



**Macoun'10**

# Die Wahrheit über Blocks

Amin Negm-Awad

[co coa:ding]

# Motivation

- Apple-Dokumentation zu “kleinsichtig”.
- Funktionale Programmierung in Objective-C?
- Beweisen, dass Objective-C-ler den längeren haben.

# Ablauf

- Kleine Einführung
- Blocks als anonyme Funktionen
- Blocks als Closures
- Blocks als Instanzobjekte

# Einleitung

# Geschichte

- Haskel Brooks Curry, kombinatorische Logik, Lambda-Kalkül, Konrad Zuse, Plankalkül
- Bestandteil von Objective-C 2.0 / OS X 10.6.
- Verbreiteter Einsatz in Cocoa.

# Begriffe

- Es gibt keine Blocks
- Block-Literal: Stück Code (einfachste Betrachtung)
- Block-Variable: Nimmt Block-Literal auf
- Vergleich: 5 ist Literal, int a nimmt 5 auf.

# Block-Literale

- Stück Code, unveränderlich
- Haben keinen Namen
- Können explizit typisierte Parameter nehmen
- Können implizit typisierte Returnwerte haben
- Werden ausgeführt



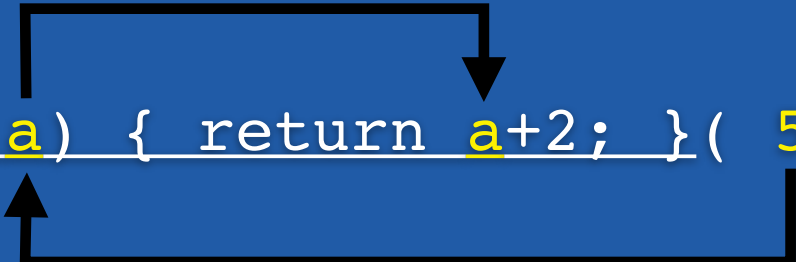
# Block-Literal

```
^(Parameterliste) {  
    Anweisungsliste  
}
```

Definition und Ausführung:

```
^(int a) {      // Parameter a hat int  
    return a+2; // Retval hat int, weil a+2 int hat.  
}
```

*^(int a) { return a+2; }( 5 ); // = 7*



# Block-Variable

- Nimmt einen literalen Block auf
- Kennt Parameter, explizite Typisierung
- Kennt Retval, explizite Typisierung.
- typedef möglich

# Block-Variable

```
Rettype (^Bezeichner)( Parameterliste );
```

```
int (^block)( int );
```

```
typedef Rettype (^Bezeichner)( Parameterliste );
```

```
typedef int(^Block)( int );
```

# Zuweisung

- Blockvar = Blockliteral
- Blockvar = Blockvar
- Kann in der Definition der Block-Variablen geschehen.
- Sehr strenge Typprüfung
- By-Reference

# Zuweisung

```
// Definition ...  
// ... und Zuweisung von Literal  
int (^block)( int );  
block = ^(int a) { return a+2; }
```

```
// Definition ...  
// ... und Zuweisung von Var  
int (^block2)( int );  
block2 = block;
```

```
// Zuweisung in Definition  
int (^block)( int ) = ^(int a) { return a+2; }
```

# Typstrenge

```
// Keine implizite Umwandlung von Skalaren
float (^blockVar)( int ) = ^( int a ) {
    return a;
}; // Fehler: Retval, int ist nicht float!

float (^blockVar)( int ) = ^( int a ) {
    return a + 1.0;
}; // Fehler: Retval, double ist nicht float!

float (^blockVar)( int ) = ^( int a ) {
    return a + 1.0f;
}; // Funktioniert
```

# Typstrenge

```
// Auch nicht bei Klassen
id (^blockVar)( void ) = ^{
    return @"Hallo";
}; // Fehler: Retval, NSString ist nicht id!

NSObject (^blockVar)( void ) = ^{
    return @"Hallo";
}; // Fehler: Retval, NSString ist nicht NSObject!

id (^blockVar)( void ) = ^{
    return (id)@"Hallo";
}; // Funktioniert
```

# Extent

- Block-Literal lebt nur in Umgebung { ... }
- Darf nicht „exportiert“ werden.
- Lösung: Kopie
- Compilerhilfe nur manchmal
- Empfehlung: Blöcke stets kopieren



