Objective C (vanaf versie 2.0) de zg “dot syntax” toegevoegd om op een makkelijker manier de betreffende instance variabele te benaderen:

demo.score = 1000; // ipv [demo setScore:1000];

NSLog(@”De waarde van score = %@”, demo.score); // ipv [demo score]

Het gebruik van de dot syntax is een kwestie van persoonlijke smaak.

**7.4 Het definieren van initializers**

Elke klasse heeft een init methode om de nodige initialisaties te verrichten, zodat een zojuist gecreeerd object een gedefinieerde status heeft. Init en alloc zijn beide methoden van NSObject. Deze worden automatisch geerfd wanneer NSObject als super class wordt opgegeven bij de definitie van een eigen klasse.

Wanneer mbv New File een klasse wordt gecreeerd in XCode, wordt automatisch een init methode gecreeerd door XCode (bij mij werkt dit niet, niet bekend waarom), die verder kan worden aangevuld:

@implementation MyClass

-(id)init {

self = [super init];

if (self) {

// eigen initialisatiecode hier, bijv. initialisatie van instance variabelen

score = 5000; // initiele score op 5000 zetten bij creeren van object

}

return self;

}

Opmerkingen bij deze code:

* *self*: dit verwijst naar het object dat is gecreeerd (in java bijv. bekend als ‘this’),
* *[super init]*: hier wordt de init methode van de super class uitgevoerd. In ons voorbeeld is dit de init methode van NSObject,
* *if (self)*: als de init methode van de super class een object teruggeeft dan slaagt de test op de waarde van self, het gecreeerde object bestaat dus. Vervolgens wordt de conditionele code doorlopen (hier: score = 5000;),
* *return self*: de eigen init methode geeft het geinitialiseerde object terug aan de aanroeper.
* *id*: het return type is ‘id’, dit is het meest generieke type voor objecten dat er bestaat, dus perfect geschikt om als type van een init methode terug te geven. Return type Player had natuurlijk ook gekund.

Zolang je een eenvoudige initialisatie taak wilt uitvoeren, volstaat een eenvoudige init methode zoals hierboven. Meer gangbaar is om specifieke initialisatie methoden te definieren. De meeste klassen hebben er een heleboel, kijk bijv. eens naar de initialisatie methoden van NSDate. Zoek hiertoe in XCode Help naar de NSDate class reference. Scroll wat naar beneden en ontdek dat er diverse init methoden zijn voor NSDate, zoals initWithString, initWithTimeIntervalSinceNow etc. In andere talen worden deze initialisatie methoden aangeduid als ‘constructors’.

Bij wijze van voorbeeld wordt een init methode initWithScore gecreeerd:

-(id) initWithScore:(int) s { // initialiseer instance variabele score met s

self = [super init];

if (self) {

score = s;

}

return self;

}

Zoals te zien is, is deze initialisatie variant vrijwel gelijk aan de eerder ontwikkelde eigen init methode.

Methode initWithScore moet nog in de interface file opgenomen worden:

@interface Player: NSObject {

@private

int score;

}

-(id) initWithScore:(int) s;

@end

De ‘eigen’ init methode hoeft niet apart opgenomen te worden, aangezien die al is gedefinieerd als onderdeel van NSObject.

NB. Zou methode initWithScore niet in de interface file zijn opgenomen, dan zou de methode een private onderdeel zijn van de klasse Player en niet van buiten af benaderbaar zijn.

**7.5 Het gebruik van dealloc**

Wanneer met XCode een nieuwe klasse wordt gecreeerd, worden automatisch 2 methoden meegeleverd: init (zie in 7.4) en dealloc. Dealloc wordt aangeroepen wanneer een object wordt verwijderd, m.a.w. wanneer de retain count op 0 komt te staan.

De reden dat XCode een eigen dealloc methode toevoegt aan de nieuwe klasse, is dat er tijdens de instantiatie (dwz creatie) van een object van deze klasse, er ‘sub objecten’ kunnen worden gecreeerd (dmv alloc) die later weer moeten worden verwijderd wanneer het object zelf wordt verwijderd.

Een voorbeeld. Voeg een NSDate variabele toe aan klasse Player:

@interface Player: NSObject {

@private

NSDate \*startDate;

}

@end

Tijdens de creatie van een Player object wordt een startDate sub-object gecreeerd:

- (id) init {

self = [super] init;

if (self) {

startDate = [[NSDate alloc] init]; // creeer en initialiseer sub-object

}

return self;

}

Omdat de Player klasse zelf een alloc laat uitvoeren tijdens initialisatie, is het zelf verantwoordelijk voor het opruimen van het gealloceerde geheugen, dmv een release op het gecreeerde object.

Wanneer het Player object released wordt en de retain count komt op 0 te staan, dan volstaat het niet om enkel het Player object zelf te dealloceren, dat (dealloceren) moet ook gebeuren voor de sub-objecten. Pas dealloc daarom als volgt aan:

-(void) dealloc {

[startDate release];

[super dealloc];

}

De aanpak in geval van dealloc is om eerst de eigen spullen op te ruimen (hier: startDate object) en het daarna over te laten aan de super class.

**Hoofdstuk 8 Collections**

**8.1 Werken met C-style arrays**

Standaard C-style array declaratie:

Int multipleValues[5];

…definieert een array met 5 elementen. Let op dat de indexering met 0 begint.

