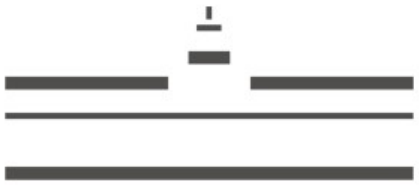


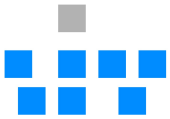
Westfälische
Wilhelms-Universität
Münster

Programmierung für mobile Endgeräte

Objective-C (Klassen und Objekte)

wissen.leben
WWU Münster

Name: Patrick Förster



ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Wiederholung

- Xcode – Erste Schritte: Workspace, Command-Line-Tool Projekt, SCM
- Objective-C:
  - Nachrichten ↔ Methoden

```
[object methodenname: param1 alias2: param2 … aliasN: paramN];
```

- Kontrollstrukturen, insb. „foreach“-Iteration, Exception-Handling

```
@try {
    // kritischer Code
}
@catch (NSException *exception) {
    // Fehlerbehandlung
}
```

- @-Strings

```
NSString* string = @"Hello World";
NSString* string = [NSString stringWithFormat:@"%i", 555];
NSMutableString* mutable = [NSMutableString string];
```

Objective-C: Klassen

- Klassen werden in Objective-C zweiteilig definiert:
  - *Interface*: Das Interface definiert die von außen mögliche Interaktion mit dem Objekt sowie die Attribute, die den Status eines Objekts bestimmen
  - *Implementation*: In der Implementierung werden die innerhalb des Interfaces definierten Methoden ausprogrammiert
- Interface und Implementierung werden in zwei verschiedenen Dateien definiert
- Die Interface-Datei wird auch Header-Datei genannt und endet mit .h
- Die Implementierungs-Datei endet auf .m (häufig mit m="message" erklärt)
- In der Regel sehen andere Klassen oder Interfaces nur den Interface-Teil
- Ein Interface wird mit Hilfe von #import in einer anderen Datei bekannt gemacht

```
@interface Phone  {
   ...
}
...
@end
```
Phone.h

```
#import "Phone.h"

@implementation Phone
...
@end
```
Phone.m

Objective-C: Klassen (II)

- Objektive-C kennt nur einfache Vererbung; Mehrfachvererbung wird nicht unterstützt
- Bei Vererbung wird die Elternklasse mit einem Doppelpunkt getrennt nach dem Klassennamen angegeben
- Ähnlich wie Object in Java existiert mit NSObject in Objective-C eine Basisklasse für Objekte
- Anders als in Java muss die Vererbung explizit angegeben werden!

```
@interface Phone : NSObject {
}
@end
```

Objective-C: Klassen (III)

- Innerhalb der geschweiften Klammen einer Schnittstelle können Objektattribute spezifiziert werden
- Klassenattribute kennt Objective-C nicht
- Analog zu den meisten OOP-Sprachen ist ein Attribut durch die Angabe seines Typs und einen Namen definiert
- Zudem kann optional eine Sichtbarkeit angegeben werden, die angibt wer direkt auf das Attribut zugreifen darf. Es gibt drei Möglichkeiten:
  - @public: auf das Attribut kann von überall zugegriffen werden
  - @protected: auf das Attribut kann nur innerhalb der Klasse und aller erbenden Klassen zugegriffen werden (default)
  - @private: Zugriff nur innerhalb der definierenden Klasse

```
@interface Phone : NSObject {
   @protected
      int lastNumberCalled;
}
@end
```

Objective-C: Klassen (IV)

- Interface: Methoden

```
+/-(RückgabTyp) methodenName: (paramTyp1) param1 paramAlias2:  (paramTyp2)
param2 … paramAliasN:  (paramTypN) paramN;

// Beispiel
-(void) callNumber: (int) number withMessage: (NSString*) message
```

- - definiert Objekt-Methoden, also Methoden, die nur über ein Objekt aufgerufen werden können
- + definiert Klassen Methoden
- Es gibt keine Sichtbarkeit: Alle innerhalb eines Interfaces definierten Methoden sind „public“
- Die Implementierung der Beispielmethode könnte innerhalb von @implementation...@end in etwa wie folgt aussehen:

```
-(void) callNumber: (int) number withMessage: (NSString*) message {
    NSLog(@"Called number %i with message: %@", number, message);
    lastNumberCalled = number;
}
```

Objective-C: Objekte

- Erzeugen eines Objektes

```
#import "Phone.h"
int main (int argc, const char * argv[]) {
    Phone* phone = [Phone alloc];
    [phone init];
    [phone …];
}
```

