Protokolle

- abstrakte Schnittstellen die von anderen Klassen implementiert werden können:
  - syntaktisch wie @interface, aber ohne Impl.:

```
@protocol Foo
// implementors must implement this
- (void)doSomething;
@optional
// implementors do not need to implement this
- (int)getSomething;
@required
// must implement
- (NSArray *)getManySomethings:(int)howMany;
@end
```

Protokolle

- im SDK gibt es viele Protokolle z.T. mit überwiegend/ausschließlich optionalen Methoden
- in einem eigenen Headerfile (zumeist) oder im Header einer Klasse, die das Protokoll implementieren will
- Beispiel: Protokoll `UIScrollViewDelegate` im Header `UIScrollViewDelegate.h` - alles optional

Protokolle

- Klassen, die das Protokoll implementieren wollen, müssen dies in ihrer Deklaration anzeigen (wie Javas `implements`):

```
@interface MyClass : NSObject <Foo>
...
@end
```

- Eine Klasse kann auch ein gleichbenanntes Protokoll implementieren:

```
@protocol NSObject  ... @end
@interface NSObject <NSObject> {
   Class isa; }  ... @end
```

Protokolle

- Der Compiler prüft statisch, ob alle `required` Methoden implementiert werden.
- Mehrere Protokolle in `< >` mit Komma getrennt

  häufig bei Controllern
- Variablen vom Typ `id` können auf Protokolle qualifiziert werden: mehr statische Prüfung
- aber keine Auswirkungen auf Laufzeit!

```
id <Foo> obj = [[MyClass alloc] init];
// compiler will love this!
id <Foo> obj = [NSArray array];
// compiler will not like this one bit!
- (void)giveMeFooObject:(id <Foo>) objectImplementingFoo;
```

  eine NSArray-Ableitung könnte Foo implementieren.

Protokolle

- Häufigste Benutzung in iOS:
  - delegates and data sources

```
@property (assign) id <UISomeObjectDelegate> delegate;
```

  - assign in der Annahme, dass der delegate (Controller) länger lebt, als das Objekt mit der property (View).

```
@protocol UIScrollViewDelegate
@optional
-(void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
-(void)scrollViewDidEndDragging:(UIScrollView *)scrollView
                  willDecelerate:(BOOL)decelerate
// many others ...
@property (assign) id <UIScrollViewDelegate> delegate;
@end
```

Protokolle

```
@interface UIScrollView : UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end

@interface MyViewController : UIViewController
                              <UIScrollViewDelegate>
...
@end

MyViewController *myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate = myVC;
```

Views

- Was sind eigentlich Views?
  - Views sind Ableitungen von UIView
  - sind mit einer rechteckigen Fläche auf dem Display verknüpft: zeichnen (sich) in diese und behandeln Ereignisse aus dieser: Swipes, Taps, Pinches, ...

- Views sind hierarchisch verknüpft:
  - ein 'root'-View kann viele Views enthalten
    `(UIView *)superview`
  - kann viele (incl. 0) `(NSArray *)subviews` enthalten

Views

- Reihenfolge im Array wichtig:
  - die Views werden mit absteigendem Index übereinander gezeichnet!
  - Index 0 - toplevel view
  - darunterliegende können 'durchscheinen' in Abh. von der Transparenz der oberen

Views

- Hierarchie oft mit dem IB konstruiert, aber auch programmatisch möglich:

```
-(void)addSubview:(UIView *)view;
// Nachricht an den superview!
```

iOS Reference:
view
The view to be added. This view is retained by the receiver. After being added, this view appears on top of any other subviews

- Beeinflussung der Reihenfolge möglich aber eher unüblich:

```
-(void)insertSubview:(UIView *)aView atIndex:(int)index;
// Nachricht an den superview !
-(void)insertSubview:(UIView *)aView belowSubview:(UIView *)otherView;
// Nachricht an den superview !
-(void)insertSubview:(UIView *)aView aboveSubview:(UIView *)otherView;
// Nachricht an den superview !
-(void)removeFromSuperview;
// Nachricht an den subview !
```

