kleiner kommunikation
max@kleiner.com

//Exklusiv für Entwickler 2012

# 1 Die Funktion von Closures

Ein Closure ist ein funktionales Objekt, das sich Werte unabhängig seines Geltungsbereichs (Scope) merkt und zurückgibt. Andere meinen auch, es sei schlicht ein anonymer Code Block mit Widerverwendung. Wenn Sie jemals eine Funktion gebaut haben, die eine andere Funktion zurückgibt, haben Sie bereits mit der Closure Idee Kontakt geschlossen.

Dieser sprachübergreifende Bericht soll Licht in die Welt der Closures bringen und so ganz nebenbei einige Tatsachen zur funktionalen Programmierung liefern (Back to Function).

## 1.1 Closure öffne dich

Ich betrachte ein Closure als Vermittler zwischen Funktionen und Objekten, agil wie eine Funktion und zudem als Referenz kompakt wie ein Objekt. Closure-nahe Implementierungen sind einfach zu realisieren, seien es mit inneren Klassen, Prozedurtypen, Funktionszeiger oder (anonymen) Delegaten, jedoch reicht das in der Regel noch nicht aus. Ja warum nicht, weil bei allen der Scope eingeschränkt ist.

Ja wie übersetzt man eigentlich Closure, etwa Abschliesser oder Auflöser, klingt genau so fremd wie wenn wir den Spring Controller als Frühlingsüberwacher bezeichnen.

Closure kommt von schließen und soll das Abschließen eines Ausdruckes in einer Methode verdeutlichen. Das klingt abstrakt, wird sich gegen Ende des Artikels aber konkretisieren.

Der Ursprung des Closure-Konzepts liegt in den bereits erwähnten funktionalen Programmiersprachen wie Lisp, Haskell oder Ruby, Lisp schon im Jahr 1959. Das Closure-Konzept selbst hat man bereits in den 60er-Jahren kreiert und erstmals in der Sprache Scheme – ein Dialekt der Sprache Lisp – umgesetzt (notabene noch ohne Objekte). Damit sich Closures auch in prozeduralen Programmier-Sprachen wie C oder Pascal verwenden lassen, muss es möglich sein, Funktionen als Parameter übergeben zu können. Dies ist bei Pascal oder in C mit einem Prozedurtyp als Schnittstelle möglich, genügt aber noch nicht:

```
Type
 TMath_func = Procedure(var x: float);  //Procedure Type

procedure fct1(var x: float);  //Implement
  begin
    x:= Sin(x);
  end;
var
  fct1x:= @fct1 //Referenz
  fct_table(0.1, 0.7, 0.4, fct1x); //Funktion als Parameter
  fct_table(0.1, 0.7, 0.4, @fct1); //alternativ Direkt ohne Referenz
```

Mit diesem Konstrukt lässt sich eine Aufrufkette bilden, in der Resultate einer Berechnung einen weiteren Funktionsaufruf auslösen, da man den Funktionszeiger munter weiter reichen kann und eben auch zurückgerufen wird.

Man sieht, die Sprachen C und Pascal sind mehr oder weniger streng typisiert und operieren mit echten Zeigern, im Weiteren betrachten wir dynamische Sprachen mit Referenztypen statt Pointern.

Konzept und Mächtigkeit sind in Closure-fähigen Sprachen in etwa ausgeglichen, man wird ohnehin mit in den Sprachen unterschiedlicher Syntax und Typen konfrontiert, beispielsweise mit Action- und

Formular-, Jason-XML-Konfigurations-, JavaScript- und auch HTML- oder JSP-Konstrukte, die sich dann mit Closures eleganter bearbeiten lassen. Es stellt sich manchmal die Frage, ob es angesichts der ohnehin hohen Komplexität anzuraten ist, Groovy, Ruby, Scala oder Dart in einem Projekt zusätzlich einzuführen, nur weil man mit Closures arbeiten will, die in Java offensichtlich noch nicht implementiert sind (http://www.javac.info/).

