KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

11/2014

Jahrgang 7

Editorial

Nachwuchs

Während manch einer froh darüber gewesen sein mag, im Alter von fünf Jahren schon den einen oder anderen Buchstaben lesen zu können, hat es jetzt ein (gerade noch) Fünfjähriger geschafft, *Microsoft Certified Professional* zu werden. Jetzt befürchte ich, dass ich meine Meinung über Zertifizierungen noch einmal gründlich überdenken muss.

Ich weiß ja nicht, was Sie von einer Zertifizierung halten, aber ich persönlich bin nicht dafür zu haben. Als Java noch jung war und plötzlich die Möglichkeit einer Zertifizierung bestand, habe ich vorbereitende Schulungen zu dem Thema gehalten und Übungsfragen erstellt, aber nie die Notwendigkeit gesehen, mich selbst zertifizieren zu lassen. Warum auch.

Hatte ich so etwa kein Verständnis für die Fragen, die zu ergründen suchten, ob sich der Beispiel-Code überhaupt übersetzen lässt. Dafür gibt es doch Compiler! Und ich war schon immer damit zufrieden, zu wissen, wenn es sich nicht übersetzen lässt, warum es sich nicht übersetzen lässt. Außerdem hat der Erfolg von Büchern wie *Effective Java* oder *Java Puzzlers* gezeigt, dass es – auch mit einer Zertifizierung – genügend Raum für Unsicherheiten gibt.

Unabhängig davon hat sich schon in der Schule gezeigt, dass man auch dann gute Noten bekommen kann, wenn man von dem Thema eigentlich keine Ahnung hat. Beispielsweise hat ein Freund für einen Aufsatz über den „Schimmelreiter" ein „ausreichend" bekommen, von dem er keine einzige Zeile gelesen hat. So sind mir im Laufe der Zeit auch einige zertifizierte Java-Entwickler über den Weg gelaufen, die etwa das Konzept der Objektorientierung nicht verstanden haben.

Eine Zertifizierung ist also nur ein Indiz dafür, dass sich jemand mehr oder weniger intensiv mit einem Thema auseinandergesetzt hat. Das hat dann wohl auch der inzwischen sechsjährige Ayan Qureshi aus Coventry genügend getan, der mit Erfolg die Zertifizierung für *Supporting Windows 8.1* [1] absolviert hat und damit am 27. September 2014 zum jüngsten Microsoft Certified Professional (MCP) aller Zeiten gekürt worden ist [2].

Das setzt jetzt möglicherweise Studienabgänger ein bisschen unter Druck, die typischerweise erst nach ihrer Ausbildung zu derartigen Prüfungen antreten. Ich konnte leider nichts darüber finden, wie hoch die Erfolgsrate ist. Eine Microsoft-Mitarbeiterin schreibt aber, dass man, um das für eine Zertifizierung notwendige (skalierte) 700-Punkte-Ziel (von 1000 Punkten) zu erreichen, je nach Thema mindestens zwischen 50 % und 85 % der Fragen korrekt beantworten muss [3].

Ich bin mir nun nicht mehr ganz sicher, ob ich jetzt nicht noch weniger von Zertifizierungen halte als vorher – völlig losgelöst davon, was ich von den Eltern halte, die den Jungen zur Prüfung gebracht haben. Vielleicht bin ich ja auch einfach zu intolerant; der Junge hat sicher in einem seiner Magazine (wie Rolf Kaukas *Bussi Bär*) von der Möglichkeit einer Zertifizierung gelesen und sich dann wohl gedacht, dass es gut für seine Karriere wäre, wenn er die Prüfung sobald wie möglich absolviert. Wobei mich schon interessieren würde, wie ich auf jemanden bei einer Bewerbung reagieren würde, der mit einer Zertifizierung glänzt, die er vor 15 bis 20 Jahren abgeschlossen hat.

Der kleine Held hat sich schon mit drei Jahren an den Umgang mit Rechnern gewöhnt. Da kannte ich in meinem Freundeskreis auch einen. Der konnte noch keine Zeile lesen, war aber sehr wohl in der Lage, jedes für ihn geeignete Computerspiel selbstständig über das Windows-Menü zu starten. Ein anderes Kind in dieser analphabetischen Altersgruppe konnte trotz der Leseschwäche quasi jede beliebige CD finden, die es einmal unter Ansage des Titels gezeigt bekommen hatte.

Hätte ich die Gelegenheit, so würde ich den Erstklässler mal fragen, warum denn bei den Netzmasken ausgerechnet der Wert 255 benutzt wird (wenn das denn bei einer Windows-Installation überhaupt noch irgendwo von Bedeutung ist). Aber vielleicht spielt das auch gar keine Rolle. Schließlich muss ja auch ein Arzt nicht verstehen, wie ein EKG funktioniert; er muss die gekrakelte Linie auf dem Papierstreifen nur korrekt interpretieren können.

Ayan hat damit übrigens den bisherigen Rekordhalter Mehroz Yawar geschlagen, der seine Zertifizierung, deutlich älter, „erst" mit sechseinhalb Jahren gemacht hat.