De initialisatie van arrays dmv assignment statements wordt onhandig als het aantal elementen groot is. Hiervoor is een shortcut beschikbaar:

int multipleValues[5] = { 50, 60, 70, 80, 90 };

Wanneer de array op deze manier wordt geinitialiseerd hoeft het aantal elementen in de declaratie niet te worden gegeven, dit bepaalt de compiler zelf aan de hand van het aantal opgegeven waarden:

int multipleValues[] = { 50, 60, 70, 80, 90 };

Geef je daarentegen teveel initialisatie waarden op, bijv 6 ipv 5 in dit voorbeeld, dan volgt een waarschuwing in XCode.

Met een C-style array kan ook een array van objecten gedefinieerd worden:

NSString \*myStringArray[5];

…creeert een array van 5 (pointers naar) strings.

De strings moeten wel gecreeerd en geinitialiseerd worden dmv alloc en init:

myStringArray[0] = [[NSString alloc] initWithString:@”Hello”];

Uiteraard moet tegenover een alloc ook een release staan:

[myStringArray[0] release];

Net zoals met integers kunnen de elementen van de array myStringarray op een eenvoudiger manier geinitialiseerd worden:

NSString \*myStringArray[] = { @”eerste”, @”tweede”, @”derde”, @”vierde” };

Op deze wijze worden de elementen zowel gecreeerd als geinitialiseerd. Omdat de creatie niet dmv een alloc gebeurt, zijn de elementen alle autorelease elementen, die automatisch worden verwijderd adhv het [pool drain] statement.

Mbt C-style arrays moeten de volgende 3 punten in de gaten gehouden worden:

* Er is geen ‘bounds checking’: de compiler en builder checken niet of een array index binnen de aangegeven grenzen zit. Er zal geen foutmelding of waarschuwing verschijnen als in het gebruikte voorbeeld zou worden verwezen naar myStringArray[121]. Dit leidt echter tot onvoorspelbaar gedrag.
* Arrays hebben een vaste grootte die bij de declaratie is bepaald. Naderhand vergroten of verkleinen is niet mogelijk.
* Het is niet mogelijk om verschillende typen elementen in een array te gebruiken, ze zijn allemaal van hetzelfde type.

**8.2 Werken met Objective C array objecten**

Een nadeel van C-style arrays is dat deze maar 1 type elementen kunnen bevatten. Hier is met de komst van Objective C wat aan gedaan. Het Foundation framework bevat een klasse NSArray, dat verschillende typen elementen kan bevatten. Het werken met een NSArray object is echter geheel anders dan met een C-style array.

Het begint al met de declaratie (en initialisatie):

NSArray \*myArray = [NSArray arrayWithObjects:@”one”, @”two”, nil];

Opmerkingen:

* Omdat we met een NSArray object werken, is myArray een pointer naar een NSArray object.
* In principe kan een NSArray object gecreeerd en geinitialiseerd worden als volgt:

NSArray \*myArray = [[NSArray alloc] **init**WithObjects:@”one”, @”two”, nil];

… echter is dan een manuele release nodig. Gebruikelijker is het om de methode **array**WithObjects te gebruiken, die zowel alloceert als initialiseert.

* Omdat de methode arrayWithObjects zelf de array elementen creeert, worden deze in de autorelease pool gezet en worden die later automatisch verwijderd dmv het [pool drain] statement.
* De grootte van de array wordt niet expliciet aangegeven omdat dit duidelijk wordt uit de initialisers.
* De waarde ‘nil’ geeft aan dat zojuist de laatste initialiser is gegeven en het aantal elementen kan worden bepaald.

Het benaderen van een array element gaat niet op de standaard C manier van de index tussen blokhaken. Hiervoor is een methode objectAtIndex ontwikkeld die als argument de index mee krijgt:

[myArray objectAtIndex:1]; //verwijzing naar het 2e element in myArray

Zoals gezegd kan een NSArray elementen van verschillende typen bevatten, onderstaand voorbeeld voegt nog een NSDate object toe aan myArray:

NSDate myDate = [[NSDate alloc] init];

NSArray \*myArray = [[NSArray arrayWithObjects:@”one”, @”two”, myDate, nil];

Merk op dat myDate via een alloc gecreeerd wordt en dus ook zelf moet worden vrijgegeven mbv een release. Dit kan direct na de declaratie van myArray gebeuren, de methode arrayWithObjects verhoogt nl. de retain count van myDate, zodat een release van myDate geen kwaad kan.

De 3 hier genoemde NSArray methoden initWithObjects, arrayWithObjects en ObjectAtIndex zijn de meest gebruikte methoden uit het grote arsenaal van NSArray methoden. Ook veelgebruikt is de count methode:

[myArray count]; // NB geeft een unsigned long terug, geen integer

Een belangrijke eigenschap van NSArray objecten is dat deze onveranderbaar (‘immutable’) zijn. Het is niet mogelijk om later elementen toe te voegen of te verwijderen. Hiervoor is een andere array klasse ontworpen: NSMutableArray. Dit is een subklasse van NSArray, zodat het de beschikking heeft over alle methoden die tot NSArray behoren en daarnaast extra methoden voor o.a. toevoegen en verwijderen van elementen: addObject en removeObjectAtIndex. Voorbeelden:

[myArray addObject:@”three”]; // voegt een nieuw laatste element toe

[myArray removeObjectAtIndex:1]; // verwijdert object op index 1 en schuift overige op

**8.3 Het gebruik van dictionaries**

NSArray wordt zeer veel gebruikt in Objective C programma’s. Maar er zijn situaties dat het niet handig is om met een index gebaseerde lijst te werken, omdat de index niet bekend is of niet relevant is voor de situatie. Voor dat doel is de klasse NSDictionary ontworpen. Deze werkt als een verzameling van beschrijvingen waarbij een term (woord) als index wordt gebruikt. Een voorbeeld hiervan is een US statentabel, waarbij de afkorting van de staat (bijv ‘AZ’) gebruikt wordt om de staatnaam (‘Arizona’) op te zoeken. Voorbeeldcode:

NSDictionary \*states = [NSDictionary dictionaryWithObjectsAndKeys:

@”Arizona”, @“AZ”, @”California”, @“CA”, @”Hawaii”, @”HI”,

@”Colorado”, @”CO”, @”New Mexico”, @”NM”, nil];

NSString \*someState = @”CA”;

NSLog(@”%@ is voor %@”, someState, [states objectForKey:someState]);

De methode dictionaryWithObjectsAndKeys accepteert dictionary entries in de volgorde dictionary description – dictionary zoekterm (bijv “Arizona”, “AZ”), totdat nil wordt tegengekomen, zoals bij NSArray objecten. De methode objectForKey geeft adhv de zoekterm de dictionary description terug. Het doet er hierbij niet toe in welke volgorde de elementen aan de dictionary worden toegevoegd. De positie van een element is niet relevant.

Net als het geval is bij NSArray objecten, zijn NSDictionary objecten onveranderbaar (‘immutable’). Ook in dit geval is er een NSMutableDictionary klasse, dat een subklasse is van NSDictionary en uitbreidingen kent voor o.a. het toevoegen en verwijderen van dictionary elementen. Het toevoegen van een dictionary element aan states gaat als volgt:

[states setObject:@”Florida” forKey:@”FL”];

**8.4 Snelle enumeratie**

NSArray en NSDictionary zijn voorbeelden van collections. Nog niet besproken is hoe alle elementen van een collection kunnen worden doorlopen. Dit gaat mbv een alternatieve for loop:

for (NSString \*key in states) {

NSLog(@”%@ is voor %@”, key, [states objectForkey:key]);

}

Opmerkingen:

* key is een tijdelijk NSString object, dat gebruikt wordt als zoekterm in states
* ‘in states’ geeft aan dat het NSDictionary object states wordt doorlopen

**Hoofdstuk 9 File management**

**9.1 Introductie van file management in Objective C**

Objective C biedt diverse klasses en methoden voor file handling. Het is daarom niet nodig gebruik te maken van de low level faciliteiten van C op dit gebied.

Het hart van de file handling klassen is NSFileManager. Deze biedt methoden voor het kopieren van files (en paden) en URL’s, het verplaatsen van files, het testen op bestaan van files, het opvragen van eigenschappen, het verwijderen van files, etc. Codevoorbeelden:

NSFileManager \*fileMgr = [[NSFileManager alloc] init];

NSString \*path = @”C:\Temp\test.txt”; // hardcoded naam ter illustratie

if (fileMgr fileExistsAtPath:path) { // check of file <path> bestaat

NSLog(@”File %@ bestaat”, path);

// vraag attributen op, negeer eventueel teruggegeven error object

// NB attributesOfItemAtPath geeft een NSDictionary object terug

NSDictionary \*fileAttr = [fileMgr attributesOfItemAtPath: path error:nil];

// Toon de file attributen

for (NSString \*key in fileAttr) {

NSLog(@”Attribuut %@ is %@”, key, [fileAttr objectForKey:key]);

}

// Wijzig de naam van de file in iets willekeurigs anders

[fileMgr moveItemAtPath:path toPath:@”C:\Temp\testA.txt” error:nil];

}

else {

NSLog(@”File %@ bestaat niet”, path);

}

// Release alle zelf gecreeerde objecten

[fileMgr release];

**9.2 Werken met paden en URL’s**

In Objective C is het gebruikelijk om met URL’s te werken voor adressen van web pagina’s maar ook voor filenamen. Dus geen NSString objecten meer om file namen aan te duiden, in plaats daarvan gebruiken we NSURL objecten. Dit is best practice geworden en wordt ook beter ondersteund dan het gebruik van NSString objecten om padnamen te specificeren.

De NSURL klasse biedt methoden die sneller en efficienter in gebruik zijn, biedt betere foutdetectie en wordt breder ondersteund, zeker door nieuwe Cocoa klassen die alleen met NSURL objecten werken. Voorzover het het werken met files en URL’s betreft heeft de NSURL klasse sterk de voorkeur boven de NSString klasse. De NSString klasse zelf heeft diverse methoden die zowel met NSString als NSURL objecten werkt. Een paar voorbeelden van klassen die alleen met NSURL objecten werken zijn:

* NSDocument, methode readFromURL
* AVPlayerItem, methode initWithURL
* NSURLRequest, methode initWithURL

Hoe moet een NSString object worden omgezet in een NSURL object? Een URL is opgebouwd uit een scheme, een domain en een pad:

<http://www.lynda.com/newreleases.aspx>

scheme: http (kan ook https, ftp of file zijn)

domain: [www.lynda.com](http://www.lynda.com)

pad: newreleases.aspx

Voor een file op onze eigen computer (localhost) zou dit als volgt worden:

<file://localhost/Users/Frank/test.txt>

Localhost kan weggelaten worden (dit is de default domain name): <file:///Users/Frank/test.txt>, de URL bevat 3 forward slashes, maar is wel geldig.

Voor de initialisatie van een URL biedt de NSURL klasse diverse methoden. Om een standaard padnaam te gebruiken voor de initialisatie, moet de methode fileURLWithPath gebruikt worden:

NSURL \*myURL = [NSURL fileURLWithPath: path]; //path = C:/Users/Frank/test.txt

NSURL biedt ook een initialisatie methode URLWithString, maar die verwacht een correcte URL (dus in de vorm van <file:///Users/Frank/test.txt>) en zou dus een fout geven op een standaard padnaam.