# Extent Fehler

```
typedef int(^Block)( int );  
Block block;  
  
if( ... ) {  
    block = ^(int a) { return a+2; }  
} // Fehler, aber kein error!
```

```
Block function( void )  
{  
    block = ^(int a) { return a+2; }  
    return block;  
} // Fehler, aber kein error!
```

# Extent Fehler

```
Block function( void )  
{  
    int b = 5;  
    return ^(int a) { return a+b; }  
} // error: returning block that lives on the local stack
```

```
Block function( void )  
{  
    int b = 5;  
    block = ^(int a) { return a+b; }  
    return block;  
} // Fehler, aber kein error!
```

# Extent Lösung

```
Block function( void )
{
    int b = 5;
    return = [[^(int a) { return a+b; } copy] autorelease]
} // Richtig
```

```
Block function( void )
{
    int b = 5;
    block = ^(int a) { return a+b; }
    return [[block copy] autorelease];
} // Richtig
```

# Blocks als anonyme Funktionen

# Ausführung

- Block-Literale enthalten Code
- Haben Parameter und Rückgabewerte
- Werden mit () ausgeführt
- Haben keinen Namen
- Also: Wie Funktionen ohne Namen?

# Ausführung

```
int (^block)( int );  
block = ^(int a){ return a+2; };  
block( 5 )
```

```
int (^block)( int ) = ^(int a){ return a+2; };  
block( 5 )
```

```
^(int a){ return a+2; }( 5 )  
←────────────────→ ←→
```

# Dynamik

- Block-Variable ist dynamisch
- Block-Literal ist statisch
- Ähnlich Funktionszeiger

# Dynamik

```
typedef int(^Block)( int, int );  
Block block;  
Block addBlock = ^(int a, int b ){ return a+b; };  
Block mulBlock = ^(int a, int b ){ return a*b; };  
  
block = addBlock;  
block( 2, 4 ) // 6  
  
block = mulBlock;  
block( 2, 4 ) // 8
```



# Blocks als Closures

# Hintergrund

- Funktionale Programmierung
- Lambda-Kalkül, Currying

# Klassische Begriffe

- Abstraktion

$\lambda x. x+2;$

- Applikation

()

Block

$\wedge(\text{int } x) \{ \text{return } x+2; \}$

Ausführung

()

# Funktion + Umgebung

- Blöcke kennen ihre Umgebung
- Wie in C-Blöcken externe Variablen bekannt
- Problemstellung/Beispiel: Currying

# Currying

- Ausgangsidee: Jede Funktion erhält nur einen Parameter
- „3+4“ ist daher nicht eine Operation mit zwei Parametern, sondern zwei Operationen mit einem Parameter:  
 $\text{add}(3, 4) \Rightarrow \text{add}(3)(4)$
- Idee:  $\text{add}(3)$  liefert eine Funktion, die etwas zu 3 addiert. Diese Funktion wird dann mit 4 aufgerufen und liefert 7.

# Curry mit C-Funktion

```
typedef NSInteger(*Function)( NSInteger );
```

```
NSInteger addTo3( NSInteger b )  
{  
    return 3+b;  
}
```

```
Function functionAdd( NSInteger a )  
{  
    return addTo3;  
}
```

```
add( 3 )( 4 ) // liefert 7
```