- Objective-C unterscheidet zwischen der Allokation von Speicher für ein Objekt und der Initialisierung des selbigen
- Beide Schritte müssen „händisch“ umgesetzt werden
- Der Datentyp * bezeichnet einen Zeiger auf ein Objekt des angegebenen Typs, das im dynamischen Speicher (Heap) liegt
- Konstruktoren, die in anderen OOP-Sprachen automatisch aufgerufen werden, sobald Speicherplatz für ein Objekt reserviert wird, gibt es in Objective-C nicht
- Die alloc-Methode übernimmt das Reservieren von Speicher
- Alle von NSObject erbenden Klassen verfügen mit init über eine Methode, die gewissermaßen als „Pseudo-Konstruktor“ zu verstehen ist

Objective-C: Objekte (II)

- Anstelle der zweischrittigen Erzeugung eines Objektes werden alloc und init zumeist verknüpft aufgerufen

```
Phone* phone = [[Phone alloc] init];
```

- Oder noch kürzer mit der Klassenmethode new:

```
Phone* phone = [Phone new];
```

- Um nun eigene Attribute zu initialisieren muss die init-Methode überschrieben oder ein neuer Pseudokonstruktor angelegt werden (new funktioniert nur in Kombination mit init!)
- Analog zu eigenen können geerbte Methoden innerhalb @implementation... @end überschrieben werden:

```
-(void) dealloc {
    NSLog(@"Deallocation of %@", self);
    [super dealloc];
}
```

- Die Methode dealloc ist von NSObject geerbt und wird aufgerufen sobald ein Objekt freigegeben wird. Ein Objekt sollte hier seinerseits die eigenen Attribute freigeben und vor allem die Freigabe „nach oben“ delegieren (mehr dazu später)

Objective-C: Konstruktoren

- Häufig genügt es den Konstruktor der NSObject-Klasse zu überschreiben:

```
-(id) init {
    if ((self = [super init])) {
        lastNumberCalled = -1;
    }

    return self;
}
```

- id ist ein spezieller Datentyp, der so viel bedeutet wie: „Zeiger auf irgendeinen Datentyp“
- Ein Aufruf der Art [[Phone alloc] init] liefert demnach eigentlich ein Objekt vom Typ id und nicht Phone zurück
- Die Rückgabe wird also impliziert auf den eigentlichen Datentyp gecastet und ist somit ein Beispiel für die „schwache Typisierung“ von Objective-C
- Hinter self verbirgt sich in der Regel ein Zeiger auf das Objekt, auf dessen Kontext der aktuelle Methodenaufruf ausgeführt wird
- Aber...

Objective-C: Konstruktoren (II)

- … die Initialisierung kann folgende drei Rückgaben liefern:
  - Einen Zeiger auf das aktuelle Objekt
  - nil um zu signalisieren, dass die Initialisierung schief gelaufen ist
  - Einen Zeiger auf ein anderes Objekt
- Aufgrund von 3. findet man in vielen Lehrbüchern die explizite Initialisierung von self durch den Aufruf des übergeordneten Konstruktors
- Natürlich können mehrere Konstruktoren mit zudem unterschiedlichen Argumente definiert werden (Konstruktoren sind normale Methoden!)
- Nach Konvention müssen solche Methoden mit einem „init“ beginnen
- Es sollte allerdings nur einen Hauptkonstruktor geben, der von den anderen gefüttert wird, anstatt dass diese eine eigene Initialisierung vornehmen. Es gilt:
  - Gibt es in einer der Elternklassen bereits einen Konstruktor, der für die aktuelle Klasse ausreichend ist, sollte kein neuer angelegt werden
  - Falls ein neuer Hauptkonstruktor angelegt wird, muss der der Elternklasse überschrieben werden

Objective-C: Selektoren

- Ein Selektor ist der Name einer Methode samt seiner Paramateraliase
- In Objektive-C ist ein Selektoren ein „First-Class“-Datentyp

```
SEL selector = @selector(callNumber:withMessage:);
```

- Hier wird der Selektor zur Methode callNumber spezifiziert, die bspw. innerhalb der Phone-Klasse definiert wurde
- Die NSObject Methode performSelector kann dazu genutzt werden, Methoden anhand eines Selektors aufzurufen:

```
[object performSelector:selector];
```

- NSObject bietet noch zwei weitere Methoden, denen ein bzw. zwei Argumente mitgeben werden können (NSInvocation bietet noch mehr Flexibilität)
- Die Methoden erwarten Objekte vom Typ id. Dabei ist Vorsicht geboten: Es findet keine Typüberprüfung statt

```
[phone performSelector:selector withObject:555 withObject: @"Shoe"];
→ „Called number 555 with message: Shoe“
[phone performSelector:selector withObject:@“a string“  withObject: @"Shoe"];
→ „Called number 30840 with message: Shoe“
```