De NSFileManager klasse is een wat oudere klasse die voornamelijk met padnamen werkt, maar ook hier zijn URL gebaseerde methoden toegevoegd. Een voorbeeld hiervan is de methode moveItemAtURL, de URL tegenpool van moveItemAtPath (zie in de voorbeeldcode in 9.1). De URL gebaseerde methoden hebben zoals gezegd de voorkeur. Met XCode kan in de klasse bibliotheek naar de juiste methode gezocht worden (CMD + klik op de klassenaam).

**9.3 Lezen en schrijven van strings**

Met de NSURL klasse kunnen eenvoudige file handelingen gedaan worden, zoals het inlezen van de inhoud van een file in een string en het schrijven van een string naar een file. Hiervoor is de NSFileManager klasse niet nodig.

Voorbeeldcode voor het inlezen van een file:

NSURL \*fileURL = [NSURL fileURLWithPath:@”/Users/Frank/text.txt”];

NSString \*tekst = [NSString stringWithContentsOfURL:fileURL

encoding: NSUTF8StringEncoding error:nil];

NSLog(@”De file inhoud is %@”, tekst);

De parameter encoding geeft aan dat de meest standaard wijze van encoding UTF8 van toepassing is op de in te lezen file. Fouten worden genegeerd (error object is nil).

We gaan nu de inhoud van de file veranderen. Hiervoor passen we de ingelezen string <tekst> aan. Deze is van het type NSString, dit is echter een onveranderbaar type. Objecten van het type NSMutableString kunnen wel veranderd worden:

NSMutableString \*tekst = [NSMutableString stringWithContentsOfURL:fileURL

encoding: NSUTF8StringEncoding error:nil];

// Breidt de file uit met 2 keer extra tekst dmv appendString methode

[tekst appendString:@” hier is wat extra tekst”];

[tekst appendString:@” en hier is nog wat extra tekst”];

// Gebruik writeToURL methode om string <tekst> naar file te schrijven.

[tekst writeToURL:fileURL atomically:YES encoding:NSUTF8StringEncoding error:nil];

NB De parameter atomically is een BOOL variabele, indien YES dan wordt eerst een tijdelijke file met de gewenste inhoud gecreeerd, lukt dit dan wordt deze renamed naar de target file. Dit is om zeker te zijn dat de file wordt bewaard mochten er bestandsproblemen optreden met de target file.

**9.4 Archiveren van objecten**

In de vorige paragrafen is ingegaan op het werken met tekstfiles. Voor het opslaan van willekeurige objecten (zoals personeelsgegevens) zijn de beschreven methoden niet erg geschikt. Hiervoor biedt het Foundation framework betere mogelijkheden, in de vorm van andere klassen.

Klasse NSKeyedArchiver biedt de mogelijkheid om een object in stukjes te breken, aan elk stukje een key te hangen en dit in een file op te slaan. Dit wordt coderen genoemd. Om zo opgeslagen objecten terug in te lezen is de klasse NSKeyedUnarchiver ontwikkeld, deze leest opgeslagen objecten uit een gecodeerde file, decodeert ze en reconstrueert het opgeslagen object. Aan de hand van de zelf gecreeerde Employee klasse wordt dit toegepast.

File Employee.h bevat de interface van de Employee klasse:

#import <Foundation/Foundation.h>

@interface Employee: NSObject {

@private

NSString \*name;

int grade;

}

@property (retain) NSString \*name;

@property int grade;

@end

File Employee.m bevat de implementatie van klasse Employee:

#import “Employee.h”

@implementation Employee

@synthesize name, grade; // getters en setters

// De description methode creeert een string met informatie over een Employee object

- (NSString \*) description {

NSString \*desc =

[NSString stringWithFormat:@”Employee: %@, Grade: %i”, name, grade];

return desc;

}

- (id) init { }

- (void) dealloc { }

@end

File main.m bevat ons testprogramma (standaard zaken weggelaten):

Employee \*frank = [[Employee alloc] init]; // creeer Employee object frank

[frank setName: @”Frank de Groot”];

[frank setGrade: 21];

NSLog(@”Details zijn %@”, [bob description]); // toon info van object frank

// save object frank ==> zie in de tekst hieronder, hier gaat deze paragraaf nl. om

Tot hier niets nieuws onder de zon. Om een Employee object op te slaan zijn een aantal zaken nodig. Ten eerste moet er een encoder methode gebouwd worden die een Employee object kan coderen. Deze methode heet encodeWithCoder, dit moet exact deze naam zijn omdat NSKeyedArchiver deze methode gebruikt. Een kijkje in de documentatie leert dat het NSCoding protocol voorschrijft dat er 2 methoden moeten zijn geimplementeerd binnen een klasse om uberhaupt objecten van deze klasse te kunnen coderen en decoderen: initWithCoder en encodeWithCoder.

Methode encodeWithCoder moet een Employee object kunnen coderen, maw. de name en grade velden kunnen coderen. EncodeWithCoder heeft als parameter een NSCoder object dat de codering representeert en wordt als volgt geimplementeerd:

-(void) encodeWithcoder: (NSCoder \*) coder {

// veld <name> wordt als key “EMPname” gecodeerd, de naam is vrij te kiezen!

