Foundation Framework

- es gibt auch veränderliche Strings:

```
NSMutableString *mutString =
    [ [NSMutableString alloc]
                        initWithString:
      @"0."];

[mutString appendString:digit];
```

- nur selten benutzt (z.B. in inner loops)
- also besser mit `NSString` arbeiten

Foundation Framework

- NSNumber
  - Object wrapper um elementare Typen int, float, double, BOOL, etc.

```
NSNumber *num =
      [NSNumber numberWithFloat:36.5];

float f = [num floatValue];
```

Foundation Framework

- NSData
  - ein beliebiger Sack voll Bits für alles mögliche (Images, untyped data, ...)

Foundation Framework

- NSDate
  - Datum/Zeit
  - siehe auch NSCalendar, NSDateFormatter, NSDateComponents

Foundation Framework

- NSArray
  - Felder von Objekten
  - unveränderlich!
  - wichtige Methoden:

```
+(id)arrayWithObjects:(id)firstObject, ...;
-(int)count;
-(id)objectAtIndex:(int)index;
-(void)makeObjectsPerformSelector:(SEL)aSelector;
-(NSArray *)sortedArrayUsingSelector:(SEL)aSelector;
-(id)lastObject; // nil wenn leer
```

Foundation Framework

- NSMutableArray
  - Felder von Objekten
  - veränderlich
  - wichtige Methoden:

```
-(void)addObject:(id)anObject;
-(void)insertObject:(id)anObject atIndex:(int)index;
-(void)removeObjectAtIndex:(int)index;
```

- DEMO: sort.xproj

Foundation Framework

- NSDictionary
  - Key/Value Paare
  - unveränderlich!
  - Keys: Objekte, die implementieren müssen:

    ```
    -(NSUInteger)hash
    -(BOOL)isEqual:(NSObject *)obj
    ```
  - Keys meist NSString
  - wichtige Methoden:

    ```
    -(int)count;
    -(id)objectForKey:(id)key;
    -(NSArray *)allKeys;
    -(NSArray *)allValues;
    ```

Foundation Framework

- NSMutableDictionary
  - veränderliche Version von NSDictionary

```
-(void)setObject:(id)anObject forKey:(id)key;
-(void)removeObjectForKey:(id)key;
-(void)addEntriesFromDictionary:
                    (NSDictionary *)otherDictionary;
```

Foundation Framework

- NSSet
  - ungeordnete Menge von Objekten
  - unveränderlich!
  - wichtige Methoden:

```
-(int)count;
-(BOOL)containsObject:(id)anObject; // dieses selbst
-(id)anyObject;
-(void)makeObjectsPerformSelector:(SEL)aSelector;
-(id)member:(id)anObject; // ein gleiches wrt isEqual
```

Foundation Framework

- NSMutableSet
  - veränderliche Version von NSSet
  - wichtige Methoden:

```
-(void)addObject:(id)anObject;
-(void)removeObject:(id)anObject;
-(void)unionSet:(NSSet *)otherSet;
-(void)minusSet:(NSSet *)otherSet;
-(void)intersectSet:(NSSet *)otherSet;
```

Foundation Framework

- warum immutable/mutable Varianten?
  - einfachere/effizientere Speicherverwaltung
  - wann immer möglich, immutable bevorzugen
  - oft gibt es Methoden, die immutable Objekte erzeugen

z.B.

```
newstr = [str stringByAppendingString:@"xyz"];
array = [NSArray arrayWithObjects:o1, o2, o3, nil];
dict = [NSDictionary dictionaryWithObjectsAndKeys:
        @"value1", @"key1",
        @"value2", @"key2", nil];
```

Foundation Framework

- Enumeration über Container:

```
NSArray *myArray = ...;
for (NSString *string in myArray) {
  double value = [string doubleValue];
   // crash wenn string kein NSString ist
}
```

Foundation Framework

- Enumeration über Container:

```
NSSet *mySet = ...;
for (id obj in mySet) {
  // do something with obj, but make sure you
  // don't send it a message it does not respond to
  if ([obj isKindOfClass:[NSString class]]) {
  // send NSString messages to obj with impunity
  } else {
    // handle others
  }
}
```

Foundation Framework

- Enumeration über Container:

```
NSDictionary *myDictionary = ...;
for (id key in myDictionary) {
  // do something with key here
  id value = [myDictionary objectForKey:key];
  // do something with value here
}
```