Ihr Michael Wiedeking

Referenzen

[1] Microsoft *MCSA: Windows 8*, https://www.microsoft.com/learning/de-de/mcsa-windows-8-certification.aspx

[2] Brown, Aaron *Byte-size British schoolboy, aged FIVE, becomes world's youngest Microsoft IT specialist*, Express, 13. November 2014, http://www.express.co.uk/news/uk/535056/Youngest-Microsoft-IT-Specialist-British-Boy-Aged-Five

[3] Munson, Liberty *The Truth About Scoring: 700 Does NOT Equal 70%* https://borntolearn.mslearn.net/btl/b/weblog/archive/2009/08/05/the_2d00_truth_2d00_about_2d00_scoring_2d00_700_2d00_does_2d00_not_2d00_equal_2d00_70.aspx

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde.

Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.

Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Frau Natalia Wilhelm via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-16 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Natalia Wilhelm via E-Mail unter natalia.wilhelm@bookware.de oder telefonisch unter 09131/8903-16.

Copyright

Impressum

KaffeeKlatsch Jahrgang 7, Nummer 11, November 2014
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Verwaltungs- und Service-Gesellschaft mbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking
Anzeigen: Natalia Wilhelm
Grafik: Nicole Delong-Buchanan

Inhalt

Artikel

Kolumnen

Typen mit System

Verständliche und flexible Typ-Systeme für DSLs 6
von Andreas Heiduk

Typ-Systeme sind kompliziert und undurchsichtig, nur etwas für Theoretiker. *If-then-else*-Kaskaden tiefer als das Challenger Deep werden zum Frühstück serviert. So sagt man. Aber für eigene domänenspezifische Sprachen (DSLs) kommt man schon mit wenig Code zum Ziel – ohne die Kaskaden.

Yeoman

Frontend Development Workflow 12
von Abderrahmen Dhouibi

Heutzutage sind viele Werkzeuge, Bibliotheken und Frameworks nötig, um ein Web-Projekt zu starten. Die Integration in das eigene Projekt erfordert einen hohen Arbeitsaufwand: Skripte, Download, Einbinden, Zusammenpacken, Auslieferung auf den Server… Dieser Prozess ist komplex und kostet viel Zeit. Es wäre schön, wenn ein neues Web-Projekt in wenigen Minuten erstellt werden könnte und viele Aufgaben und Updates von Ressourcen automatisiert laufen würden.

Datenbänkchen

WebStorage als eine Alternative zu Cookies 15
von Zeeshan Ali

WebStorage ermöglicht Websites bzw. Web-Servern, wie *Cookies*, Daten auf der Benutzerseite zu speichern, aber mit höherer Kapazität und einfacher Nutzung.

Typen mit System

Verständliche und flexible Typ-Systeme für DSLs

von Andreas Heiduk

Typ-Systeme sind kompliziert und undurchsichtig, nur etwas für Theoretiker. *If-then-else*-Kaskaden tiefer als das Challenger Deep werden zum Frühstück serviert. So sagt man. Aber für eigene domänenspezifische Sprachen (DSLs) kommt man schon mit wenig Code zum Ziel – ohne die Kaskaden.

Eine DSL verzichtet bewusst auf die Freiheiten einer *General Purpose Language* (GPL). Daher muss das Typ-System einer DSL ebenfalls nicht mit dem Typ-System einer GPL konkurrieren – insbesondere wenn die DSL „fachlich" motiviert ist und „nur" Daten hin- und hergeschoben werden.

Für das hier beschriebene Typ-System habe ich unter *GitHub* [1] eine kleine DSL inklusive Typ-System bereitgestellt. Damit können Sie die Details besser nachvollziehen und auch selbst experimentieren.

Die Sprache selbst erlaubt es Strukturen (*Entities*) zu beschreiben. Eine *Entity* kann wie üblich Attribute oder Referenzen zu anderen Entities haben. Attribute können ein paar vordefinierte primitive Typen haben: *string*, *int*, *float*, *boolean*, *date*, *time* und *timestamp*. Ein kleines Beispiel für die Strukturen:

```
entity Person {
  string firstName
  string lastName
  date birthday
  Address homeAddress
}

entity Address {
  string street
  string city
  string postalCode
  string country
}
```

Es gibt außerdem noch drei Arten von Statements:

- *declare* definiert eine Variable mit einem Typ.
- *set* setzt einen Ausdruck auf den Wert eines anderen Ausdruckes.
- *eval* wertet einen Ausdruck aus, ohne ihn zuzuweisen.

Und zuletzt gibt es noch verschiedene Ausdrücke mit den Operatoren *and*, *or*, +, –, *, /, ==, !=, <, <=, >, >= und dem *Punkt*-Operator.

Damit kann man das erste Beispiel ergänzen:

```
declare timestamp ts
declare Person p
set ts = p.address.city + 42 – p.lastName < (true * 3.1415)
set p.lastName = ts
eval p
```

Wie man sieht – ohne Typ-System kommt man zwar syntaktisch aber nicht semantisch sehr weit.