[coder encodeObject:name forkey:@”EMPname”];

// veld <grade> wordt als key “EMPgrade” gecodeerd. Omdat grade een integer is en // geen object, moet het in een object ingekapseld worden. Dit gebeurt dmv //NSNumber methode numberWithInt. We gebruiken een temp var gradeBox

NSNumber gradeBox = [NSNumber numberWithInt:grade];

[coder encodeObject:gradeBox forkey:@”EMPgrade”];

}

Om het coderen in gang te zetten en het resultaat in een file op te slaan moet in main.m onder het kopje “//save object frank” nog de methode NSKeyedArchiver worden aangeroepen:

//save object frank

[NSKeyedArchiver archiveRootObject:frank toFile:@”Users/frank/frank.plist”];

Er moet een root object gearchiveerd worden, object frank bevat nl. o.a. een NSString object. Het resultaat van de archivering moet in een zg. property list (plist) file opgeslagen worden. Wordt het programma nu gebuild en gerund, dan wordt de property file frank.plist gecreeerd. Deze kan met XCode worden geopend. Een property file geeft naast de data meer gedetailleerde informatie over de opgeslagen gegevens, bijv. tot welke klasse ze behoren etc, van welke klasse geerfd wordt, etc.

Tot zover het archiveren. Het complement van hiervan moet nog worden uitgevoerd, nl. de-archiveren (restore). Hiervoor moet de eerder genoemde methode initWithCoder worden geimplementeerd. Deze doet in feite het tegenovergestelde van methode encodeWithObject:

-(id) initWithCoder(NSCoder \*coder) {

name = [coder decodeObjectForKey:@”EMPname”];

// grade is als een object opgeslagen, dit object wordt eerst gedecodeerd.

NSNumber \*gradeBox = [coder decodeObjectforKey:@”EMPgrade”];

// nu nog een integer maken, dit kan mbv NSNumber methode integerValue, die //geeft echter een NSInteger terug die gecast wordt naar een integer

grade = (int) [gradeBox integerValue];

// het nu geinitialiseerde Employee object wordt aan de caller teruggegeven

return self;

}

In main.m moet nu nog code worden toegevoegd om het gearchiveerde Employee object in te lezen. Hiervoor creeren we een nieuw Employee object eric, dat we initialiseren mbv NSKeyedUnarchiver en daarna mbv NSLog weergeven:

Employee \*eric = [NSKeyedUnarchiver unarchiveObjectWithFile:@”Users/Frank/frank.plist”];

NSLog(@”Het gerestorede object is %@”, [eric description]);

NB 1:

De NSLog zoekt in geval van “%@” altijd naar de description methode van het weer te geven object. Dit betekent dat in dat geval niet expliciet de description methode in de NSLog aanroep aangegeven hoeft te worden:

NSLog(@”Het gerestorede object is %@”, eric);

…volstaat in dit geval ook.

NB 2:

In de code snippet library in XCode is een code snippet voor initWithcoder opgenomen. Deze kan standaard gebruikt worden. Voor enodeWithCoder is er geen code snippet voorhanden merkwaardig genoeg.

NB 3:

De initWithCoder methode geeft een autoreleased object terug. In geval van het name veld, dat een object is, moet daarom gezorgd worden dat het object bewaard blijft indien elders wordt verwijderd. Er moet een [name retain] worden toegevoegd na de decodeObjectForKey aanroep, kan ook geschreven worden als:

name = [[coder decodeObjectForKey:@”EMPname”] retain];

In de dealloc methode van de Employee klasse moet dan als tegenhanger een release op name worden uitgevoerd:

[name release];

**Hoofdstuk 10 Meer complexe klassen**

**10.1 Erven en NSObject**

In dit deel wordt uitleg gegeven over het erven van methoden van super classes (ook wel parent classes genoemd). NSObject is de root class van het Foundation framework. Alle methoden die toegankelijk zijn voor de parent class, zijn dat ook voor de (child) klasse zelf. De (child) klasse kan eigen methoden toevoegen, het is echter niet mogelijk om methoden van de parent class te verwijderen.

Het is mogelijk om methoden van de parent class te ‘overiden’, door ze een eigen implementatie te geven. Als voorbeeld wordt de description methode gebruikt, die wordt aangeroepen als een object mbv de placeholder ‘@’ in NSLog wordt afgedrukt.

Verder niets bijzonders in deze les.

**10.2 Klassen met categories uitbreiden**

Soms is het gewenst een bestaande klasse uit te breiden met eigen methoden, bijv. een String methode die precies datgene met een string doet wat gewenst is, bijv. alle spaties van een string vervangen door een underscore. Dit is in de meeste talen alleen mogelijk door een subklasse te creeren van de standaard String klasse, en binnen die subklasse de nieuwe methode te ontwikkelen.

In Objective C kan de bestaande NSString klasse met eigen methoden uitgebreid worden, dit wordt in Objective C een ‘category’ genoemd. De syntax lijkt op een klasse definitie, maar is net even anders:

@interface NSString (<category>)

// het is niet mogelijk om nieuwe instance variabelen te definieren

// voeg methoden toe

@end

Let op dat { en } haakjes ontbreken, er kunnen door een category nl. geen instance variabelen worden toegevoegd aan een bestaande klasse.

Voeg een category to in XCode dmv File en vervolgens New. Selecteer vervolgens Cocoa en kies “Objective C category” uit het scherm dat verschijnt. XCode heeft nl. een code template voor een nieuwe category. Klik nu op Next. XCode vraagt voor welke klasse een category wordt aangemaakt. Vul hier NSString in (niet de default NSObject) en klik op Next. Nu wordt gevraagd de naam van de category in te voeren. Dit is een zelfgekozen naam. De standaard naamgeving volgens Objective C is om de naam te laten beginnen met de klassenaam die wordt uitgebreid gevolgd door het ‘+’ teken en een zelfgekozen beschrijvende naam. We kiezen hier voor ‘NSString+ConvertWhitespace’, dit maakt het later opzoeken van de gecreeerde file later eenvoudiger.