So, genug der Vorgeschichte, schauen wir nun mal in eine Closure rein, die auf Python setzt:

```
def generate_power_func(n):
    print "id(n): %X" % id(n)
    def nth_power(x): //Closure
       return x**n
    print "id(nth_power): %X" % id(nth_power)
  return nth_power
```

Die innere Funktion `nth_power(x)` ist ein Closure, da sie Zugriff auf den Parameter n hat, der von der äusseren Funktion `generate_power_func(n)` gekapselt wird (enclosing scope genannt). Als Rückgabe dient die innere Funktion, also der Closure!

Das Erstaunlich ist, auch wenn der Programmfluss die Funktion verlässt, ist der Zugriff auf die lokale Variable n immer noch möglich. Hier spricht man von einer externen lokalen Variablen, die Sprache Lua hat mal den Begriff `upvalue` geprägt. Die Funktion `nth_power(x)` schliesst also n mit ein.

Nun rufen wir den Closure auf und weisen den Funktionszeiger einer Variablen (Referenz) zu:

```
>>> raised_to_4 = generate_power_func(4)
```

Wie erwartet, erzeugt der Aufruf ein Funktions-Objekt und speichert es in `raised_to_4`. Im idDebugger lässt sich das als Zuweisungsoperator erkennen

```
 id(raised_to_4) == 0x00C47561 == id(nth_power)).
```

Nun löschen wir die Funktion aus dem globalen Namensraum mit `>>> del generate_power_func` und rufen den Closure einfach auf (it's time for magic):

`>>> raised_to_4(2)` und erhalten als Schlussresultat `16`. Das Erstaunliche ist der angeblich lokale Parameter `n=4` welcher der Closure bekannt ist und somit als vollständige Funktion dient. Das `nth_power` Funktions Objekt kennt also die internen Details der umschliessenden Funktion im Scope und ist deshalb ein Closure.

Das Fantastische ist nun, dass ein lokales Wieder verwenden von Code Blöcken (Snippets) oder ein Refactoring mit dem Extrahieren einer Methode nun viel eleganter und effizienter möglich ist und durch den globalen Scope garantiert zu weniger Duplicated Code führt, frei nach dem Motto „don't repeat yourself" (aka dry).

Als weiteres Beispiel einer Anwendung will ich in einer bestehenden Methode eine Ausgabe als Liste einmalig definieren, ohne Closure benötige ich eine neue Methode im zugehörigen Objekt, als Closure genügt das Einschliessen des Blockes mit einer zugehörigen Referenz für den späteren Aufruf:

```
mylst:= TStringList.create;
  with TSession.Create(NIL) do try
    SessionName:= 'Mars3'
    getAliasNames(mylst);
    for i:= 1 to mylst.count-1 do  //Define as Closure: var Outlist{}
      write(mylst[i]+' ');         //
  finally
    Free;
    mylst.Free;
  end;
```

Das Beispiel soll nur zeigen, dass Sprachen ohne Closures viel Substanz vergeben oder Redundant sind, indem häufig Code Blöcke dupliziert werden.

Closures sind auch sehr nützlich für Call-Back Methoden oder die in C# und Delphi bekannten Delegates und Method Pointer. Wobei die Anonymen Delegates und Closure zwei unterschiedliche

Konzepte darstellen. Denn die Effizienz als Funktion bei den Closures ist eben besser als ständig als neues Objekt zu definieren. So ein Delegate benötigt neben der Struktur, dem Typ und dem Konstruktor auch ein Main, die Closure lässt sich direkt einer Referenz zuweisen oder eben als Parameter übergeben. Hier will ich das Design Pattern Visitor erwähnen, dass von der Idee her eine Closure am besten verdeutlicht.
Ein einfaches Beispiel, nun in PHP, soll nochmals mit einer konkreten Referenz `t3` den gewöhnungsbedürftigen Scope und das zugehörige Konstrukt verdeutlichen:

```
def times(n):
def _f(x):
  return x * n
return _f

t3 = times(3)
 print t3 #
 print t3(7) # 21
```

Auch hier wieder deutlich, die äussere Funktion `times(n)` hat als Rückgabewert die innere Funktion `_f()`, hier als Lambda Expression[1]. Mit der Referenz `t3` wird dem Closure der Parameter `n=3` bekannt gemacht, den er beim Aufruf von `t3(7)` verwerten kann!

