Closures in Java

Michael Wiedeking

MATHEMA Software GmbH
(www.mathema.de)

Java Forum Stuttgart 2010
1. Juli 2010

Closure? Closure!

Gängige Vorstellung:

- Wiederverwendbares Stück Code
- Syntaktischer Zucker

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

Motivation (1/4)

```
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);
    }
    …
}
```

Motivation (1/4)

```
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;
        }
    }

}
```

Motivation (1/4)

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

sort(numbers, new AscendingIntComparator());
```

Motivation (1/4)

```
class AscendingIntComparator implements IntComparator {
    public static final ASCEND = new AscendingIntComparator();
    public int compare(int a, int b) {
        …
    }
}

sort(numbers, AscendingIntComparator.ASCEND);
```

Motivation (1/4)

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

Allgemeine Syntax

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

Expression

```
#(int x)(x * x)
#(int x, int y)(x * y)
```

Statement

```
#(int x){ return x * x; }
#(int x, int y){ return x * y; }
```

Aufrufsyntax

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

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

Typ-Inferenz

#()(5)	#int()
#() {}	#void()
#(int x)()	#void(int)
#(int x)(x*x)	#int(int)
#() { if (…) return; else return; }	#void()
#() { if (…) return 1; else return 2; }	#int()
#() { if (…) return new Integer(1); else return 2; }	#Integer()
#() { if (…) return "1"; else return 2; }	#Object()
#() { if (false) return 5; throw new Error(); }	#int()
#() { throw new Error(); }	?

Süße Sache: Syntactic Sugar

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

```
Transformer square = #(int x)(x * x);
System.out.println(square.transform(3)); // 9
```

```
setActionListener(#( ){ button.setText(”Done”); });
```

```
new Thread(#( ) calculateComplicatedThing()).start();
```

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

Nicht lustig! – C# (1/3)

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


for (int i = 0; i < 3; i++) {
    actions[i] = #( ){ System.out.println(i) };
}


for (Action a : actions) {
    a( );
}
```

Nicht lustig! – C# (2/3)

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

i = 0;
actions[0] = #( ){ System.out.println(i); };

i = 1;
actions[1] = #( ){ System.out.println(i); };

i = 2;
actions[2] = #( ){ System.out.println(i); };

i = 3;

for (Action a : actions) {
    a( );
}
```

Nicht lustig! – C# (3/3)

```
for (int i = 0; i < 3; i++) {
    int n = i;
    actions[i] = #( ){ System.out.println(n); };
}

for (Action a : actions) {
    a( );
}
```

Effektiv finale Variablen (1/2)

- 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

Effektiv finale Variablen (2/2)

```
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

    }
}
```

Umgang mit nicht-finalen Variablen (1/2)

- *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

Umgang mit nicht-finalen Variablen (2/2)

```
class CountingSorter {

    @Shared
    private int count = 0;

    public void sort(List<String> data) {
        Collections.sort(
            data,
            #(String a, String b) {
                CountingSorter.this.count++;
                return a.length() – b.length();
            }
        );
    }

}
```

Selbstreferenzen

#int(int) *factorial* = #(int *i*)(*i* == 0 ? 1 : *i* * factorial.(*i* – 1));

ist nicht möglich, weil *factorial* bei der Wiederverwendung noch nicht vollständig initialisiert

#int(int) *factorial* = #(int *i*)(*i* == 0 ? 1 : *i* * ***this***.(*i* – 1));

this innerhalb eines Lambda-Ausdrucks möglich, weil *this* dort einen Funktionstyp hat

Closures und Checked Exceptions

- #int()(**throws** IOException)
- #void(int)(**throws** SecurityException | IOException)

```
try {
    …
} catch (final SecurityException | IOException e) {
    …
    throw e;
} catch (NoSuchFieldException | NoSuchMethodException e) {
    …
}
```

Hin zur funktionalen Ansätzen

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

Collection<Integer> *collection* = …;

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

Beispiel Iterator

- „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

Typische Methoden für Collection<A>

```
boolean          exists(#boolean(A) predicate)
Iterable<A>      filter(#boolean(A) predicate)
int              findIndexOf(#boolean(A) predicate)
Iterable<B>      flatMap(#Iterable<B>(A) f)

boolean          forall(#boolean(A) predicate)
void             foreach(#void(A) f)

Iterable<B>      map(#B(A) f)

