Objective-C

- C als Teilmenge incl. Präprocessing

  `#import` für Objective-C Header

  `#include` für C/C++ Header

- z.B. wichtig für IB - Verknüpfungspunkte:

```
#define IBAction void
#define IBOutlet
```

Objective-C

- C als Teilmenge incl. Präprocessing

  `#import` für Objective-C Header (no guards needed)

  `#include` für C/C++ Header

- z.B. wichtig für IB - Verknüpfungspunkte:

```
#define IBAction void
#define IBOutlet
```

Objective-C

- Klassen bestehen aus

  Schnittstellenbeschreibung (zumeist in einem gleichnamigen Headerfile *.h)

```
@interface X : Super { // root base: NSObject
   double d;  // Memberdaten/Instanzvariablen
   int i;
   id o; // irgendein Objektzeiger
   NSString* name;
}
- foo; // Memberfunktion: ohne Returntyp: id !
+ (int) numberOfXs; // Klassenfunktion
@end
```

Objective-C

- Aufruf von Methoden ~ Senden einer Nachricht

```
[someX foo];
[X numberOfXs];
```

- Objekterzeugung

```
X * someX = [[X alloc] init];
```

- Objekte leben IMMER im Heap !
- in -Methoden: direkter Zugang zu Memberdaten

Objective-C

- ... und
  - Implementation (zumeist in einem gleichnamigen Impl.file *.m)

```
@implementation X  // no base no body
- foo {
   // whatever foo has to do
}
+ (int) numberOfXs {
   // whatever numberOfXs has to do
}
@end
```

ohne Returntyp:
(id) nicht (void)

Objective-C

- Instanzmethoden starten mit -
- ungewöhnliche Methodensyntax:

Resultat- und Parametertypen in Klammern

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation
               withDamage:(BOOL)damaged;

- (void)splitViewController:(UISplitViewController*)svc
     willHideViewController:(UIViewController *)vc
          withBarButtonItem:(UIBarButtonItem *) bbi
       forPopoverController:(UIPopoverController *)pc;
```

mehrere Parameter durch (lesbaren) Teil des Methodennamens und :

Objective-C

- Klassenmethoden starten mit +

  Beispiele: Allocation, Singletons, Utilities

```
+ (id)alloc;
// makes space for an object of the receiver's
// class (always pair with init)
+ (id)motherShip; // returns the one and only,
// shared (singleton) mother ship instance
+ (int)turretsOnShipOfSize:(int)shipSize;
// informational utility method
```

- KEIN Zugang zu Instanzvariablen (a la static)

Objective-C

- Instanzvariablen sind per default `@protected`:
  Klasse selbst und Ableitungen können zugreifen!
- können auch `@private` oder `@public` sein

```
@interface MyObject : NSObject {
    int foo;  // @protected
@privat
    int eye;
@protected
    int bar;
@public
    int forum; int apology;
@private
    int jet;
}
```

Objective-C

- ABER: besser **nicht** benutzen und statt dessen alles `@private` als sog. `@property`
  - **setter** und **getter** per Namenskonvention:
  - **getter** heißt wie die *Property*
  - **setter** heißt `set`*Property*

```
@interface MyObject : NSObject {
  @private int eye;
}
- (int)eye;
- (void)setEye:(int)anInt;  // NOT seteye!!!
@end
```

Objective-C

- deshalb Objective-C-Konvention:
  Instanzvariablen immer klein!
- NUR mit getter/setter kann man sog. Dot-Notation verwenden:

```
someObject.eye = newEyeValue;
// set the instance variable
int eyeValue = someObject.eye;
// get the instance variable's current value
```

- gänzlich anders als in C++, Java !

Objective-C

- setter/getter kann man wie oben deklarieren, besser ist es aber stattdessen sog. Properties zu deklarieren:

```
@interface MyObject : NSObject {
    @private int eye;
}

@property int eye; // declares setter/getter !!!

@end
```

- die `@property`-Deklaration ersetzt NICHT die Deklaration der Instanzvariablen!

Objective-C

- read-only Properties (kein setter):

```
@interface MyObject : NSObject {
   @private int eye;
}

@property (readonly)int eye;
// declares getter only !!!

@end
```

- write-only gibts nicht

Objective-C

- Property muss nicht namentlich identisch zu einem Instanzvariablennamen sein! Es muss nicht mal eine separate Instanzvariable für die property geben

```
@interface MyObject : NSObject {
    @private  int p_eye; // anderer Name
}
@property int eye;
@end
```

- oder

```
@interface MyObject : NSObject {
}
@property (readonly) int eye;
@end
```

Objective-C

- Abbildung erfolgt in der Implementation der getter/setter im entspr. .m-File:

```
// property name == instance name
@implementation MyObject
- (int)eye {
  return eye;
}
- (void)setEye:(int)anInt {
  eye = anInt;
}
@end
```

Objective-C

- oder:

```
// property name != instance name
@implementation MyObject
- (int)eye {
  return p_eye;
}
- (void)setEye:(int)anInt {
  p_eye = anInt;
}
@end
```

Objective-C

- oder:

```
// no instance var. for property name
@implementation MyObject
- (int)eye {
  return <some_calculated_value>;
}
// no setter at all
}
@end
```

Objective-C

- im Normalfall (Property-Name == Instanzname) kann man die Implementation sogar automatisch erzeugen lassen:

```
@implementation MyObject
@synthesize eye;
@end
```

- das geht sogar im Falle abweichender Namen:

```
@synthesize eye = p_eye; // mapping
```

- PREFER USING @synthesized @propertys
- 3. Fall (keine Inst.var.) natürlich NICHT synthetisierbar.

Objective-C

- eigene Implementationen sind trotzdem möglich und ersetzen die generierten:

```
@synthesize eye;
- (void)setEye:(int)anInt {
    if (anInt > 0) eye = anInt;
}
```

- Werte prüfen/adaptieren
- weitere Operationen hinterlegen (lassen)
  Memory Management !!! (später)

Objective-C

- es ist häufig sinnvoll Dot-Notation auch in Methoden zu verwenden, weil dann getter/setter benutzt werden!