Foundation Framework

- Property Lists (PL)
  - jedes beliebig strukturiertes Objektgefüge, welches NUR `NSArray`, `NSDictionary`, `NSNumber`, `NSString`, `NSDate`, `NSData` - Objekte enthält (NICHT `NSSet`!)
  - z.B.: ein `NSArray` von `NSDictionarys` mit `NSString` keys und `NSNumber` values ist eine PL
  - es gibt ein paar Methoden im SDK, die auf PLs operieren, z.B.

```
[plist writeToFile:(NSString *)path atomically:(BOOL)];
// plist muss PL sein
```

Foundation Framework

iOS Developer Library

(NSArray/NSDictionary) writeToFile:atomically:

Writes a property list representation of the contents of the dictionary to a given path.

`- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag`

Parameters
path
The path at which to write the file.

If path contains a tilde (~) character, you must expand it with `stringByExpandingTildeInPath` before invoking this method.

flag
A flag that specifies whether the file should be written atomically.

If flag is `YES`, the dictionary is written to an auxiliary file, and then the auxiliary file is renamed to path. If flag is `NO`, the dictionary is written directly to path. The `YES` option guarantees that path, if it exists at all, won't be corrupted even if the system should crash during writing.

Return Value
`YES` if the file is written successfully, otherwise `NO`.

Discussion
This method recursively validates that all the contained objects are property list objects (instances of `NSData`, `NSDate`, `NSNumber`, `NSString`, `NSArray`, or `NSDictionary`) before writing out the file, and returns `NO` if all the objects are not property list objects, since the resultant file would not be a valid property list.

If the dictionary's contents are all property list objects, the file written by this method can be used to initialize a new dictionary with the class method `dictionaryWithContentsOfFile:` or the instance method `initWithContentsOfFile:`.

For more information about property lists, see Property List Programming Guide.

Foundation Framework

- Spezialfall: `NSUserDefaults`
  - leichtgewichtiger Speicher für Property Lists.
  - im Prinzip ein NSDictionary welches die App überdauert.
  - keine Datenbank, eher für kleine Sachen wie User Preferences

Returns the shared (singleton) defaults object.

```
[[NSUserDefaults standardUserDefaults]
    setArray:rvArray forKey:@"RecentlyViewed"];
```

  - set/get Methoden für alle PL-Typen: (z.B.)

```
-(void)setDouble:(double)aDouble forKey:
                                (NSString *)key;
-(double)doubleForKey:(NSString *)key;
....
```

Foundation Framework

- Spezialfall: `NSUserDefaults`

```
-(NSInteger)integerForKey:(NSString *)key;
// NSInteger: typedef to 32 or 64 bit int
- (void)setObject:(id)obj forKey:(NSString *)key;
// obj must be a Property List
- (NSArray *)arrayForKey:(NSString *)key;
// will return nil if value for key is not NSArray
```

- permanent machen mit:

  `[[NSUserDefaults standardUserDefaults] synchronize];`

- nicht App-übergreifend!
- Einlesen bei erster Anfrage nach einem Schlüssel implizit !!

Memory Management

- Plattform hat sehr begrenzten Speicher - und Leistung: keine Garbage Collection !!!
- Speicher (für Objekte) immer mit `alloc` (auf dem Heap) beschaffen
- danach alle Instanzvariablen auf Null!
- Initialisierung immer mit `init` (oder einer Methode beginnend mit `init...`)
- in jeder Klasse muss es einen designated initializer (DI) geben

Memory Management

- bei Ableitungen muss dieser den DI der Basis rufen und sicherstellen, dass dieser nicht fehlschlägt (indem er `nil` liefert)
- `NSObject`'s DI heißt `init` und wird demnach immer eingeerbt!
- ein DI sollte nur die allernötigsten (ideal keine) Parameter haben (ggf. Standardannahmen treffen) siehe: http://stackoverflow.com/questions/5246622/uitableviewcontroller-init-vs-initwithstyle

Memory Management

- man kann neben dem DI weitere (convenient initializers (CI) definieren) mit weiteren Argumenten, diese sollten immer den eigenen DI rufen

```
[self init];
```

- alle Initializer sollten `id` liefern (nicht statisch typisiert sein) kein Casting nötig beim Aufruf

```
[super init];
```