Pair<(Iterable<A>, Iterable<A>>     partition(#boolean(A) p)

B                reduceLeft(#B(B, A) op)
B                reduceRight(#B(A, B) op)

B                foldLeft(B zero, #B(B, A) op)
B                foldRight(B zero, #B(A, B) op)
```

„Currying“ (1/2)

#int(int, int) *multiplyer* = #(int *a*, int *b*)(*a* * *b*);

#int(int) *doubler* = #(int *a*)(*multiplyer*.(2, *a*));

„Currying“ (2/2)

```
#int(int, int) multiplyer = #(int a, int b)(a * b);

#int(int) doubler = #(int a)(multiplyer.(2, a));

##int(int)(int) curriedMultiplyer = ##int(int)(int a) {
    return #int(int b)(a * b);
};

#int(int) doubler = curriedMultiplyer(2);
```

Konstanten

```
assert #( )(42).( )  == 42;

#int( ) fortyTwo = #( )(42);
assert fortyTwo.() == 42;

#int(int) doubler = #(int x)(x + x);
assert doubler.(fortyTwo.( )) == 84;
```

Komposition (1/3)

```
public class Plotter {
    public void draw(#double(double) function) { … }
    …
}
```

```
#double(double) f = …;
Plotter plotter = new Plotter(…);
plotter.draw(f);
```

Komposition (2/3)

```
#double(double) f;

f = #double(double x)(Math.sin(x));
plotter.draw(f);

f = #double(double x)(Math.sin(Math.cos(x)));
plotter.draw(f);
```

Komposition (2/3)

```
#double(double) f = …;
#double(double) g = …;

#double(double) compose(#double(double) f, #double(double) g) {
    return #double(double x)(f.(g.(x)));
}
```

Business-Komposition (1/2)

#void(Person) *action* = #void(Person *p*){ *p*.verify(); };

Person *p* = …;

action.(*p*);

Business-Komposition (2/2)

```
#void(Person) compose(#void(Person) f, #void(Person) g) {
    return #void(Person p) { f.(p); g.(p); };
}

#void(Person) a1 = #void(Person p){ p.verify(); };
#void(Person) a2 = #void(Person p){ p.sendConfirmation(); };

#void(Person) action = compose(a1, a2);

if (p.isCritical()) {
    action = compose(action, #void(Person p){ p.log(); });
}

action.(p);
```

Bedingte Ausführung

```
#void(Person) createAction(
    #boolean(Person) condition,
    #void(Person) action
) {

    return #void(Person p) {
        if (condition.(p)) {
            action.(p);
        }
    };

}
```

Maßgeschneiderte Closures

```
#( ) createAction(A a, B b, C c) {
    return #( ) {
        … a … b … c …
    };
}
```

```
public void maybe(boolean condition, #void() action) {
    if (condition) {
        action.( );
    }
}

Special x = …;
Info info = …;

maybe(x != null, #( ) {
    if (x.isSpecial( )) {
        x.append(info);
    }
}
```

Nicht lustig!

```
public void loop(#boolean( ) condition, #void( ) action) {
    while (condition) {
        action.( );
    }
}

Special x = …;

loop(#( ) x.isValid(), #( ) {
    if (x.isSpecial( )) {
        break;
    }
    x = x.nextInLine( );
}
```

Das Gelernte überprüfen

- Was beschreibt der folgende Typ?

  ##int()(#int(), #int())

Das Gelernte überprüfen

- Was beschreibt der folgende Typ?

  # #int() **(** #int(), #int() **)**

- Das folgende Beispiel hat diesen Typ:

  #int() compose(#int() *a*, #int() *b*)

Das Gelernte überprüfen

- Was beschreibt der folgende Typ?

  ###int()(#int(), #int())(##int()(#int(), #int()), ##int()(#int(), #int()))

Das Gelernte überprüfen

- Was beschreibt der folgende Typ?

  ###int()(#int(), #int())(##int()(#int(), #int()), ##int()(#int(), #int()))

- Das folgende Beispiel hat diesen Typ:

  # #int() **(** #int(), #int() **)** hyperCompose(
  # #int() **(** #int(), #int() **)** $composer_1$,
  # #int() **(** #int(), #int() **)** $composer_2$
)

Noch Fragen?

Vielen Dank!

michael.wiedeking@mathema.de
www.mathema.de