```
int x = eye;
// in einer Methode ist etwas anderes als:
int x = self.eye;
```

- ABER VORSICHT: Dot-Notation in der Impl. der getter/setter führt zu endloser Rekursion:

```
- (void)setEye:(int)anInt {
        self.eye = anInt; // endless loop
}
```

Objective-C

- kann auch beim getter passieren:

```
- (int)eye {
    if (self.eye > 0) // endless loop
        { return self.eye; } // dito
  else { return -1; }
}
```

Objective-C

- Dot-Notation lehnt sich an Zugriff auf Strukturen an.
- Variablen von Strukturtypen sind aber WERTE!

```
typedef struct {
  float x;
  float y;
} Point;

@interface Bomb ...
@property Point position;
@end

Bomb *bomb = ...;

bomb.position.x >= leftEdge
```

Objective-C

- private Properties (alles im .m-File), im *.h File muss natürlich entsprechende Instanzvariable deklariert werden

```
@interface MyObject()
@property double myEyesOnly;
@end
@implementation MyObject
@synthesize eye, myEyesOnly;
@end
```

so kann ein Interface um (private) Properties/ Methoden (NICHT Instanzvariablen) erweitert werden

- myEyesOnly kann nur via self.myEyesOnly benutzt werden.

back

Objective-C

- Dynamic Binding

```
NSString *s = ...; // "static" typed
id obj = s;        // "dynamic" typed
```

- NICHT id* , id ~ „special void*“

```
[s doThis];    // same as
[obj doThis];
```

- statische Typisierung ermöglich lediglich stärkere Kontrolle während der Übersetzung
- wenn man den (Mindest-)Typ kennt, sollte man ihn auch statisch (im Quellcode) verwenden !

Objective-C

- Casting ist möglich, aber nicht sicher:

```
id obj = ...;
NSString *s = (NSString *)obj;
```

- aber wenn `obj` kein `NSString` ist: crash ahead

Objective-C

- wie üblich bei OOP: Ableitung kann alles aus den Basisklassen:

```
@interface Vehicle
- (void)move;
@end

@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];
```

Objective-C

- Polymorphie:

```
Vehicle *v = s; // isA !
[v shoot]; // Warnung (kein Fehler)!
```

```
id obj = ...;
[obj shoot]; // KEINE Warnung: möglich ist's
```

- aber besser KEINE Messages an id's senden, Fehler wird erst zur Laufzeit sichtbar

Objective-C

- (nur) Warnung bei

```
NSString *hello = @"hello";
[hello shoot]; // how could it ever?
```

durch „()-Magie“ (s.o. private properties) können in Objective-C Klassen um Methoden angereichert werden - sog. **Categories**

- KEINE Warnung bei

```
Ship *helloShip = (Ship *)hello;
[helloShip shoot];  // compiler cannot help
[(id)hello shoot];  // dito
```

Objective-C

- Introspection
  - id zu benutzen ist dennoch sinnvoll
  - man kann konkreten Typ erfragen
- alle Ableitungen von `NSObject` verstehen:

  `isKindOfClass: T` - bist du wenigstens vom Typ `T`?

  `isMemberOfClass: T` - bist du genau vom Typ `T`?
- die Klassenmethode `class` liefert eine Typbeschreibung vom (Meta-)Typ `Class`, diese muss anstelle von `T` verwendet werden

Objective-C

- (static) Cast absichern:

```
if ([obj isKindOfClass:[NSString class]]) {
    NSString *s = [(NSString *)obj
        stringByAppendingString:@"xyzzy"];
}
```

Objective-C

- man kann auch unabhängig von Klassen erfragen:

  `respondsToSelector: S` - verstehst du die Nachricht `S`?

- Selektoren

```
if ([obj respondsToSelector:@selector(shoot)])
{
        [obj shoot];
}
```

- Methode incl. aller :Argumentnamen
- Selektoren haben den Objective-C-Typ `SEL`

Objective-C

```
SEL shootSelector = @selector(shoot);
SEL moveToSelector = @selector(moveTo:);
```

- man kann SEL's direkt aufrufen:

```
[obj performSelector:shootSelector];
[obj performSelector:moveToSelector
          withObject:coordinate];
```

- DEMO: `functions.xproj`

Objective-C

- `nil` ein leerer (Objekt-)Zeiger
- Vorinitialisierung aller Objektvariablen auf `nil`
- kann getestet werden:

```
if (obj) ...
```

- Messages an `nil` sind erlaubt und tun NICHTS (ggf. Returnwert 0)

Foundation Framework

- die wichtigsten aller Klassen !
- `NSObject`
  - Basisklasse für (fast) alles!
  - kümmert sich um Memory Management, Introspektion, etc.

    `(NSString*) description` für %@ in Textausgaben
  - für eigene Klassen geeignet redefinieren

Foundation Framework

- NSString
  - für Strings aller Art
  - nie const char* verwenden
  - Literale mit @"foo"
  - NSStrings sind unveränderlich

```
display.text =
[display.text stringByAppendingString:digit];
// a new String, what about the old one?
```

- diverse Methoden siehe Ref.

http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/Reference/NSString.html