- der Aufrufer kann & sollte statisch typisieren:

```
MyObject *obj = [[MyObject alloc] init];
```

Memory Management

- woher weiss man, ob `super init` gelungen ist?
  - liefert `nil` beim Scheitern:

```
@implementation MyObject
-(id)init {
        if ([super init]) {
          // initialize our subclass here
          return self;
      } else {
          return nil;
      }
  }
@end
```

Memory Management

- besser (weil nur ein exit point):

```
@implementation MyObject
- (id)init {
    if (self = [super init]) {
        // initialize our subclass here
    }
    return self; // kann auch nil sein
}
@end
```

Memory Management

- Zuweisung an self NUR im DI!
- EIN init-Ruf pro Object!
- ein CI könnte sein (Semantik: man kann einem Objekt auch ein Feld von irgendwas [z.B. in einer Property myArray] mitgeben)

```
@implementation MyObject
- (id)initWithArray:(NSArray *)anArray {
    self = [self init]; // ruft den DI
    self.myArray = anArray;
    // FRAGE: was, wenn self == nil ???
    return self;
}
@end
```

Memory Management

- ein weiteres Beispiel:
  - UIView muss immer mit einem Rechteck (seiner Ausdehnung auf dem Display) initialisiert werden,

UIView's DI ist also:

```
-(id)initWithFrame:(CGRect)aRect;
...
UIView *view =
[[UIView alloc] initWithFrame:myFrame];
```

Memory Management

- in einer Ableitung `MyView : UIView` hat man die Wahl zwischen

  A) der DI soll wie die Basisklasse ein `GCRect` verwenden, ein CI soll mit einer Figur (Shape) rufbar sein:

```
@implementation MyView
-(id)initWithFrame:(CGRect)aRect { // DI
   if (self = [super initWithFrame:aRect]) {
   // initialize my subclass here
   }
   return self;
}
-(id)initToFit:(Shape *)aShape { // CI
   CGRect fitRect = [MyView sizeForShape:aShape];
   return [self initWithFrame:fitRect];
} @end
```

Memory Management

- und

  B) die Shape-Variante soll zum DI werden, die andere CI:

```
@implementation MyView
-(id)initWithFrame:(CGRect)aRect { // CI
   return [self initToFit:[self defaultShape]];
}
-(id)initToFit:(Shape *)aShape { // DI
   CGRect fitRect = [MyView sizeForShape:aShape];
   if (self = [super initWithFrame:fitRect]) {
      // initialize my subclass here
      return self;
}
@end
```

Memory Management

- `alloc/init` ist nicht der einzige Weg, neue Objekte zu erhalten, zahlreiche SDK-Funktionen liefern neu erzeugte Objekte:

```
NSString *newDisplay  = [display.text
                          stringByAppendingString: digit];
NSArray  *keys        = [dictionary allKeys];
NSString *lowerString = [string lowercaseString];
NSNumber *n           = [NSNumber numberWithFloat:42.0];
NSDate   *date        = [NSDate date]; // jetzt !
```

Memory Management

- wer gibt all diese Objekte wieder frei (incl. mit der `alloc/init` erzeugten) ?
  - KEINE Garbage Collection auf der iOS Plattform! (wohl aber auf Mac)
  - stattdessen: Referenzzählung!
  - ein einfaches und (Energie-)effizientes Schema :-)
- Referenzzählung wird von `NSObject` unterstützt
  - alle Objekte haben einen sog. retain count
  - arbeitet aber nur korrekt, wenn einige Regeln eingehalten werden !

Memory Management

Regeln der iOS Speicherverwaltung

1. Wer einen Objektzeiger aufbewahren will (für dessen spätere Benutzung), MUSS das Objekt besitzen (take ownership of it). Es kann beliebig viele Besitzer für einen Objektzeiger geben.
2. Man erlangt man Besitz an einem Objekt durch
   a) der Zeiger ist das Ergebnis einer Methode die namentlich anfängt mit `alloc...`, `copy...` oder `new...` ODER
   b) man ruft am Objekt die (`NSObject`-)Methode `retain`

Memory Management

Regeln der iOS Speicherverwaltung

3. Wird der Objektzeiger nicht mehr benötigt, MUSS man den Besitz aufgeben (give up Ownership).
4. Man gibt den Besitz auf

   durch Aufruf der Objekt (NSObject-)Methode `release`

   `release` darf man nur rufen, wenn man ein Objekt besitzt - nach `release` besitzt man es nicht mehr.