Die Sprache ist sehr einfach gehalten, daher gibt es keine wirklichen *Mapping*-Anweisungen oder UI-Beschreibungen. Aber *set* mit seinem Ausdruck dient als Platzhalter für alle Stellen, an denen in größeren DSLs ein Ausdruck stehen kann und ein Typ entweder implizit vorgegeben ist oder sich ableiten lässt. *eval* dagegen steht für Stellen, an denen zwar Berechnungen erlaubt sind, der Typ des Ergebnisses aber keine Rolle spielt. Sowohl *set* als auch *eval* sind also „Ankerpunkte" für Ausdrücke.

Die Sprache ist auch aus Sicht der Typ-System-Theorie sehr einfach:

- keine Ableitungen bzw. Sub-Typen
- keine *Generics*
- keine Inferenz von Variablen-Typen (da der Typ von Variablen explizit definiert wird)
- keine *Lambda*-Ausdrücke

Dennoch verbergen sich schon in diesem Kern ein paar interessante Gemeinheiten:

- *and* und *or* können nur mit booleschen Operanden arbeiten und ergeben wieder einen booleschen Wert.
- Die Vergleichsoperatoren funktionieren nur mit „vergleichbaren" Operanden und liefern einen booleschen Wert.
- *Plus* ist hoffnungslos überladen und ergibt je nach der Kombination der Operanden einen anderen Typ:
  - Wenn ein Operand ein *String* ist, ist das Ergebnis ein *String*.
  - Wenn beide Operanden *int* oder *float* sind, ist das Ergebnis *int* bzw. *float*.

  - Wenn ein Operand *float* und der andere *int* ist, ist das Ergebnis *float*.
  - Wenn die Operanden *date* und *time* sind, ist das Ergebnis *timestamp*.
- Subtraktion, Multiplikation und Division funktionieren nur mit Zahlenwerten (*int*, *float*), die aber auch gemischt werden können. Ergebnis ist wieder ein Zahlenwert.
- Wenn *int* mit *int* verrechnet wird, soll das Ergebnis wieder *int* sein. Gleiches gilt für *float*. Gemischte Rechnungen aber ergeben *float*.

Interessant ist die Addition von *date* und *time* zu *timestamp*: Ohne diese Anforderung lassen sich die anderen Regeln so erklären, dass jeweils einer der beiden Operanden-Typen gewinnt. Da hier aber drei unterschiedliche Typen im Spiel sind, kann es nicht so einfach gelöst werden. Je mehr primitive Datentypen man hat, desto wahrscheinlicher ist dieses Phänomen.

Für Statements gilt nur folgende Regel: Bei *set* muss der Typ des rechten Ausdruckes dem Typ des linken Ausdruckes zuweisbar sein.

Nachdem nun die erwünschten Anforderungen definiert sind, geht es im nächsten Schritt darum, diese in verständlichen und wartbaren Code zu gießen.

Divide et impera

Wartbarer und verständlicher Code schließt aus, dass man die aufgezählten Punkte einfach so herunter programmiert, denn dann landet man schnell bei den *If-then-else*-Kaskaden. Und wehe es steht dann eine Erweiterung um neue Operatoren oder Datentypen an!

Aber eine mögliche Aufteilung wurde schon durch die Formulierung der Punkte oben angedeutet. Das Typ-System hat im Prinzip zwei Aufgaben:

- Berechnung des Ergebnis-Typs zu einem Teilausdruck
- Sicherstellen, dass die Operanden den Anforderungen entsprechen

Diese zwei Aufgaben sollte man aus mehreren Gründen getrennt angehen:

- *Xtext* verarbeitet eine Datei in folgenden Schritten. Parsen, Linking/Scoping der Querbeziehungen, Prüfungen, evtl. noch Code generieren.
- Die Typ-Berechnung selbst wird an mehreren Stellen verwendet:
  - *Scoping*: Der Punkt-Operator muss für den rechten Operanden den Typ des linken kennen, denn nur Namen von Attributen und Referenzen aus diesem sind auf der rechten Seite erlaubt.
  - *Prüfung*: Natürlich wird die Typ-Prüfung selbst intensiv die Typ-Berechnung verwenden.
  - *Code-Generierung*: Die Typen von Ausdrücken haben durchaus Einfluss auf den generierten Code.
  - *Code-Completion im Editor*: Hier kann die Typ-Berechnung die Vorschlagsliste erweitern oder einschränken.

Das heißt, die Typ-Berechnung für einen Ausdruck wird an verschiedenen Stellen mehrfach durchgeführt und sollte daher so effizient wie möglich sein.[1]

Die Typ-Prüfung dagegen findet nur in einer Phase und für jeden Teilausdruck nur einmal statt.

Die Trennung von Berechnung und Prüfung führt zum Prinzip „*Loose typing, strict checking*“ [2]. Das heißt, im Extremfall kann ein Typ in der ersten Phase nicht korrekt ermittelt werden – dann wird ein Platzhalter-Typ wie ANY_TYPE zurückgegeben. Da aber in der zweiten Phase die Prüfungen stattfinden, werden diese Stellen als fehlerhaft markiert und vom Code-Generator nicht mehr bearbeitet.