Views

- View können einander überlappen
- Views sind per default undurchsichtig
- Views können verborgen werden, reagieren dann auch nicht auf events: jeder `UIView` hat eine Property:

```
@property(nonatomic, getter=isHidden) BOOL hidden;
// default is NO. doesn't check superviews
```

- `nonatomic`: nicht thread-safe
- Standard für Properties ist `atomic`: thread-safe (mehr Aufwand durch locking)
- `getter=` so heißt der synthetisierte getter (statt `hidden`)
- es gibt auch `setter=`

Views

```
myView.hidden = YES; // weg isser
myView.hidden = NO;  // da isser wieder
```

- Superviews besitzen ihre Subviews (automatisch)
- nach Einfügen in Superview:

  ```
  [subview release];
  ```

- nach `removeFromSuperview` ist der Subview weg: kein `retain` mehr möglich
  - also vorher `retain`, wenn man den Subview weiter verwenden will

Views

- `IBOutlet`'s sollten `retained` sein (im Controller), müssen also auch (im Controller) `released` werden
- wann und wie:
  - wann:

    offensichtlich im `dealloc` des Controllers

    aber auch andwerswo: die Speicherverwaltung in iOS darf nicht sichtbare Views löschen, wenn Speicherknappheit herrscht weil der Controller den View jederzeit aus dem xib-File (automatisch) wiederherstellt wenn der View wieder sichtbar wird … selten - aber möglich, also zu berücksichtigen

Views

in der (UIViewController-)Methode

`-(void)viewDidUnload`

es gibt dual dazu auch

`-(void)viewDidLoad`

wird gerufen, wenn der View komplett geladen ist: hier kann man den inititalen Zustand eines Views über die Möglichkeiten des IB hinaus einstellen

- wie:

  indem die IBOutlets `@property (retain)` sind

  typisch als private Properties (s. Folie 47)
  (sonst kann jeder drauf zugreifen)

Views

MyVC.h

```
@interface MyVC : UIViewController {
    IBOutlet UILabel *outlet;  // IB hook !
}
@end
```

MyVC.m

```
@interface MyVC()
@property (retain) IBOutlet UILabel *outlet;
@end
@implementation MyVC
@synthesize outlet;
- (void)viewDidUnload {
    self.outlet = nil; // setter ok
}
- (void)dealloc {
    [outlet release]; // don‘t use setter
    [super dealloc];
}
@end
```

CG Core Graphic

- `#import <UIGraphics.h>`

  (importiert `<CoreGraphics.h>`)

- CGFloat
  - reelle Zahlen, Koordinaten in Points (nicht Pixels)