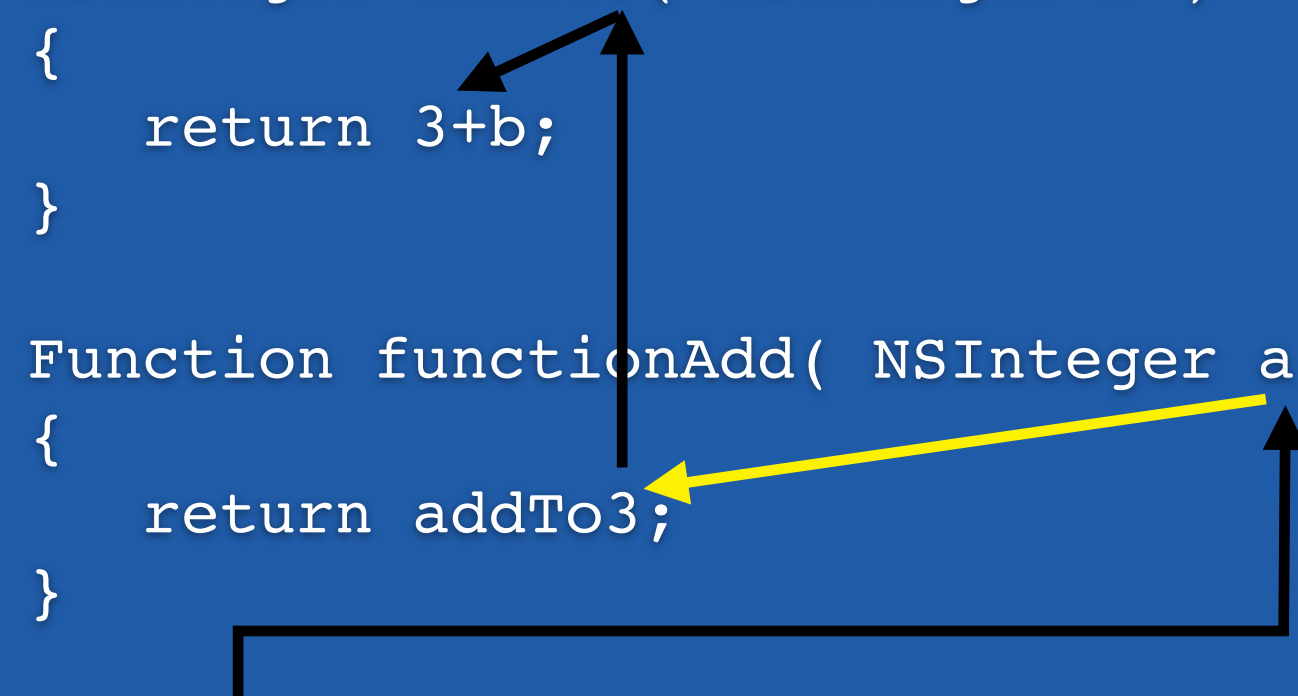
# Problem

```
typedef NSInteger(*Function)( NSInteger );
```

```
NSInteger addTo3( NSInteger b )  
{  
    return 3+b;  
}
```

```
Function functionAdd( NSInteger a )  
{  
    return addTo3;  
}
```

```
add( 3 )( 4 ) // liefert 7
```



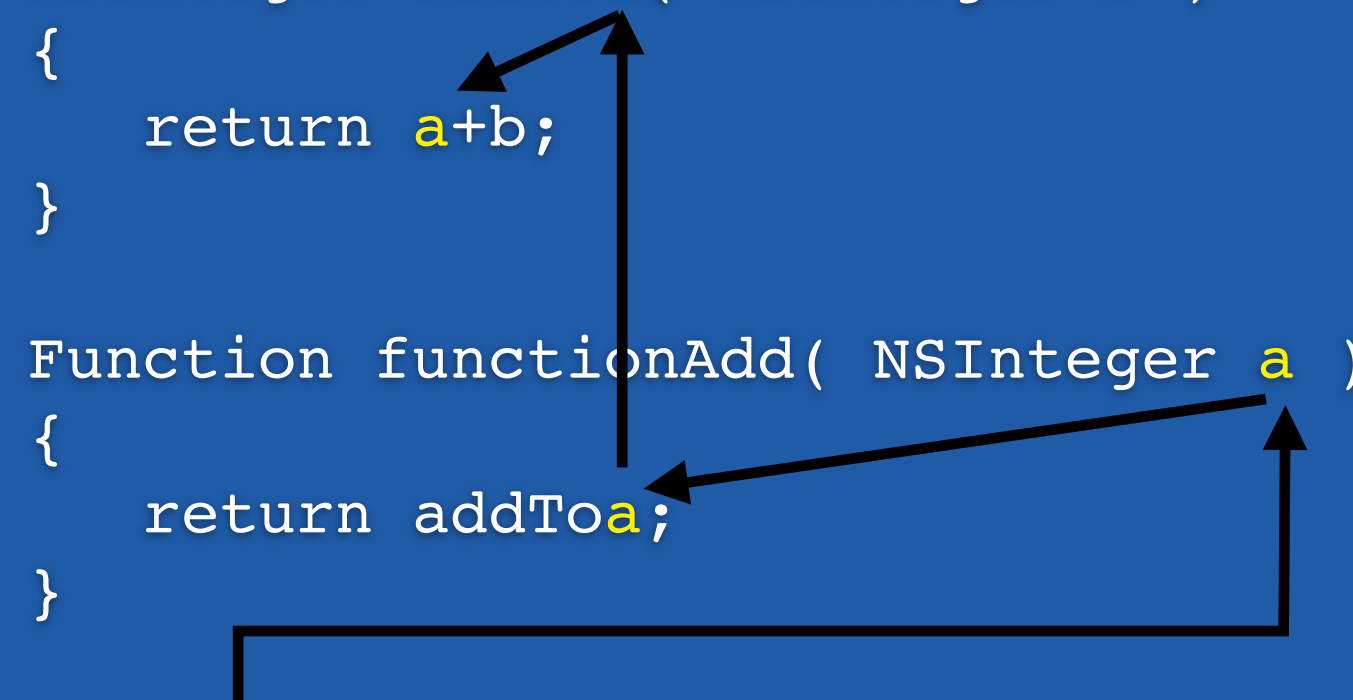
# Lösung?

```
typedef NSInteger(*Function)( NSInteger );
```

```
NSInteger addToa( NSInteger b )  
{  
    return a+b;  
}
```

```
Function functionAdd( NSInteger a )  
{  
    return addToa;  
}
```

```
add( 3 )( 4 )
```