Das Prinzip beinhaltet auch, dass die Typ-Berechnung eines Teilausdrucks möglichst nicht dessen abstrakten Syntax-Baum (AST) rekursiv bis in die Blattspitzen traversieren sollte.

Loose typing...

Sehen wir uns die Typ-Berechnung ein wenig genauer an: Im Beispiel ist sie in der *Xtend*-Klasse *TypeCalculator* gekapselt. Die von außen sichtbare Signatur lautet:

```
def dispatch Type getType(Expression it)
```

Xtend bietet von Haus aus dynamisches *Dispatching* an. Um das auszunutzen, wurde die Grammatik so geschrieben, dass *Xtext* für jede Operator-Klasse eine eigene Java-Klasse erzeugt, aus denen dann der Syntax-Baum zusammengesetzt wird. Es gibt also z. B. Java-Klassen wie

- *LogicalExpression* (*and, or*)
- *CompareExpression* (==, !=, <, <=, …)
- *PlusExpression* (+)
- *MinusExpression* (-)
- *LvalueExpression* (Variablenreferenzen, Attribute)
- *Literal* (mit vielen Unterklassen für die primitiven Datentypen)
- ...

[1] Man kann natürlich die Ergebnisse auch jederzeit *cachen*.

Für jede dieser Klassen gibt es also eine *getType*-Methode. Übergibt man *getType* irgendein *Expression*-Objekt, so delegiert *Xtend* zur Laufzeit an die passendste Methode.

Die einfachsten dieser Methoden sind die für die *Literale* – also Ausdrücke für die Konstanten der primitiven Datentypen.

```
def dispatch getType(StringLiteral it){
  return STRING_TYPE
}
```

Wobei STRING_TYPE eine Konstante vom Typ *StringType* ist. *StringType* ist wie folgt definiert:

```
class Type {}
class PrimitiveType extends Type {}
class StringType extends PrimitiveType {}
```

Analoge Definitionen gibt es für alle primitiven Datentypen.

Durch die Verwendung einer flachen Klassenhierarchie können auch für die Typen selbst *Dispatch*-Methoden verwendet werden.

Aber interessanter sind natürlich die Operatoren. Am einfachsten sind die logischen Operatoren *and* und *or*, denn das Ergebnis ist immer vom Typ *boolean*. Daher ist die Implementierung ebenso einfach wie von *Literalen*:

```
def dispatch getType(LogicalExpression it){
  BOOLEAN_TYPE
}
```

Gleiches gilt für die Vergleichsoperatoren. Zwar fordert die Beschreibung des Typ-Systems am Anfang, dass nur „vergleichbare" Operanden gültig sind, aber darum kümmert sich dem „*Loose typing, strict checking*"-Prinzip nach nicht die Berechnung, sondern erst die Prüfung.

Nach dem gleichen, geradezu deklarativen Verfahren können alle Ausdrücke behandelt werden, die einen festen Ergebnis-Typ haben.

Nur etwas schwieriger sind Ausdrücke, die Variablen „enthalten". Zwar spricht man von „Variablen in Ausdrücken", aber aus Sicht der Grammatik steht im Syntax-Baum nicht die Variable selbst, sondern eine Referenz auf ihre Definition. Und bei dieser Definition wiederum steht eine Referenz auf einen Typ, entweder auf einen primitiven Typ oder auf eine Entität. Daher berechnet sich der Typ einer Variablenreferenz so:

```
def dispatch getType(VariableRef it){
  variable?.type?.toType
}
```

Das heißt für den letzten Schritt, von der Referenz auf den Typ der Variablen zum tatsächlichen Typ, wird eine Hilfsmethode *toType(TypeRef)* aufgerufen, die hauptsächlich Referenzen auf primitive Datentypen auf die schon bekannten Konstanten wie STRING_TYPE und BOOLEAN_TYPE umsetzt.

Moment mal: Eine ähnliche Umsetzung wurde doch schon für die *Literale* mittels *getType* implementiert! Warum wird jetzt noch *toType* für Typ-Referenzen benötigt? *getType* ist nur für *Expression* definiert, aber *TypeRef* erbt nicht von *Expression*. Es wird in der Grammatik nur im *declare*-Statement und beim Typ von Attributen verwendet. Daher ist ein rekursiver Aufruf von *getType* an dieser Stelle nicht möglich, ohne die Signatur auf *Object* aufzuweichen, was doch ein wenig zu weit geht.

Eine weitere Neuigkeit versteckt sich aber noch in *toType*: Welchen Typ hat den die Variable *p*?

```
declare Person p
```

Bisher wurden für die Unterklassen von *Type* immer Konstanten verwendet. Das funktionierte, weil zur Beschreibung eines primitiven Typs keine weitere Information nötig ist, also ein *immutable Singleton* ausreicht. Für Entitäten muss man sich aber auch die Art der Entität merken, damit später Zuweisungen wie hier verboten werden können:

```
declare Person p
declare Address a
set p = a // ?
```

Um Informationen dieser Art zu verpacken, gibt es einen „parametrisierten Typ" *EntityType*:

```
@Data class EntityType extends Type {
  val Entity entity
}
```

Dabei ist zu beachten, dass *entity* nicht eine konkrete Person wie z. B. „Müller" referenziert, sondern nur *Person*.