Festzuhalten ist, dass die Deklaration zur Entwurfszeit nicht mit dem Verhalten zur Laufzeit übereinstimmt, hier werkelt die Compiler-Magic indem die lokalen Variablen ja als externe Referenzen auf einem Stack gespeichert sind.

Mit anderen Worten, wenn `times(n)` ausgeführt wird, liegt eine Referenz des Closure Object `_f()` auf dem Stack; später lässt sich das Closure Object jederzeit benutzen.

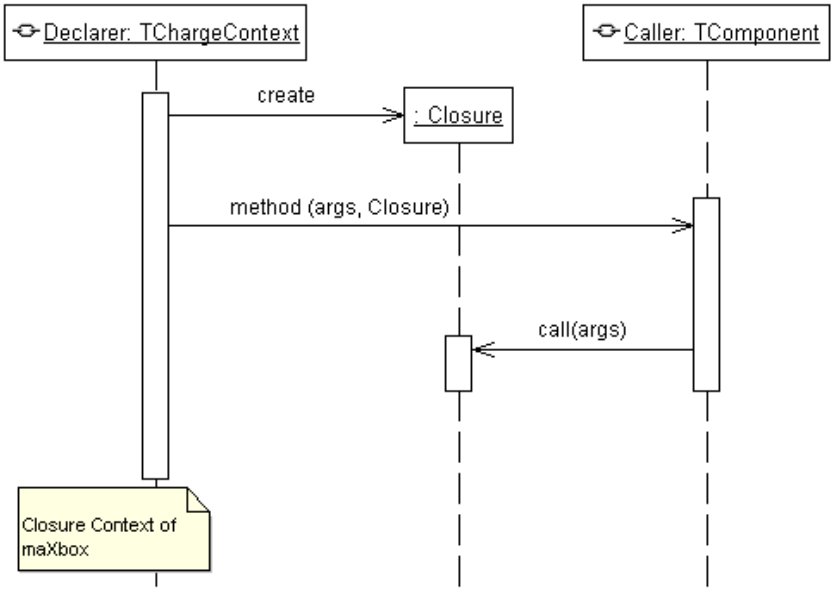
Zitat aus Wiki: “The closure object can later be invoked, which results in execution of the body, yielding the value of the expression (if one was present) to the invoker. A closure expression can have parameters, which act as variables whose scope is the body”.

Fassen wir nochmals das Grundprinzip im nachfolgenden Code zusammen:

```
>>> def outer(x):
...   def inner(y):
...     return x+y
...   return inner
...
>>> customInner=outer(2)
>>> customInner(3)
result:→ 5
```

Eine äussere Funktion (body) erweitert den inneren Closure mit seinen Variablen und Parametern zur Laufzeit. Nachfolgendes Sequenzdiagram visualisiert den Closure als moderne Call-Back Methode und zeigt auch das Prinzip der so genannten Inversion of Control (IoC). (als Belohnung für die, die bis hierher gefolgt sind ;)):

[1] Eine Lambda Expression ermöglicht das Kürzen einer Funktionsdeklaration




*// Seq_Closure2.png*

Abb. 1: Im Diagram ersichtlich ist die Übergabe wie der Rückruf

## 1.2 Closures als Threads und Class Helpers

Closures implementieren, zumindest bei Groovy, von Hause aus das Interface `Runnable` und lassen sich somit auch asynchron ausführen. Im folgenden Beispiel starte ich zwei Closures als parallel laufende Threads:

```
groovy> Thread.start { ('A'..'Z').each {sleep 100; println it} }
groovy> Thread.start { (1..26).each {sleep 100; println it} }
A
1
B
2
```

Jeder einzelne Thread schreibt eine Liste von Zeichen (Buchstaben) bzw. Zahlen, wobei er bei jedem Durchlauf eine kurze Pause einlegt, damit jeweils der andere Thread zum Zug kommt – und es für uns verständlich wird, ansonsten die Reihenfolge unbestimmt ist.