Er zijn nu 2 files NSString+ConvertWhitespace.h en NSString+ConvertWhitespace.m aangemaakt. In de .h file wordt de interface van de nieuwe methode gedefinieerd (XCode creeert de standaard code):

#import <Foundation/Foundation.h>

@interface NSString (NSString\_ConvertWhitespace)

// eigen code volgt nu, de methode naam is vrij te kiezen:

-(NSString \*) convertWhitespace;

@end

In de implementation .m file wordt de nieuwe methode gedefinieerd. Ook hier heeft XCode al de nodige standaard code gecreeerd:

#import “NSString+ConvertWhitespace.h”

@implementation NSString (NSString\_ConvertWhitespace)

-(NSString \*) convertWhitespace {

return [self stringByReplacingOccurrencesOfString:@” “ withString:@”\_”];

}

@end

Hier wordt een beschikbare NSString methode gebruikt voor het gewenste resultaat, waardoor het in feite niet nodig is om er een category voor te bouwen, maar het gaat hier om het idee van hoe een category wordt geimplementeerd.

Let op dat, om de category te kunnen gebruiken, in de betreffende code file een import van de category header file wordt gedaan. In het voorbeeld wordt in main.m een import “NSString+ConvertWhitespace.h” toegevoegd.

**10.3 Protocollen definieren**

Een protol is een lijst van methoden die een object moet uitvoeren. Wie (object, klasse) ze moet uitvoeren en hoe ze werken is niet gespecificeerd. Een protocol dient als standaard wijze van interactie tussen objecten, Een wat concreter voorbeeld is het protocol voor een schoonmaker. Die moet vloeren kunnen dweilen, kunnen stofzuigen, de vuilnisbakken legen en optioneel de ramen kunnen zemen. In Objective C zou dat er als volgt uitzien:

@protocol Cleaner

@required // dit keyword is optioneel

- (void) cleanFloors;

- (void) vacuum;

- (void) emptyTrash;

@optional

- (void) cleanWindows;

@end

Zoals gezegd is een protocol klasse onafhankelijk, elke klasse is vrij om het protocol te ondersteunen, maw. zich te conformeren aan het protocol. Dit houdt in dat in ieder geval de required methoden geimplementeerd moeten worden door de klasse.

Als Objective C programmeur zul je niet snel zelf protocollen definieren, maar je conformeren aan bestaande protocollen. Dit hebben we in deze cursus al eerder meegemaakt in par. 9.4, waar de klasse Employee zich moest conformeren aan het NSCoding protocol om gebruik te kunnen maken van de methoden van de voorgedefinieerde klassen NSKeyedArchiver en NSKeyedUnarchiver. Hiervoor moesten de mehoden initWithCoder en encodeWithCoder worden geimplementeerd. Het NSCoding protocol schrijft enkel deze 2 methoden voor, meer niet. Het Foundation framework zit er vol mee, en voor het bouwen van iOS en desktop applicaties krijg je er zeker mee te maken.

Een klasse geeft als volgt aan zich te conformeren aan een protocol. In de interface file wordt de naam van het protocol tussen < en > weergegeven:

@interface MyClass: NSObject <Cleaner, OtherProtocol1, OtherProtocol2>

Verder niets nieuws onder de zon. De required methoden van de protocollen moeten ook in de interface file worden opgenomen – als ware het ‘eigen methoden’ – en uiteraard worden geimplementeerd in de implementation file.

**10.4 Dynamic typing**

Het type ‘ id’ zijn we al een aantal keren tegengekomen. Het kan voor alle typen objecten gebruikt worden en wordt door veel methoden uit het Foundation framework geretourneerd (bijv. alloc en init). Het is een concreet voorbeeld van dynamic typing.

Dit in tegenstelling tot static typing. Om aan te geven hoe strikt static typing werkt worden 3 vrijwel identieke klassen gedefinieerd, genaamd Class1, Class2 en Class3. Class1 en Class2 zijn leeg, hebben geen instance variabelen en methoden. Class2 heeft 1 methode, genaamd ‘secondOnly’. Van elk van de 3 klassen in main wordt een object gecreeerd.

Methode secondOnly wordt nu voor het Class2 object aangeroepen, dit verloopt zonder problemen. Vervolgens wordt methode secondOnly voor het Class1 object aangeroepen. Xcode geeft tijdens het coderen al een warning ‘<object> may not respond to ‘secondOnly’’. Toch wordt het programma gerund waarna een run-time error optreedt ‘unrecognized selector sent to instance xyz’. De selector secondOnly is nl niet bekend bij Class1 en kan dus niet worden aangeroepen.

Een object kan ook dynamisch gedefinieerd worden. In plaats van statisch dmv

Class1 \*obj1 = [[Class1 alloc] init];

…kan dit ook dynamisch dmv

id objX; // let op: geen pointer variabele, de \* dus weglaten

objX = [[Class1 alloc] init];

Pas bij de initialisatie wordt het type van variabele objX bepaald. Wordt nu echter

[objX secondOnly];

aangeroepen, dan volgt dezelfde runtime error als eerder beschreven. Het object objX ‘weet’ echter zelf wel tot welke klasse het behoort, daarvan kan gebruik gemaakt worden door te testen of een object variabele tot een bepaalde klasse behoort en/of dat het met een bepaalde methode kan werken:

if [objX isKindOfClass:[Class2 class]] {

// code indien objX tot Class2 behoort

}

…test of objX van het type Class2 is. De methode isKindOfClass is beschikbaar voor iedere object variabele. Let ook op het keyword ‘class’ dat moet worden gebruikt.

Rechtstreeks testen of een object een bepaalde methode kan verwerken kan ook:

if [objx respondsToSelector:@selector(secondOnly)) {

// code indien objX methode secondOnly aan kan

}

Methode respondstToSelector kan voor iedere object variabele gebruikt worden. Let hier op de @selector, tussen haakjes erachter de methodenaam invullen.