Das Konzept der parametrisierten Typen kann auch für *Enum*-Typen oder auch für Listen von Entitäten u. Ä. verwendet werden. Am Ende dieses Weges lauern also die *Generics*.

Bleiben wir vorerst bei den Operatoren. Beim *Plus*-Operator gibt es die nächste Schwierigkeit: Im Gegensatz zu den bisherigen Ausdrücken muss man hier zuerst rekursiv die Typen der beiden Operanden berechnen. Das widerspricht dem *Loose-Typing*-Prinzip nicht, da nur für die Typ-Berechnung unnötige Rekursionen vermieden werden sollen; hier aber ist es zwingend notwendig. Hat man die beiden Typen berechnet, muss der Ergebnis-Typ ermittelt werden – so denn einer für die jeweilige Kombination existiert.

Am einfachsten realisiert man diese Abbildung mit einer Tabelle, die für jedes gültige Paar von Typen den Ergebnis-Typ enthält. Wird kein Typ gefunden, ist die Kombination ungültig. Bei kommutativen Operatoren muss man dabei natürlich beide Richtungen prüfen.

Im Beispiel wird die Tabelle durch *BinaryOperatorTable* gekapselt, wobei auch gleich Methoden für das Befüllen kommutativer Regeln angeboten werden. Es gibt zwei Instanzen dieser Klasse, eine für den *Plus*-Operator und eine für die anderen arithmetischen Operatoren, die sich bezüglich der Ergebnis-Typen nicht unterscheiden. Die Instanzen werden platzsparend mit dem *With-Operator* von *Xtend* initialisiert:

```
val plusTable = new BinaryOperatorTable => [
  put(
    STRING_TYPE, STRING_TYPE, STRING_TYPE
  )
  // ...
  // chrono-stuff
  putCommutative(
    DATE_TYPE, TIME_TYPE, TIMESTAMP_TYPE
  )
]
```

Die sehr dichte Darstellung der Überladungsregeln mit Hilfe der Tabelle funktioniert aus zwei Gründen:

- Es gibt in diesem Typ-System keine Sub-Typen und
- die Operatoren müssen keine parametrisierten Typen verarbeiten.

Beide Bedingungen zusammen erlauben es, den Ergebnis-Typ mit einem direkten Blick in die Tabelle zu ermitteln. Will man aber das Typ-System z. B. um einfache Listen erweitern und diese Listen mit *Plus* verketten, so gelingt das nicht. In diesem Fall wäre der Typ der beiden Operanden z. B. *List<Entity<Person>>*. Aber man kann ja schlecht für jede existierende Entität einen Eintrag in der Tabelle erzeugen.

Die Lösung wäre, stattdessen mehrere *Dispatch*-Methoden zu erzeugen. Die arithmetische Tabelle würde durch folgende Methoden ersetzt:

```
def dispatch Type getType(MultiplyExpression it){
  arithmeticType(left.type, right.type)
}

def dispatch arithmeticType(Type t1, Type t2){...}
def dispatch arithmeticType(FloatType t1, FloatType t2){...}
def dispatch arithmeticType(IntType t1, FloatType t2){...}
def dispatch arithmeticType(FloatType t1, IntType t2){...}
def dispatch arithmeticType(IntType t1, IntType t2){...}
```

Zur Laufzeit wird die jeweils korrekte Methode angesprungen und ersetzt dadurch den Tabellenzugriff. In den Methoden selbst kann man nun auch eventuelle Typ-Parameter programmatisch statt deklarativ behandeln.

Zusammenfassung Typ-Berechnung

Die bisher besprochenen Operatoren und das Vorgehen bei deren Typ-Berechnung kann als Vorlage für Spracherweiterungen dienen. Das Vorgehen ist immer ähnlich:

- *Loose Typing*-Prinzip beachten,
- schnelle Berechnung und
- Rekursion auf das Minimum beschränken.
- „Regel-artige“ bzw. deklarative statt imperative Programmierung durch
  - *Dispatch*-Methoden oder
  - Tabellen.

Den letzten Punkt demonstriert die Klasse *TypeCalculator*. Es gibt keine verschachtelten *If-Then-Else*-Kaskaden, nur mehrere dynamische *Dispatch*-Methoden und ein paar Tabellen. Jede dieser Methoden ist sehr kurz, oft nur ein einfaches *Return-Statement* und erinnert daher eher an Deklarationen von Regeln als an imperative Programmierung.

… Strict Checking

Der zweite Teil des Typ-Systems beschäftigt sich ausschließlich mit der Typ-Prüfung von Ausdrücken. Dazu traversiert man den AST des Modells und prüft für jede *Expression*, ob

- für sie selbst ein Typ berechnet werden kann und
- die Operanden akzeptable Typen haben.