- CGPoint

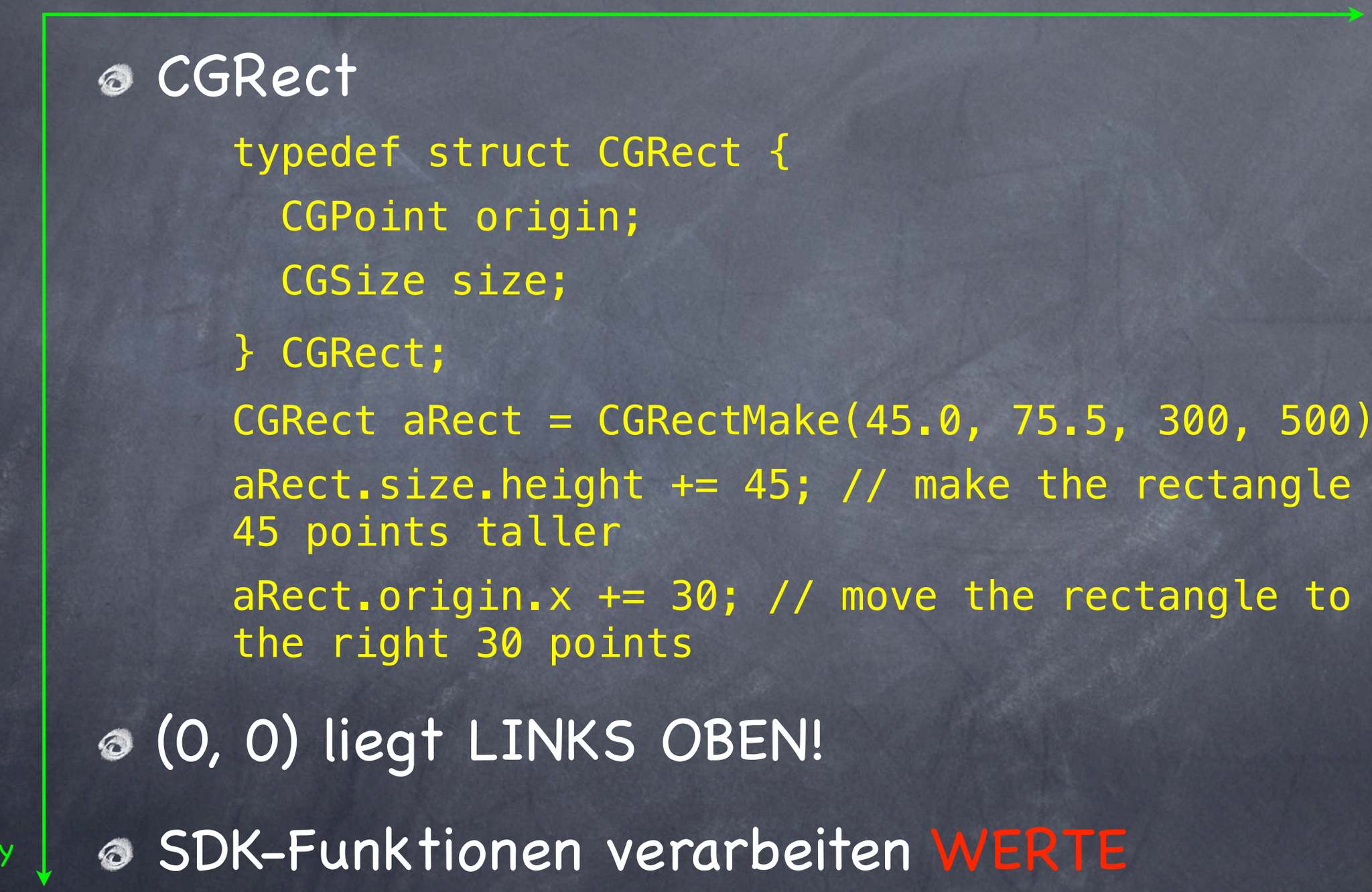
```
typedef struct CGPoint {
  CGFloat x;
  CGFloat y;
} CGPoint;
CGPoint p = CGPointMake(34.5, 22.0);
p.x += 20; // move right by 20 points
```

CG Core Graphic

- CGSize

```
typedef struct CGSize {
  CGFloat width;
  CGFloat height;
} CGSize;
CGSize s = CGSizeMake(100.0, 200.0);
s.height += 50; // make the size 50 points taller
```

Views



- CGRect

```
typedef struct CGRect {
  CGPoint origin;
  CGSize size;
} CGRect;
```

```
CGRect aRect = CGRectMake(45.0, 75.5, 300, 500);
aRect.size.height += 45; // make the rectangle
45 points taller
aRect.origin.x += 30; // move the rectangle to
the right 30 points
```

- (0, 0) liegt LINKS OBEN!
- SDK-Funktionen verarbeiten WERTE

CG Core Graphic

- Points<->Pixel Umrechnung möglich durch

  (UIView-) `@property CGFloat contentScaleFactor;`

  ```
  // returns pixels per point on the screen this view is on
  // may vary on different devices
  ```

- weitere (UIView-)Properties:

  ```
  @property CGRect bounds;
  // your view's internal drawing space's origin
  // and size
  ```

  - `bounds` wird von der Implementation des Views benutzt, ist unabhängig davon, wie und wo dieser View in seinem Superview dargestellt wird

CG Core Graphic

```
@property CGPoint center;
// the center of your view in your superview's
// coordinate space

@property CGRect frame; // a rectangle in your
    // superview's coordinate space which entirely
    // contains your view's bounds.size
```

- `frame` und `center` sind immer bezogen auf den Superview!