# Curry mit Block-Literal

```
typedef NSInteger(^Block)( NSInteger );
```

```
Block addTo3 = ^(NSInteger b)
{
    return 3+b;
};
```

```
Block blockAdd( NSInteger a )
{
    return addTo3;
}
```

# Problem:

```
typedef NSInteger(^Block)( NSInteger );
```

```
Block addToa = ^(NSInteger b)
{
    return a+b;
};
```

```
Block blockAdd( NSInteger a )
{
    return [[addToa copy] autorelease];
}
```

# Lösung: Closure

- Ein Closure kennt die Umgebung, seiner Erzeugung.
- Variablen werden eingesetzt.

# Lösung: Closure

```
typedef NSInteger(*Block)( NSInteger );

Block blockAdd( NSInteger a )
{
    return [[^(NSInteger b){ return a + b; } copy] autorelease];
}

Block addBlock = blockAdd( 4 ); // ^(NSInteger b){ return 4 + b; }
addBlock( 5 )                  // ^(NSInteger b){ return 4 + 5; }

addBlock( 4 )( 5 )
```

The diagram illustrates the closure concept with three arrows:

- An arrow from the variable `a` in the `blockAdd` function signature to the `a` in the block's return statement `return a + b;`.
- An arrow from the `a` in the block's return statement to the value `4` in the call `addBlock = blockAdd( 4 );`.
- An arrow from the `a` in the block's return statement to the value `4` in the final call `addBlock( 4 )( 5 )`.

# Umgebung

- Block ist Kombination aus Code und Umgebung.
- Auch globale Umgebung
- Statics?
- Bezogene Werte werden By-Value gemerkt.
- Objective-C: Keine Stack-Objekte
- retain statt copy (Bug: retain erst bei Heap-Shift)

# Stack-Block

```
main() {  
  int a = 4;  
  Block addBlock  
  = ^(NSInteger b){ return a + b; };  
  int a = 17;  
  addBlock( 5 ); // return a + b: 9  
}
```

Stackframe

a = 4



# Stack-Block

```
main() {  
    int a = 4;  
    ▶ Block addBlock  
      = ^(NSInteger b){ return a + b; };  
    a = 17;  
    addBlock( 5 ); // return a + b: 9  
}
```

Stackframe  
a = 4



# Stack-Block

```
main() {  
    int a = 4;  
    Block addBlock  
    = ^(NSInteger b){ return a + b; };  
    ▶ a = 17;  
    addBlock( 5 ); // return a + b: 9  
}
```

Stackframe

a = 17

a = 4






# Stack-Block

```
main() {  
  int a = 4;  
  Block addBlock  
  = ^(NSInteger b){ return a + b; };  
  a = 17;  
  ▶ addBlock( 5 ); // return a + b: 9
```

Stackframe

a = 17  
a = 4



# Globals

- Gehören zur Umgebung
- Müssen noch nicht definiert sein.
- Nehmen nicht am Closure teil
- Auch Blöcke können global sein.

# Globals

```
extern int global;

void (^globalBlock)() = ^{
    NSLog( @"%d", global ); // Deklaration reicht
};

int global = 5;
```

# Globals + Closure

```
int var = 11;

void (^block)() = ^{
    NSLog( @"%d", var );
};

...
{
    block();
    var = 98;
    block(); // 98!
}
```

# Globals + Closure

```
{  
    int var = 11;  
  
    void (^block)() = ^{  
        NSLog( @"%d", var );  
    };  
  
    block();  
    var = 98;  
    block(); // 11  
}
```

# Statics

- Statische Variablen im Block werden geteilt.
- Keine Kopie pro Instanz

# Statics

```
void (^block)() = ^{
    static int stat = 0;
    NSLog( @"static %d", stat++ );
};

block();           // 0, 0 -> 1
block();           // 1, 1 -> 2
void (^copyOfBlock)() = [[block copy] autorelease];
block();           // 2, 2 -> 3
copyOfBlock();     // 3, 3 -> 4
```

# Instanzen

- ID wird kopiert
- Nicht: Objekt wird kopiert
- retain? (Bug!)