Zusätzlich muss man noch alle „Ankerpunkte“ von Ausdrücken prüfen, ob für sie ebenfalls akzeptable Typen berechnet werden können. Dies betrifft z. B. den rechten Ausdruck des *set*-Statements. Zur Erinnerung: Ein Statement ist selbst kein Ausdruck, es verwendet nur welche. In richtigen DSLs hat man sehr viele dieser Ankerpunkte mit jeweils spezifischen Typ-Anforderungen.

Prüfungen werden in *Xtext* in speziellen Validator-Klassen implementiert – im Beispiel ist das die Klasse *TypesystemValidator*. *Xtext* traversiert den AST selbst und ruft alle Methoden auf, die eine *@Check*-Annotation und genau einen Parameter mit passendem Typ haben. Folgende Methode wird also für den *Plus*-Operator aufgerufen.

```
@Check
def void checkPlsExpr(PlusExpression it){...}
```

Um die Traversierung des AST muss sich die Typ-Prüfung also auch nicht kümmern.

Die Prüfung der Ankerpunkte und die Prüfung gültiger Operanden lässt sich in einem einzigen Verfahren effizient und übersichtlich zusammenfassen:

Man implementiert eine generische Prüfung (im Beispiel *checkExpectedType*, s. u.), die für jede *Expression* ausgeführt wird. Zuerst prüft sie, ob für den gerade untersuchten Ausdruck ein Typ berechnet werden kann. Wenn nicht, wird der gesamte Ausdruck als fehlerhaft markiert. Dabei wird in *handleNoType* noch unterschieden, ob es sich um einen überladenen Operator handelt oder nicht, um ggf. eine ausführlichere Fehlermeldung mit den berechneten Typen der Operanden zu erzeugen.

Wenn ein Typ ermittelt werden kann, wird mit *eContainer* der Vater des aktuellen Knotens im AST ermittelt, und mit *eContainingFeature* in welchem Zweig des Vaters sich der aktuell untersuchte Ausdruck befindet. Der Vater kann in diesem Fall ebenfalls eine *Expression* sein, oder aber ein *SetStatement* bzw. ein *EvalStatement*.

Nun benötigt man einfach wieder eine *Dispatch*-Methode (*expectedType*), die anhand des Typs des Vaters und des Features ermittelt, welchen Typ der Vaterknoten von dem aktuellen Knoten denn erwartet. Abschließend prüft der Check mit *isAssignableTo*, ob der tatsächliche Typ mit dem vom Vater gewünschten „kompatibel" ist. Zur Kompatibilität kommen wir gleich noch.

```
@Check
def void checkType(Expression it){
  val actualType = type
  if( actualType == null ){
    // If there is no result type then something is wrong!
    handleNoType
  }
  else {
    val expectedType = expectedType(
      eContainer, eContainingFeature
    )
    if( expectedType != null ){
      if( ! actualType.isAssignableTo(expectedType) ){
        val msg = '''Incompatible types:
        expected '«expectedType.asString»' but is actually
        '«actualType.asString»' '''
        error(msg, null, INCOMPATIBLE_TYPES)
      }
    }
  }
}
```

Zur Beruhigung: Dies ist die einzige und damit tiefste *If-then-else*-Kaskade im ganzen Typ-System!

Die verschiedenen Methoden für *expectedType* selbst sind normalerweise wieder sehr deklarativ, denn die Typen der Ankerpunkte sind meist fest vorgegeben. Da es normalerweise sehr viele Ankerpunkte für Ausdrücke gibt, hilft das wieder der Übersichtlichkeit ungemein. Auch viele Operatoren haben für ihre Operanden feste Vorstellungen – so können die logischen Operatoren *and* und *or* nur boolesche Operanden haben. Ebenso muss die Bedingung eines *if-then-else*-Ausdrucks immer einen booleschen Typ haben. Im Beispiel-Typ-System gibt es leider zu wenig Operatoren und Ankerpunkte, um das typische Mengengerüst für *expectedType* gut darzustellen.

Für Vergleiche benötigt man doch einen zusätzlichen Check, denn die beiden Operanden müssen nur einen „kompatiblen" Typ haben. Das heißt, hier kann der Vaterknoten seinen Kindern nicht mitteilen, welchen Typ genau er von ihnen erwartet.

Im Kern berechnet der Check daher die Typen seiner beiden Operanden selbst und prüft, ob sie kompatibel sind. Daher sollte für *CompareExpression* keine *expectedType*-Methode existieren, da dann die Kinder zweimal geprüft werden und eventuell zwei Fehlermeldungen erhalten.

Bleibt nur noch die Frage, was sich hinter dem mehrmals erwähnten „kompatibel" verbirgt.

In dieser Sprache ist die Definition sehr einfach. Ein Typ A ist zu einem Typ B kompatibel, wenn jeder Wert vom Typ A einer gedachten oder tatsächlichen Variable vom Typ B „zuweisbar" ist. Diese Definition ist bewusst nicht kommutativ.