Memory Management

Regeln der iOS Speicherverwaltung

5. Es gibt die Möglichkeit, temporär Besitzer zu sein. Die Aufgabe des Besitzes wird dabei schon geplant, aber noch nicht vollzogen. (Hä? s.u.)
6. Gibt der letzte seinen Besitz an einem Objekt auf, wird es (automatisch) zerstört. Der Speicher wird dem Heap als frei zurückgegeben. Danach darf an das Objekt keine Nachricht mehr versendet werden - andernfalls Programmabsturz.

Memory Management

- Wer besitzt die Objekte aus Methoden, die nicht aus `alloc/copy/new` herauskommen?
  - Diese haben einen temporären Besitzer: Entweder man übernimmt selbst Ownership (`retain`), oder gibt den Zeiger an einen Aufrufer weiter oder das Objekt verschwindet - demnächst - automatisch !
  - temporärer Besitz wird durch `autorelease` veranlasst. Danach besitzt man das Objekt nicht mehr. Es besteht aber die Möglichkeit, dass jemand anderes den Besitz (per `retain`) übernimmt, BEVOR das Objekt zerstört wird.

Memory Management

- ein Beispiel

```
-(Money *)showMeTheMoney:(double)amount {
    Money *theMoney = [[Money alloc] init:amount];
    // diese Methode besitzt nun das Objekt und
   // müsste den Besitz auch wieder aufgeben
    return theMoney;
   // Fehler: Methode kommt ihrer Pflicht NICHT nach!
}

-(Money*)showMeTheMoney:(double)amount {
     Money *theMoney = [[Money alloc] init:amount];
   // diese Methode besitzt nun das Objekt und
   // müsste den Besitz auch wieder aufgeben
   [theMoney release]; // Pflicht erfüllt aber
   return theMoney; // Fehler: Zeiger auf ein Zombie
}
```

Memory Management

- die Lösung: temporärer Besitz

```
-(Money *)showMeTheMoney:(double)amount {
    Money *theMoney = [[Money alloc] init:amount];
    // diese Methode besitzt das Objekt und müsste
     // den Besitz auch wieder aufgeben
    [theMoney autorelease]; // Pflicht erfüllt
    return theMoney; // OK
}
```

der Rufer kann entscheiden, ob er den Besitz übernimmt bevor das Objekt verschwindet, z.B.

```
Money *myMoney = [bank showMeTheMoney:4500.00];
[myMoney retain]; // now it‘s mine
```

Memory Management

- noch ein Beispiel:

```
// schlecht:
@implementation MyObject
-(NSArray *)coolCats {
  NSMutableArray *returnValue =
                        [[NSMutableArray alloc] init];
  [returnValue addObject:@"Steve"];
  [returnValue addObject:@"Ankush"];
  [returnValue addObject:@"Sean"];
  return returnValue;
  // Fehler: Besitz nicht aufgegeben!
}
@end
```

Memory Management

```
// gut:
@implementation MyObject
- (NSArray *)coolCats {
    NSMutableArray *returnValue =
                    [[NSMutableArray alloc] init];
    [returnValue addObject:@"Steve"];
    [returnValue addObject:@"Ankush"];
    [returnValue addObject:@"Sean"];
    [returnValue autorelease];
    return returnValue;
}
@end
```

```
return [returnValue autorelease];
```

Memory Management

```
// besser: gar nicht erst besitzen
- (NSArray *)coolCats
{
    NSMutableArray *returnValue =
    [NSMutableArray array];
    // liefert einen autoreleased Zeiger
    [returnValue addObject:@"Steve"];
    [returnValue addObject:@"Ankush"];
    [returnValue addObject:@"Sean"];
    // NICHT: [returnValue autorelease];
    // ich bin nicht der Besitzer!
    return returnValue;
}
@end
```

Memory Management

```
// am besten: immutable Container
@implementation MyObject
- (NSArray *)coolCats {
    return [NSArray arrayWithObjects:
                        @"Steve", @"Ankush",
                        @"Sean", nil
                ];
    // liefert einen autoreleased Zeiger
}
@end
```

Memory Management

- Container besitzen ihre enthaltenen Objekte!
  - bevor der Container verschwindet ruft es an allen enthaltenen Objekten `release`!
  - viele SDK-Methoden liefern `autorelease`'d Zeiger:

```
[NSString stringWithFormat:
            @"Meaning of %@ is %d", @"life", 42];
[NSDictionary  dictionaryWithObjectsAndKeys:
                ankush, @"TA",
                janestudent, @"Student", nil];
[NSArray arrayWithContentsOfFile:(NSString *)path];
```

- die meisten Objekte sind `autorelease`'d !