# Instanzen

```
id object = [NSMutableSet setWithObjects:@"Amin", nil ];
Block block = ^( void) { return [object count]; };
NSLog( @"%d", block() ); // 1
object = [NSMutableSet setWithObjects:@"Amin", @"Negm", nil];
NSLog( @"%d", block() ); // Anderes Objekt, daher immer noch 1
```

```
id object = [NSMutableSet setWithObjects:@"Amin", nil ];
Block block = ^( void) { return [object count]; };
NSLog( @"%d", block() ); // 1
[object addObject:@"Negm-Awad"];
NSLog( @"%d", block() ); // 2, nur ID gesichert
```

# Problem

- Ursprung kann verschwinden.
- Speicher erforderlich.
- Stack-Heap-Shift

# Stack-Heap-Shift

```
typedef NSInteger (^Block)( void );
```

```
Block function() {  
    NSInteger a = 5;  
    Block block  
    = ^( void ) { return a; };  
    return block;  
}
```

...

```
Block block = function();  
// block = ^( void ) { return a; };  
NSLog( @"%d", block() );
```

Stackframe

a = 5



# Stack-Heap-Shift

```
typedef NSInteger (^Block)( void );

Block function() {
    NSInteger a = 5;
    Block block
    = ^( void ) { return a; };
    return block;
}

...
Block block = function();
// block = ^( void ) { return a; };
NSLog( @"%d", block() );
```

Stackframe  
a = 5



# Stack-Heap-Shift

```
typedef NSInteger (^Block)( void );

Block function() {
    NSInteger a = 5;
    Block block
    = ^( void ) { return a; };
    return [[block copy] autorelease];
}

...
Block block = function();
// block = ^( void ) { return a; };
NSLog( @"%d", block() );
```

Stackframe  
a = 5

Heap  
a = 5



# Stack-Heap-Shift

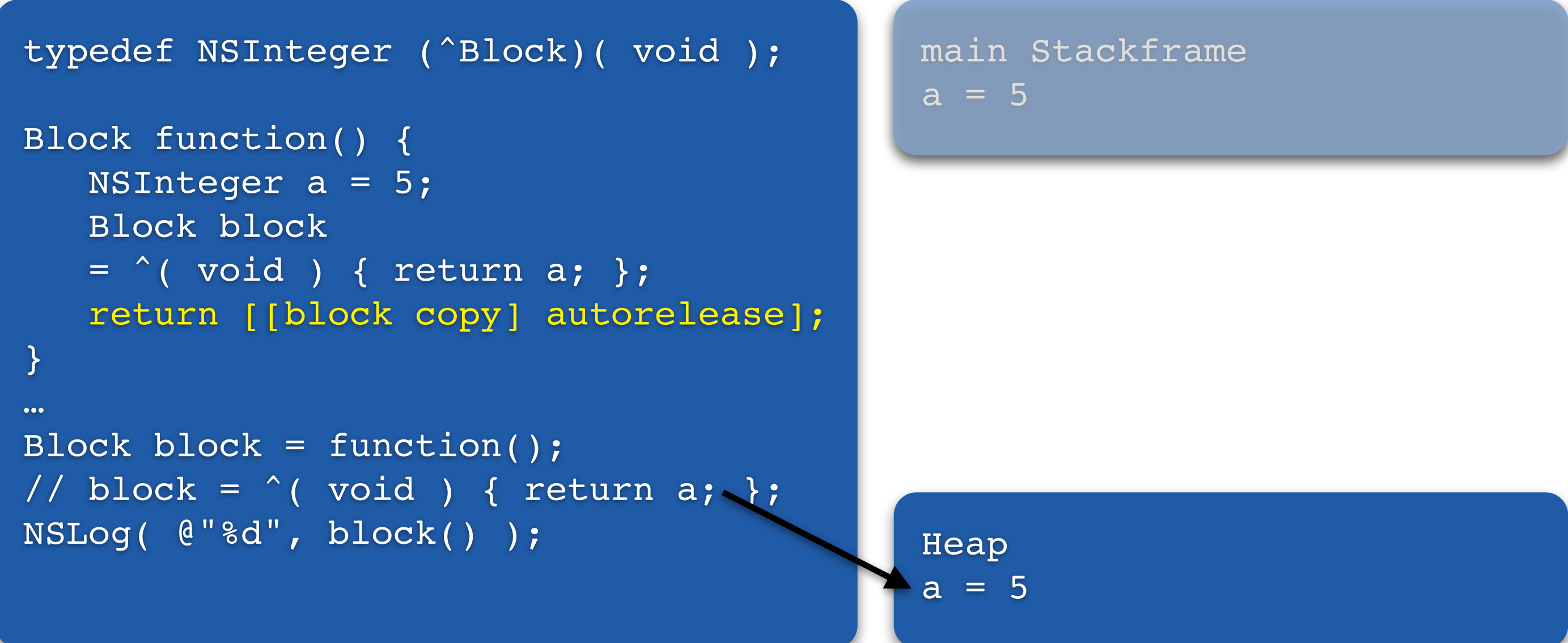
```
typedef NSInteger (^Block)( void );

Block function() {
    NSInteger a = 5;
    Block block
    = ^( void ) { return a; };
    return [[block copy] autorelease];
}

...
Block block = function();
// block = ^( void ) { return a; };
NSLog( @"%d", block() );
```

