

## Closures in Java und Lambdas in C#

Michael Wiedeking

Karlsruher Entwicklertag 2010  
24. Juni 2010

MATHEMA Software GmbH  
([www.mathema.de](http://www.mathema.de))

Gängige Vorstellung:

- Wiederverwendbares Stück Code
- Syntaktischer Zucker

```
static void sort(int[] a) {  
    ...  
    if (a[i] < a [j]) {  
        swap(a, i, j);  
    }  
    ...  
}
```

```

interface IntComparator {
    int compare(int a, int b);
}

static void sort(int[] a, IntComparator c) {
    ...
    if (c.compare(a[i], a [j]) < 0) {
        swap(a, i, j);
    }
    ...
}

```

```
class AscendingIntComparator implements IntComparator {  
    public int compare(int a, int b) {  
        if (a < b) {  
            return -1;  
        } else if (a == b) {  
            return 0;  
        } else {  
            return +1;  
        }  
    }  
}
```

```
int[] numbers = {3, 6, 1, 7, 4, 8, 9, 2, 0, 5};  
sort(numbers, new AscendingIntComparator());
```

```
IntComparator ASCEND = new AscendingIntComparator();  
sort(numbers, ASCEND);
```

```
IntComparator DESCEND = new IntComparator() {  
    public int compare(int a, int b) {  
        return Integer.compare(a, b);  
    }  
}  
sort(numbers, DESCEND);
```

```
delegate int IntComparator(int a, int b);
```

```
static void sort(int[] a, IntComparator compare) {  
    ...  
    if (compare(a[i], a [j]) < 0) {  
        swap(a, i, j);  
    }  
    ...  
}
```

```
sort(a, delegate (int a, int b) { return Integer.compare(a, b); });
```

`sort(a, delegate (int a, int b) { return Integer.compare(a, b); });`

`sort(a, (int a, int b) => Integer.compare(a, b));`

`sort(a, (a, b) => Integer.compare(a, b));`

`sort(a, (int a, int b) => { return Integer.compare(a, b); });`

*(parameter-list) => expression-or-statement-block*

### Expression

$x \Rightarrow x * x$

$(x, y) \Rightarrow x * y$

### Statement

$x \Rightarrow \{ \mathbf{return} \ x * x; \}$

$(x, y) \Rightarrow \{ \mathbf{return} \ x * y; \}$

```
delegate int Transformer(int i);
```

```
Transformer square = x => x * x;
```

```
WriteLine(square(3)); // 9
```

```
Func<int, int> sqr = square; // ???
```

```
WriteLine(sqr(3)); // 9
```

**delegate**  $T_{\text{Result}}$  **Func**<**out**  $T_{\text{Result}}$ >

**delegate**  $T_{\text{Result}}$  **Func**<**in**  $T_1$ , **out**  $T_{\text{Result}}$ >

**delegate**  $T_{\text{Result}}$  **Func**<**in**  $T_1$ , **in**  $T_2$ , **out**  $T_{\text{Result}}$ >

**delegate**  $T_{\text{Result}}$  **Func**<**in**  $T_1$ , ..., **in**  $T_{16}$ , **out**  $T_{\text{Result}}$ >

**delegate** void **Action**

**delegate** void **Action**<**in**  $T_1$ >

**delegate** void **Action**<**in**  $T_1$ , **in**  $T_2$ >

**delegate** void **Action**<**in**  $T_1$ , ...,  $T_{16}$ >

- Schreibvarianten

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = (\text{int } x) \Rightarrow x * x;$

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = (\text{int } x) \Rightarrow (x * x);$

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = ((\text{int } x) \Rightarrow (x * x));$

- Typinferenz

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = x \Rightarrow x * x;$

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = (x) \Rightarrow x * x;$

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = (x) \Rightarrow (x * x);$

$\text{Func}\langle \text{int}, \text{int} \rangle \text{ square} = ((x) \Rightarrow (x * x));$

$\#(\textit{parameter-list}) \Rightarrow \textit{expression-or-statement-block}$

### Expression

$\#(x)(x * x)$

$\#(x, y)(x * y)$

### Statement

$\#(x) \{ \mathbf{return} \ x * x; \}$

$\#(x, y) \{ \mathbf{return} \ x * y; \}$

```
#int(int) sqr = #(x)(x * x);
```

```
System.out.println(sqr(3)); // 9
```

```
interface Transformer {  
    public int transform(int i);  
}
```

```
Transformer square = #(x)(x * x);
```

```
WriteLine(square.transform(3)); // 9
```

```
#int(int) sqr = square; // ???
```

```
#int(int) sqr = #(x)(x * x);
```

```
WriteLine(sqr.(3)); // 9
```

```
static int F(int i) {  
    int a;  
    int b;  
    ...  
    {  
        int tmp = a;  
        a = b;  
        b = tmp;  
    }  
}
```

```
Action[] actions = new Action[3];
```

```
for (int i = 0; i < 3; i++) {  
    actions[i] = () => Console.WriteLine(i);  
}
```

```
foreach (Action a in actions) {  
    a();  
}
```

```

int i;
Action[] actions = new Action[3];

i = 0;
actions[0] = () => Console.Write(i);

i = 1;
actions[1] = () => Console.Write(i);

i = 2;
actions[2] = () => Console.Write(i);

i = 3;

foreach (Action a in actions) {
    a();
}