CG Core Graphic

View B's `bounds = ((0,0),(200,250))`

View B's `frame = ((140,65),(320,320))`

View B's `center = (300,225)`

View B's Mitte im eigenen Koo.System:

```
(bounds.size.width/2+bounds.origin.x,
 bounds.size.height/2+bounds.origin.y)
```

also hier: (100,125)

CG Core Graphic

- Viele Views werden mit dem IB (zusammen-)gebaut.
- Auch CustomViews können erst mal mit dem IB einbezogen werden, im Inspector wird dann die Klasse angepasst.
- Views können aber auch programmatisch angelegt werden:

```
CGRect buttonRect = CGRectMake(20, 20, 120, 37);
UIButton *button = [[UIButton alloc]
                    initWithFrame:buttonRect];
button.titleLabel.text = @"Do it!" ;
[window addSubview:button]; // window s.u.
[button release];
// okay because button is in view hierarchy now
```

CG Core Graphic

- window - a (toplevel-) View (leer)
- Wann baut man CustomViews?
  - wenn man in diese selbst zeichnen will oder spezielle Events impl. möchte (die standardmäßig nicht angeboten werden)
- Zeichnen:
  - man implementiert in einer UIView-Ableitung:

```
- (void)drawRect:(CGRect)aRect;
```

- kann den ganzen View zeichnen oder (effizienter) sich auf aRect beschränken

CG Core Graphic

- drawRect wird NIEMALS selbst gerufen
- stattdessen teilt man dem iOS mit, dass redraw nötig ist mit

```
- (void)setNeedsDisplay;
// oder
- (void)setNeedsDisplayInRect:(CGRect)
aRect;
```

- Wie zeichnet man?
  - mit den Funktionen aus C-Lib (not OO, Framework) Core Graphics

CG Core Graphic

- GraphicContext-basiert
  - iOS stelle bei jedem drawRect-Ruf einen solchen (neuen! - nicht merken) Context bereit:

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

- Beispiel:

```
CGContextBeginPath(context);
CGContextMoveToPoint(context, 75, 10);
CGContextAddLineToPoint(context, 10, 150);
CGContextAddLineToPoint(context, 160, 150);
CGContextClosePath(context); // not strictly required
```

CG Core Graphic

```
[[UIColor greenColor] setFill];
[[UIColor redColor] setStroke];  // leider nicht einheitlich
CGContextSetLineWidth(context, 1.0); // line width in points
CGContextSetFillPattern(context,
                        (CGPatternRef)pattern, (CGFloat[])components);
CGContextDrawPath(context, kCGPathFillStroke);
```

CG Core Graphic

- Kontexte kann man stapeln:

```
- (void)drawGreenCircle:(CGContextRef)ctxt {
    UIGraphicsPushContext(ctxt);
    [[UIColor greenColor] setFill];
    // draw my circle
    UIGraphicsPopContext();
}

- (void)drawRect:(CGRect)aRect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    [[UIColor redColor] setFill];
    // do some stuff
    [self drawGreenCircle:context];
    // do more stuff and expect fill color to be red
}
```

CG Core Graphic

- Farben (sind OO)

```
UIColor *red = [UIColor redColor];
// class method, returns autoreleased instance

UIColor *custom = [[UIColor alloc] initWithRed:
(CGFloat)red // 0.0 to 1.0
          blue:(CGFloat)blue
         green:(CGFloat)green
         alpha:(CGFloat)alpha); // 0.0 to 1.0
[red setFill]; // fill color set in current
//graphics context (stroke color not set)

[custom set]; // sets both stroke and fill color
// to custom (would override [red setFill])
```

CG Core Graphic

- alpha beeinflusst die Durchsichtigkeit
  - mit der UIView- @property CGFloat alpha
    kann man den ganzen View beeinflussen (fade in/out)
  - dazu MUSS aber eine andere Property
    @property BOOL opaque
    auf NO gesetzt werden! (sonst undef. Ergebnisse)