Memory Management

- an `NSString`s wird generell nicht `retain` gerufen - stattdessen werden sie per `copy...` kopiert
- `copy` liefert immer einen unveränderlichen String (auch wenn die Quelle mutable war)
- `copy` ist sehr effizient für NSStrings
- die Kopie besitzt man natürlich

Memory Management

- `release` rufen, sobald wie möglich - wenn man das Objekt nicht mehr braucht:
  - auch für den Leser des Quelltextes eine wichtige Information: der Besitzer wird das Objekt fortan nicht mehr benutzen.
- Was passiert, wenn der letzte Besitzer `release` ruft?
  - eine (`NSObject`-)Methode `dealloc` wird (implizit) gerufen.
  - danach wird der Speicherplatz des Objektes an den Heap als frei zurückgegeben
  - danach darf an das Objekt keine Nachricht mehr gesendet werden!

Memory Management

- `dealloc` sollte also in allen Klassen speziell implementiert aber NIE explizit gerufen werden
- in `dealloc` ruft man `release` für alle (Objekt-) Instanzvariablen (Properties)
  - nicht setter rufen: `self.prop = nil;` (you‘re off the cliff) sondern `[prop release];`
- Einzige Ausnahme: die LETZTE Aktion in eine `dealloc`-Implementation ist immer

```
[super dealloc];
```

Memory Management

- Besitz von Properties
  - Wer besitzt die Objekte die von gettern kommen und in setter eingehen?
  - wichtig zu verstehen!!!
  - getter liefern zumeist den Zeiger auf die Instanzvariable selbst.
  - für den Aufrufer: ähnlich wie `autoreleased`: solange das Objekt lebt, ist die Instanzvariable gültig.
  - will der Aufrufer den Zeiger auch darüber hinaus behalten, muss er `retain` rufen (sonst nicht)
  - wenn der getter die Property erst berechnet (oder als Kopie der Instanzvariablen) bereitstellt, sollte (im getter) explizit `autorelease` gerufen werden

Memory Management

- Beispiel:

```
display.text = [display.text stringByAppendingString:digit];
```

  - kein retain auf das Resultat des getters display.text: wir wollen es ja nicht besitzen (display besitzt es), sondern nur verwenden (um eine neuen autoreleased String zu erzeugen)
  - den wollen wir auch nicht besitzen, sondern an den setter weiterreichen (der über den Besitz entscheiden kann)

Memory Management

- Die bislang synthetisierten setter für Properties kümmern sich NICHT um den Besitz.
  - Dies muss bei der Vereinbarung der Property angezeigt werden:

```
@property (retain) NSArray *myArrayProperty;
@property (copy) NSString *someNameProperty;
@property (assign) id delegate;
```

  - Bei `assign` muss sicher sein, dass das übergebene Objekt das rufende Objekt besitzt, dann lebt es offenbar länger als letzteres.
  - häufiges Szenario: ein Controller besitzt seinen View, also kann ein View eine (assigned) Property haben, die den Controller (als sog. Delegate) referenziert

Memory Management

- die per @sythesize erzeugten Setter sind dann:

```
@property (retain) NSString *name;
...
-(void)setName:(NSString *)aString {
  if (name != aString) {
    [name release];   // alten Besitz aufgeben
    name = [aString retain]; // neuen übernehmen
  }
}
```

Memory Management

- die per @sythesize erzeugten Setter sind dann:

```
@property (copy) NSString *name;
...
-(void)setName:(NSString *)aString {
  if (name != aString) {
    [name release]; // alten Besitz aufgeben
    name = [aString copy];  // neuen übernehmen
  }
}
```

Memory Management

- die per `@sythesize` erzeugten Setter sind dann:

```
@property (assign) NSString *name;
...
-(void)setName:(NSString *)aString {
    name = aString; // kein Besitz
}
```