main Stackframe  
a = 5

Heap  
a = 5



# \_\_\_block

- \_\_\_block-Referenzen werden nicht kopiert.
- Dangling-Pointer möglich!

# Blocks als Instanzobjekte



# Instanzobjekte

- Haben eine Klasse
- Empfangen Nachrichten

# Klasse

- Globaler Kontext:  
`__NSGlobalBlock__:__NSGlobalBlock:NSBlock:NSObject`
- Stack-Kontext:  
`__NSStackBlock__:__NSStackBlock:NSBlock:NSObject`
- Heap-Kontext:  
`__NSMallocBlock__:__NSMallocBlock:NSBlock:NSObject`

# Klasse

```
do {
    NSLog( @"%s", class_getName( class ) );
    unsigned int count;
    Method* methodList = class_copyMethodList( class, &count );
    while( count-- > 0 ) {
        NSLog( @"%s", method_getName( methodList[ count ] ) );
    }
    class = class_getSuperclass( class );
    if( class_getSuperclass( class ) == Nil ) { // ⚡ NSObject
        break;
    }
} while( class != Nil );
```

# Möglichkeiten

# Lazyness

- Problem:

Parameter werden evaluiert, vllt aber nicht benötigt

- Lösung:

Block wird evaluiert erst, wenn er ausgeführt wirds

# Funktion

```
NSInteger multiply( NSInteger a, NSInteger b ) {  
    if( a == 0 ) { return 0;  
    } else { return a * b; }  
}  
  
NSInteger function() {  
    sleep( 10 );  
    return 4;;  
}  
...  
NSLog( @"%d", multiply( 0, function() ) ); // 10 Sekunden für nichts
```

# Block

```
NSInteger multiply( NSInteger a, Block b ) {  
    if( a == 0 ) { return 0;  
    } else { return a * b(); }  
}
```

```
Block block = ^(void) {  
    sleep( 10 );  
    return (NSInteger)4;  
};
```

...

```
NSLog( @"%d", multiply( 0, block ) );
```

# Callback

- Problem:

Callback benötigen Hilfsinformatioun (Info-Dictionary)

- Lösung:

Blocks speichern benötigte Information der Umgebung



# Callback

```
void callBack( id from, id info )
{
    NSInteger state1 = info.state1;
    NSString* text = info.text;
    ...
}
...
{
    ...
    info = ...;
    info.state1 = state1;
    info.text = text;
    doSomethingThatCallsBack( callBack, info );
}
```

# Callback

```
{  
  ...  
  void (^block)( id from ) = ^{  
    // text und state sind vorhanden.  
  }  
  doSomethingWithBlock( callBack, block );  
}
```

# Threading

- Problem:  
Konflikte auf gemeinsame Ressourcen.
- Lösung:  
Bei rein funktionaler Programmierung keine Seiteneffekte

# Smart-Pointer

- Problem:

Bei Verlassen eines Scopes sollen Instanzen entfernt werden

- Lösung?:

Ein Block schickt ein retain und löscht sich beim Verlassen des Scopes (release), aber Compiler-Bug

# Abschluss

# Fragen

**Vielen Dank**



**Macoun'10**