„Zuweisbar" wiederum hat zwei Ausprägungen:

- Typ A ist ein Sub-Typ von B (oder derselbe Typ) oder
- es gibt eine „implizite Konvertierung" (*Coercion*) für alle Werte vom Typ A in Werte vom Typ B.

Dabei ist der Unterschied an einigen Stellen unklarer als man auf den ersten Blick meint. In Java ist selbstverständlich ein *String* ein Sub-Typ von *Object*. Aber ist z. B. ein *int* ein Sub-Typ von *double*?

Dafür spricht, dass alle Werte von *int* verlustfrei einem *double* zugewiesen werden können.

Dagegen spricht das Liskovsche Substitutionsprinzip, denn ein korrektes Programm, in dem *double* durch den *int* ersetzt wird, hat garantiert ein anderes Verhalten bezüglich *Typ-Subtyp*-Rundung, Wertebereich u.v.m. Also stehen die Typen *int* und *double* in keiner Beziehung zueinander.

Eine einfachere Daumenregel für diesen und ähnliche Fälle lautet: Eine *Coercion* erfordert zur Laufzeit oder beim Generieren immer ein wenig Arbeit. *Subtyping*

erfordert nur eine Neuinterpretation des Typs, aber keine Arbeit.

Die „Arbeit“ kann sehr klein sein, wie z. B. ein Maschinenbefehl, der ein Register erweitert. Demzufolge ist das Umschreiben des Bit-Musters von *int* nach *double* garantiert nicht-triviale Arbeit und damit eine Konvertierung.

Welche *Coercions* man in seine Sprachen einbaut, ist natürlich eine Design-Entscheidung der Sprache. Da aber *Coercions* im Prinzip an jeder Stelle auftreten können, sollte man erstens sparsam damit umgehen und zweitens dafür sorgen, dass alle *Coercions* zusammen keine Zyklen bilden können. Beides dient dem langfristigen Verständnis des Anwenders.

In der Beispielsprache gibt es übrigens nur eine allgemeine *Coercion*: *int* kann bei Zuweisungen in *float* umgewandelt werden.

Die gesamten Kompatibilitäts-Regeln sind im Beispiel in der Klasse *TypeCompatibility* mit nur 4 Regeln beschrieben.

Und fertig ist die Typ-Prüfung.

Zusammenfassung

Sowohl bei Typ-Berechnung als auch bei der Typ-Prüfung zeigt sich, dass der größte Teil mit einfachen Regeln statt mit komplexen Prozeduren beschrieben werden kann. Es sind auch nicht viele Regeln und da sie voneinander unabhängig sind, kann man sie leicht verstehen und an neue Anforderungen anpassen. So kann man z. B. einfach Listen und Sub-Typen hinzufügen. Versuchen Sie es doch selbst! Sie sehen ja, keine Theorie und keine *If-Then-Else*-Kaskaden.

Referenzen

[1] GitHub *types-with-system*
https://github.com/asheiduk/types-with-system

[2] Bettini, Lorenzo *Implementing Type Systems*
XtextCON2014 http://xtextcon.org/slides/Typesystems%20-%20Lorenzo%20Bettini.pdf

Kurzbiographie

Andreas Heiduk (andreas.heiduk@mathema.de) ist als Senior Consultant für die MATHEMA Software GmbH tätig. Seine Themenschwerpunkte umfassen die Java Standard Edition (JSE) und die Java Enterprise Edition (JEE). Daneben findet er alle Themen von hardware-naher Programmierung bis hin zu verteilten Anwendungen interessant.

Yeoman

Frontend Development Workflow

von Abderrahmen Dhouibi

Heutzutage sind viele Werkzeuge, Bibliotheken und Frameworks nötig, um ein Web-Projekt zu starten. Die Integration in das eigene Projekt erfordert einen hohen Arbeitsaufwand: Skripte, Download, Einbinden, Zusammenpacken, Auslieferung auf den Server… Dieser Prozess ist komplex und kostet viel Zeit. Es wäre schön, wenn ein neues Web-Projekt in wenigen Minuten erstellt werden könnte und viele Aufgaben und Updates von Ressourcen automatisiert laufen würden.

Hier kommt *Yeoman* [1] ins Spiel. *Yeoman* ist eine Sammlung von Werkzeugen und Bibliotheken. Es ist ein *Workflow*, der dem Entwickler helfen kann, schnell und mit wenig Aufwand Web-Projekte zu erstellen. Es generiert Code, verwaltet die Projekt-Pakete und stellt einen Entwicklungs-Server vor, damit der Entwickler ein Web-Projekt mit den notwendigen Ressourcen (*Frameworks, Plugins*) unter Berücksichtigung von Abhängigkeiten erhält.

Werkzeuge

Yeoman besteht aus 3 Komponenten: *yo* für das initiale Einrichten eines Projektes, *Grunt* für den Ablauf der Tasks und *Bower* für das Verwalten der Artefakte.

yo (Scaffolding)

yo erstellt die Ordnerstruktur und legt die Projektdateien an. Es arbeitet mit *Bower* zusammen um die Abhängigkeiten aufzulösen und um die *Grunt*-Konfigurationsdatei zu erstellen.