Übrigens einige der Konzepte und Muster der Closures lässt auch mit der maXbox (es muss nicht immer eine neue JVM-Sprache sein) nachvollziehen, die man unter: http://sourceforge.net/projects/maxbox/ kostenlos beziehen kann. Dazu steht ein Script bereit: `271_closures_study.txt`.

Abschließend noch ein konkretes Funktionsmuster mit einer Klasse, die mit einer Closure als Logger zusammenarbeitet, in Java oder Delphi hätte man eine Inner oder Nested Class genommen. Delphi 2010 kennt auch anonyme Methoden /Funktionen, die einer Closure sehr nahe kommen. Warum heißen die anonym, ganz einfach weil man sie nur via Referenz und nicht über einen Namen aufrufen kann. Wenn man die anonymen Methoden noch mit dem Scope erweitert, hat man Closures, die in Delphi 2010 auch realisiert sind.

Wir bauen abschließend eine Klasse, die irgendwas implementiert, dies aber in einer flexiblen Weise protokollieren soll. Die Protokollmethode ist durch ein property definiert, der wir eine Closure zur Ausgabe des Protokolltextes zuweisen.

```
class LoggingWorker {
 Closure log = { step,text ->}
 def somework () {
   log(0, "somework started")
   //...
```

```
    log(1, "Error happens")
    //...
    log(0, "somework finished")
  }
```

Hier hat der Closure gleich zwei Parameter, die nach der Zuweisung im Kontext der Methode `somework` arbeiten:

```
 def worker = new LoggingWorker()
 worker.log = {step,text -> println(['INFO','ERROR'][step]+': '+text) }
 worker.somework();
```

Solche Konstrukte vermeiden auch zunehmende Namenskonflikte, da der Closure ja intern symbolisiert ist. Nach all dem Staunen stellt sich die Frage nach Nachteilen und die gibt es in der Tat mit einem Wort erwähnt: Seiteneffekte! Wer nicht diszipliniert arbeitet, dem spielen die `upvalues` (externe lokale Variable) böse Streiche, dies soll nicht unerwähnt bleiben.
Was auch erstaunt, die komplette Übergabe einer Closure gleich als Definition!:

```
with TMyClass.Create do begin //Delphi 2010
  DoWork(
     procedure(value: string)
     begin
       Writeln(value);
     end);
  Free;
end;
```

Als Gag hab ich noch versucht eine Closure zu visualisieren, so dass der ganze Monitor einfach zur grauen Fläche mutiert, hinter der Fläche sind die symbolischen Objekte, auf der Fläche die Funktion und die Fläche selbst dient als Closure (Die Fläche mit F4 wieder verlassen ;)):

```
//Object Pascal
  with TForm.Create(self) do begin
    BorderStyle:= bsNone;
    WindowState:= wsMaximized;
    Show;
  end;
```

Für Java Entwickler interessant sind die Diskussionen ob mit Closures die Sprache oder die Java VM selbst erweitert werden soll. Hierbei handelt es sich um eine Binärversion des Kompromiss-vorschlags für die Unterstützung von Closures in Java (aka BGGA), siehe Links. Am besten gleich heute ein Non-Disclosure Statement unterschreiben ;-).

Max Kleiner

Literatur:

Michael Bolin, “Closure: The Definitive Guide", O'Reilly Media; Auflage: 1 (20. Oktober 2010).

Links:

http://www.javac.info/

http://sourceforge.net/projects/maxbox/