**Hoofdstuk 11 Debugging**

**11.1 Veel voorkomende compilatie fouten**

Dmv code voorbeelden worden veel gemaakte codeerfouten aangegeven:

* ‘;’ aan het einde van een statement vergeten;
* ‘\*’ vergeten bij declaratie van object. Let op dat dit voor een object van type ‘id’ niet moet;
* Het aantal ‘{‘ haken correspondeert niet met het aantal ‘}’ haken, maw de compiler verwacht nog een ‘}’ of constateert juist een te vroeg einde van de subroutine;
* Interface file vergeten te importeren, bijv bij zelf gemaakte klasse (#import “<MyClass.h>”);
* Kleine ipv hoofdletter gebruikt bij klassenaam, bijv NSstring ipv NSString, NLlog ipv NSLog. Vergeet niet dat Objective C case sensitive is;
* Single quotes (‘) ipv double quotes (“) gebruikt in NSLog aanroep. NSLog drukt alleen NSStrings af, waarvoor double quotes gebruikt moeten worden.

De stelregel is om bij de eerste fout te beginnen en dan naar beneden toe te werken, aangezien de eerste fout vaak meerdere tot gevolg heeft en verder in de code doorwerkt. Gebruik de issue navigator in XCode als hulpmiddel bij het oplossen van codeerfouten. Dit is het icoon met driehoek en uitroepteken linksboven in het scherm.

**11.2 Veel voorkomende compilatie waarschuwingen**

De meest voorkomende compiler warning is die voor ‘unused variables’. Dit is feitelijk een overbodige foutmelding, want hij kan geen kwaad. Dit geldt echter niet voor de meeste warnings, daarom is het beter ze niet te negeren.

Warnings worden door de issue navigator getoond en ook in het kleine dashboard scherm midden bovenin het XCode window.

Veel voorkomende warnings die wel tot problemen leiden:

* ‘@’ teken vergeten bij NSLog call of NSString object initialisatie;
* ‘=’ ipv ‘==’ gebruikt in if statement;
* non-void functions die in sommige gevallen geen waarde retourneren, bijv. omdat er vlak voor het return statement een if statement staat, waardoor het return statement conditioneel wordt uitgevoerd en daardoor soms ook niet;
* Een object wordt met een verkeerde methode aangeroepen, van een klasse die die methode niet heeft gedefinieerd. De build slaagt maar het programma crashed in dit geval wanneer het gerund wordt. Vanwege dynamic typing geeft de compiler hier geen fout maar een warning.

XCode kan zo ingesteld worden dat warnings als fouten worden behandeld. Selecteer hiervoor het hoogste niveau van het project in de project navigator in XCode. Selecteer rechts in het scherm de build settings. Zorg dat alles getoond wordt (‘All’ selectie, naast Basic knopje). Bijna onderaan het window is een optie ‘Treat warnings as errors’. Zet dit op Yes. Deze project setting heeft tot gevolg dat warnings tijdens de build als errors beschouwd worden, waardoor de build faalt indien de compiler warnings in het programma heeft gevonden.

**11.3 Veel voorkomende runtime errors**

Een veel voorkomende fout is het gebruik van null objecten, waar methoden op los gelaten worden (oftewel: messages naartoe gestuurd worden). In tegenstelling tot bijv java en C# is dit toegestaan in Objective C. Indien tijdens runtime berichten naar null objecten worden gestuurd, dan worden deze genegeerd en loopt het programma (vaak) door alsof er niets aan de hand is. In java en C# zou het programma crashen. De oorzaak is overigens meestal dat de object variabelen weliswaar gedefinieerd zijn, maar niet geinitialiseerd mbv alloc en init.

Een ander voorbeeld is het toepassen van een methode op een object, terwijl de methode niet tot de klasse behoort van het object. Hiervoor verschijnt tijdens de build een warning en crashed het programma tijdens runnen. Concreet voorbeeld:

NSDate \*today = [[NSDate alloc] init];

[today uppercaseString]; // NB uppercaseString is geen methode van NSDate

Omdat de compiler een warning geeft, was dit gedrag te verwachten. Wordt nu echter variabele today van het type ‘id’ gedefinieerd, dan geeft de compiler geen foutmelding omdat hier sprake is van dynamic typing, waardoor de compiler niet weet van welk type variabele today is en het dus mogelijk ook goed kan zijn. Tijdens runnen volgt echter dezelfde runtime error.

Een ander voorbeeld is dat een object released wordt (met retain count 0), terwijl daarna hetzelfde object met een methode wordt aangeroepen. Het programma crashed met de runtime error ‘Program received signal: EXC\_BAD\_ACCESS’. Hoewel de object variabele nog steeds naar een geldig geheugen adres wijst – en dus geen null pointer is – is dit geheugen vrijgegeven waardoor de inhoud inmiddels overschreven kan zijn en feitelijk onzin bevat. Was de object variabele direct na de release op ‘nil’ gezet, dan was de aanroep van de methode genegeerd en was het programma niet gecrashed. Je kunt je echter afvragen of dit wel zo verstandig is…

**11.4 Exception handling dmv ‘try catch’**

Gestructureerde exception handling werkt in Objective C dmv try-catch-finally. Aan de hand van de volgende code wordt dit uitgelegd:

id today = [[NSDate alloc] init];

[today uppercaseString];

Methode uppercaseString behoort niet tot klasse NSDate maar NSString. Het runnen van deze code zal een run-time error SIG\_ABORT geven en afgebroken worden. De compiler vindt deze fout niet omdat variabele today als ‘id’ is gedeclareerd en dus een pointer naar elk mogelijk object kan zijn.