```

```
for (int  $i = 0$ ;  $i < 3$ ;  $i++$ ) {  
    int  $n = i$ ;  
     $actions[i] = () \Rightarrow Console.Write(n)$ ;  
}
```

```
foreach (Action  $a$  in  $actions$ ) {  
     $a()$ ;  
}
```

```
static void Main( ) {  
    int factor = 2;  
    Func<int, int> mul = n => n * factor;  
    WriteLine(mul(3)); // 6  
}
```

```
static void Main( ) {  
    int factor = 2;  
    Func<int, int> mul = n => n * factor;  
    WriteLine(mul(3)); // 6  
}
```

- *factor* ist sogenannte *captured* Variable
- Lambdas mit *captured* Variablen sind eine *Closure*

```
static void Main( ) {  
    int factor = 2;  
    Func<int, int> mul = n => n * factor;  
    factor = 10;  
    WriteLine(mul(3)); // 30  
}
```

```
int i = 0;  
Func<int> counter = () => i++;  
WriteLine(counter()); // 0  
WriteLine(counter()); // 1  
WriteLine(i); // 2
```

```
static Func<int> CreateCounter() {  
    int i = 0;  
    return () => i++;  
}
```

```
static void Main() {  
    Func<int> counter = CreateCounter();  
    WriteLine(counter( )); // 0  
    WriteLine(counter( )); // 1  
}
```

```
static Func<int> CreateCounter() {
    return ( ) => {
        int i = 0;
        return i++;
    }
}
```

```
static void Main() {
    Func<int> counter = CreateCounter();
    WriteLine(counter( )); // 0
    WriteLine(counter( )); // 0
}
```

- Lokale Variablen und formale Methoden- oder Exception-Handler-Parameter, die in einem Lambda-Ausdruck benutzt aber nicht deklariert werden, müssen *effektiv final* sein
- Lokale Variablen und formale Methoden- oder Exception-Handler-Parameter sind *effektiv final*, wenn sie innerhalb des Lambda-Ausdrucks weder das Ziel einer Initialisierung oder einer Zuweisung sind, außer wenn diese definitiv uninitialisiert sind

```

class C {
  void m(int x) {
    int y;
    y = 1;
    ... #( ) (x + y) ... // Legal; x und y sind effektiv final

    y = 2;
    ... #( ) (x + y) ... // Illegal; y ist nicht effektiv final

    int z = 1;
    ... #( ) (z + 1) ... // Legal; z ist effektiv final
  }
}

```

- *this* in einem Lambda-Ausdruck ist vergleichbar zu *this* in einer anonymen Klasse
- Keine Einschränkung bei qualifizierten Namen
- Erlaubt unqualifizierte Referenzen auf *unveränderliche* Variablen aller Art
- Erlaubt unqualifizierte Referenzen auf äußere Variablen (aller Art), wenn sie mit *@Shared* markiert sind
- Erlaubt unqualifizierte Referenzen auf äußere Methoden, wenn sie mit *@Shared* markiert sind

```
class CountingSorter {  
    @Shared  
    int count = 0;  
  
    void sort(List<String> data) {  
        Collections.sort(  
            data,  
            #(String a, String b) {  
                CountingSorter.this.count++;  
                return a.length() - b.length();  
            }  
        );  
    }  
}
```

- Problemlösung für *Checked Exceptions* nötig
- `#int()(throws IOException)`
- `#void(int)(throws NoSuchFieldExc...|NoSuchMethodExc...)`

- C#

`Collection<T> Collection.filter(Func<T, boolean> predicate)`

`Collection<Integer> collection = ...;`

`Collection<Integer> c = collection.filter(x => even(x));`

- Java

`Collection<T> Collection.filter(#boolean(T) predicate)`

`Collection<Integer> collection = ...;`

`Collection<Integer> c = collection.filter(#(Integer x)(even(x)));`

- „Äußere“ Iteration
  - Verkettete Liste
    - Triviale Implementierung
  - Baum
    - Grundsätzlich trivial (via Rekursion)
    - Leider passt die Iterator-Schnittstelle nicht dazu
- „Innere“ Iteration
  - Struktur ist fürs Iterieren irrelevant
  - Rekursion ist damit unproblematisch

boolean	exists(#boolean(A) <i>predicate</i> )
Iterable<A>	filter(#boolean(A) <i>predicate</i> )
int	indexOf(#boolean(A) <i>predicate</i> )
Iterable<B>	flatMap(#Iterable<B>(A) <i>f</i> )
boolean	forall(#boolean(A) <i>predicate</i> )
void	foreach(#void(A) <i>f</i> )
Iterable<B>	map(#B(A) <i>f</i> )
Pair<(Iterable<A>, Iterable<A>>	partition(#boolean(A) <i>p</i> )
B	reduceLeft(#B(B, A) <i>op</i> )
B	reduceRight(#B(A, B) <i>op</i> )
B	foldLeft(B <i>zero</i> , #B(B, A) <i>op</i> )
B	foldRight(B <i>zero</i> , #B(A, B) <i>op</i> )

$\#int(int, int) \text{ multiply} = \#(int\ a, int\ b)(a * b);$

$\#int(int) \text{ double} = \#(int\ a)(\text{multiply}(2, a));$

Noch Fragen?

Vielen Dank!

[michael.wiedeking@mathema.de](mailto:michael.wiedeking@mathema.de)  
[www.mathema.de](http://www.mathema.de)