Objective-C: Selektoren (II)

- Selektoren sind unabhängig von Klassen oder Objekten
- Die Möglichkeit eine bestimmte Methode unabhängig von einem bestimmten Typ aufzurufen, wird oft auch als „Duck-Typing“ bezeichnet:

```
@interface Phone: NSObject

-(void) quak;

@end
```

```
@interface CoffeeMashine : NSObject

-(void) quak;

@end
```

```
Phone* phone = ...;
CoffeeMashine* mashine = ...
NSArray* array = [[NSArray alloc] initWithObjects: phone, mashine, nil];

SEL selector = @selector(quak);

for (int i = 0; i < [array count]; i++) {
    [[array objectAtIndex: i] performSelector:selector];
    ... //  aufräumen
}
```

- Beide Klassen können quarken, obwohl sie keine Ente sind!

Objective-C: Properties

- Da in der Regel Attribute als @protected oder @private deklariert werden, gibt es per se keine Möglichkeit von außerhalb auf diese zuzugreifen
- Zumeist werden für Attribute – je nach Lese/Schreibemöglichkeit – Getter/Setter-Methoden definiert, die nichts weiter tun (sollten!) als den Wert eines Attributes zu setzen oder zu liefern
- In Objective-C müssen diese Methoden nicht explizit angeben werden; es genügt die Definition von sogenannten *Properties*

```
@interface Phone : NSObject {
      int lastNumberCalled;
}
// Property defiinieren
@property int lastNumberCalled;
...
@end
```
Phone.h

```
@implementation Phone

…
// mit @synthesize autogenerieren
@synthesize lastNumberCalled;
...
@end
```
Phone.m

- Der Zugriff auf eine Property erfolgt dann über die Punktnotation

```
Phone* phone = [Phone new];

phone.lastNumberCalled = 10;
```

Objective-C: Properties (II)

- Properties sind kein Muss
- Es können auch wie gehabt Getter/Setter-Methoden definiert werden
- Die Punktnotation zum Zugriff ist dann allerdings nicht mehr möglich
- Es besteht auch die Möglichkeit durch das Weglassen von @synthesize eine Property zu definieren, für die man die Methoden eigenständig implementiert
- Die Getter-Methode muss dabei genauso heißen wie das Attribut, während die Setter-Methode mit „set“-Prefix definiert werden muss
- Attribute, die als Property spezifiziert werden, müssen im Interface nicht innerhalb von {} angegeben werden
- Falls das Attribute aus irgendwelchen Gründen anders heißen soll, als die zugehörigen Methoden, so kann nach @synthesize ein abweichender Name definiert werden

```
@synthesize lastNumberCalled = _lastNumberCalled;
```

- Innerhalb eines Objektes muss dann der abweichende Name verwendet werden
- Wird ein Attribut mit self. angesprochen wird die Property-Methode aufgerufen!

Objective-C: Properties (III)

- Der @property- Anweisung können Einstellungen zugeteilt werden, die das Verhalten der Getter-, bzw. Setter-Methode beeinflussen:

```
@property((readwrite | readonly), (assign | copy | retain | strong | weak), nonatomic) int lastNumberCalled;
```

- Generell:
  - readwrite: Getter-, sowie Setter werden implementiert (default)
  - readonly: Nur die Getter-Methode wird umgesetzt
- Verhalten der Setter-Methode bzgl. Speicherverwaltung
  - assign/weak: Einfache Zuweisung des Wertes (primitive Datentypen) oder der Referenz (Objekte) (default)
  - retain/strong: (nur für Objekte) Zuweisung der Referenz mit Erhöhung des Referenzzählers
  - copy: (nur für Objekte) eine Kopie des Objektes wird zugewiesen
- Multithreading:
  - nonatomic: Kein Blockieren des Threads bei Aufruf der Getter- oder Setter-Methode (per Default sind alle Properties atomic)

Objective-C: Protokolle

- Ein Protokoll ist in Objective-C das Pendant zu einem Interface in bspw. Java
- Innerhalb eines Protokolls werden Methoden definiert, die von allen das Protokoll unterstützenden Klassen implementiert werden müssen
- Neben obligatorischen sind allerdings auch optionale Protokoll-Methoden möglich

```
@protocol App // <Protocol, ....>
@required // default
-(void) run;
@optional
-(NSString*) infos;
@end

// Einsatz
@interface AngryBirds  <App> ... @end
```

- In anderen Programmiersprachen werden Schnittstellenmethoden des Öfteren als „Stubs ohne Logik“ implementiert, nur weil die Schnittstelle das Vorhandensein der Methode vorschreibt; durch @protocol wird mehr Flexibilität geboten