Door de code in een try-catch block te zetten kan de run-time error afgevangen worden, waardoor het programma door kan lopen. Een code snippet voor try-catch is in XCode in de code snippet library te vinden. Het ‘try’ gedeelte bevat de code waar de zorg zit. De code in het ‘try’ block wordt normaal uitgevoerd, bij de eerste de beste exceptie wordt naar het ‘catch’ block gesprongen. De code in het ‘finally’ block wordt altijd uitgevoerd, of er nu wel of geen exceptie is opgetreden in het ‘try’ block. Toegepast op bovenstaand voorbeeld ontstaat de volgende voorbeeldcode:

@try {

id today = [[NSDate alloc] init];

[today uppercaseString];

}

@catch (NSException \*excep) {

NSLog(@”Fout gevonden : %@”, excep); // druk alleen de fout af

}

@finally {

NSLog(@”Finally blok bereikt”);

}

Wordt deze code gerund, dan treedt de run-time error nog steeds op, echter crashed het programma niet en wordt de fout in het ‘catch’ block afgedrukt.

**11.5 Breakpoints en debuggen**

XCode biedt de nodige mogelijkheden voor debugging. Om een breakpoint te zetten dient in de gutter - direct links naast de code regel – geklikt te worden, er verschijnt een blauwe pijl ten teken dat het breakpoint is gezet. Hetzelfde symbool is linksboven in het XCode scherm te zien (de breakpoint navigator), wordt er op geklikt dan worden alle gezette breakpoints getoond. Wordt eenmalig op een breakpoint in de gutter geklikt, dan wordt het lichtblauw ten teken dat het nog wel bestaat maar niet meer actief is. Om het te verwijderen kan het in de breakpoint navigator worden verwijderd of uit de gutter weg worden gesleept.

Wordt een programma gerund dan stopt het op het eerste bereikte breakpoint. Linksonder in het XCode scherm bevindt zicht de debugging area. Hierin worden de waarden van variabelen die in scope zijn getoond. Door op de variabelen te klikken kan eventueel de waarde ervan worden gewijzigd. Informatie zoals het type, en de actuele waarde van variabelen kan ook opgevraagd worden door met de muis over de variabelen in de code te bewegen.

XCode biedt verschillende mogelijkheden om het programma voort te zetten. Deze zijn in de vorm van buttons te zien links onder in het code scherm, o.a. continue to next breakpoint, step over, step into (function), etc.

De debugger die wordt gebruikt is de GNU debugger. Ben je daar bekend mee dan kun je in het debugger scherm nog commando’s ingeven. Verder is het mogelijk om debugger watches en expressies te definieren, daar wordt niet verder op ingegaan.

**Hoofdstuk 12 Conclusie**

**12.1 Andere frameworks (naast Foundation)**

Het Foundation framework bestaat uit ca. 100 klassen waarmee allerlei basieke zaken gerealiseerd kunnen worden en alleen commandline tools kunnen worden gebouwd. Voor het bouwen van desktop applicaties is het Cocoa framework nodig, voor iOS applicaties is het Cocoa Touch framework nodig.

De term framework voor Cocoa is enigszins misleidend. Dit wordt duidelijk wanneer in XCode een Cocoa applicatie wordt gecreeerd en in de project navigator het Cocoa framework wordt onderzocht. Cocoa bestaat slechts uit de header file Cocoa.h, die op zijn beurt niets meer bevat dan #imports van Foundation.h, AppKit.h en CoreData.h.

AppKit bevat klassen voor het ontwikkelen van desktop applicaties, mbt windows, menu’s, buttons, fonts, afbeeldingen etc, maar ook voor event handling en achtergrond functionaliteit, spelling checker en de applicatie zelf. In het geval van iOS applicaties wordt de UIKit gebruikt ipv de AppKit. Deze lijken overigens erg sterk op elkaar.

CoreData bevat klassen tbv opslaan en teruglezen van objecten. Technisch gezien is CoreData niet nodig omdat het Foundation framework hier al in voorziet, echter wordt het gebruik van het CoreData framework hiervoor aanbevolen.

De hier genoemde frameworks (muv CoreData) vormen de basis voor de ontwikkeling van desktop en iOS applicaties. Daarnaast zijn er nog vele frameworks beschikbaar in XCode en is het ook mogelijk om eigen frameworks te ontwikkelen, die vanuit de eigen applicatie kunnen worden benaderd. Het linken naar andere frameworks gaat in XCode door de build phases van een project te selecteren en binnen dit window de ‘Link Binary With Libraries’ te selecteren. Hierin is standaard het Foundation framework te zien, maar er kunnen ook andere toegevoegd worden dmv de + knop. Er verschijnt dan een enorme lijst met frameworks waaruit het zonder achtergrond kennis lastig kiezen is.

Het advies is om eerst de documentatie door te nemen, die vanuit XCode ingezien kan worden. Ga hiervoor naar Help en selecteer XCode Help. Gebruik de zoekfunctie om informatie op te zoeken over het onderwerp waar informatie over nodig is, zoals strings, timers, etc. Met name in de System Guides is relevante programmeerinfo te vinden over het ingevoerde onderwerp, zoals de Programming Guide. Ook is veel sample code te vinden, maar veel is tamelijk oud en daardoor minder interessant. Met name de programming guides voor Low-level file management, Strings en Numbers and Values.

Na het toevoegen van een framework in XCode kan het pas gebruikt worden als de betreffende header files worden geimporteerd. De veilige manier is om de header file met dezelfde naam als het framework te importeren, deze importeert dan alle header files van het framework. De filenaam moet tussen < en > geschreven worden, ten teken dat het een framework header file betreft.