Grunt (Task Runner)

Nach Einrichten seiner Konfigurationsdatei, ist *Grunt* [2] für die wiederkehrenden, projektrelevanten Aufgaben und die Automatisierung der *Build*-Prozesse zuständig. Es basiert auf *JavaScript* und läuft dank *Node.js* auf allen gängigen Plattformen. Es erlaubt *Unit*-Tests um JavaScript-Code auf Syntaxfehler zu überprüfen, und minimiert den JavaScript-Code für das *Deployment*.

Um *Grunt* im Projekt nutzen zu können, legt *Yeoman* im Hauptverzeichnis des Projektes eine Konfigurationsdatei (*gruntfile.js*) ab (aktuelle stabile Version v0.4.5). Die *gruntfile.js* beinhaltet die Anweisungen und Informationen, die *Grunt* für das Verarbeiten des eigenen Projekts benötigt.

Beispiel einer *gruntfile.js*-Datei:

```
$ module.exports = function(grunt) {
  // Do grunt-related things in here
};
$ grunt.initConfig({ // 'package.json' wird importiert   });
$ grunt.registerTask(
  'serve', 'Compile then start a connect web server',
    function (target)
) { … };
§ grunt.registerTask('default', [
  'newer:jshint',
  'test',
  'build'
]);
```

Bower (Package Manager for the Web)

Hier werden Web-Pakete heruntergeladen um Abhängigkeiten aufzulösen und um Updates der Pakete vorzunehmen. *Bower* [3] ist ein Paket-Manager und in *Yeoman* integriert. Es sorgt dafür, dass man die Frameworks, Plugins und Ressourcen in das eigene Projekt integrieren kann. Nach jedem Update werden die Abhängigkeiten verschiedener Komponenten zueinander überprüft. Es stellt sicher, dass jedes *Bower-Package* existiert oder in der benötigten Version heruntergeladen wird (zum Beispiel ein *jQuery-Plugin* [4]).

Bower definiert die Packages in der Manifest-Datei *bower.json*. Diese ähnelt der *package.json*-Datei in *Node* [5] oder der *Gemfile*-Datei in *Ruby* [6].

Um *bower.json* zu erstellen, muss man *bower init* eingeben sowie einige Informationen übergeben (etwa Name, Version und *dependencies*).

Beispiel einer *bower.json*-Datei:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "dependencies": {
    "angular": "1.2.16",
    "json3": "~3.3.1",
    "es5-shim": "~3.1.0",
```

```
    "bootstrap-sass-official": "~3.2.0",
    "angular-resource": "1.2.16",
  },
  "keywords": {
    "123456789"
  },
  "devDependencies": {
    "angular-mocks": "1.2.16",
    "angular-scenario": "1.2.16"
  },
  "appPath": "app"
}
```

Installation

Zuerst muss eine Installation von *Node.js* und *Git* durchgeführt werden, da *Yeoman* auf *Node.js* basiert. Jetzt kann man *Grunt* und *Bower* als *Node-Package-Module* (*npm*) installieren. Mit einem Terminal-Befehl kann *Yeoman* installiert werden:

```
$ npm install -g yo
```

Grunt und *Bower* können automatisch (ab der *npm*-Version 1.2.10) oder manuell installiert werden.

Um zu prüfen, ob und welche Versionen installiert wurden, gibt man

```
$ yo --version && bower --version && grunt --version
```

ein. Den Standardgenerator kann man nun mit

```
$ npm install -g generator-webapp
```

installieren. In diesem Standardgenerator findet man *HTML5-Boilerplate*, *jQuery*, *Modernizr* und *Bootstrap*. Es gibt mehr als 1000 verfügbare Generatoren.

Starten eines kleinen Projektes mit *Yeoman*

Yeoman hat keine grafische Oberfläche, jedoch kann über das Terminal ein Projekt erstellt werden. Man muss nur einen Projekt-Ordner anlegen und in diesen wechseln.

```
$ mkdir myWebApp
$ cd myWebApp

$ …
$ …
$ …

$ npm install -g generator-angular
```

Danach muss der Generator konfiguriert werden. Der *AngularJS*-Generator stellt *Sass* (mit *Compass*) und *Twitter Bootstrap* zur Verfügung. *Yeoman* wird aufgerufen und *angular* als Parameter übergeben.

```
$ yo angular
```

Hier kann man sich entscheiden, ob für das Projekt Komponenten wie *Bootstrap* oder *Sass* benötigt werden.

Als Output erhält man im Projekt-Verzeichnis sauber geordnete Verzeichnisse und Dateien. In diesem Verzeichnis sind u. A. die folgenden Dateien enthalten:

- *index.html*: Basis HTML-Datei
- *404.html*, *favicon.ico*, und *robots.txt*
- *scripts*: JS-Dateien
  - *app.js*: Angular-Applikation-Code
  - *controllers*: Angular-Kontrollers
- *styles*: CSS-Dateien
- *views*: Angular-Templates

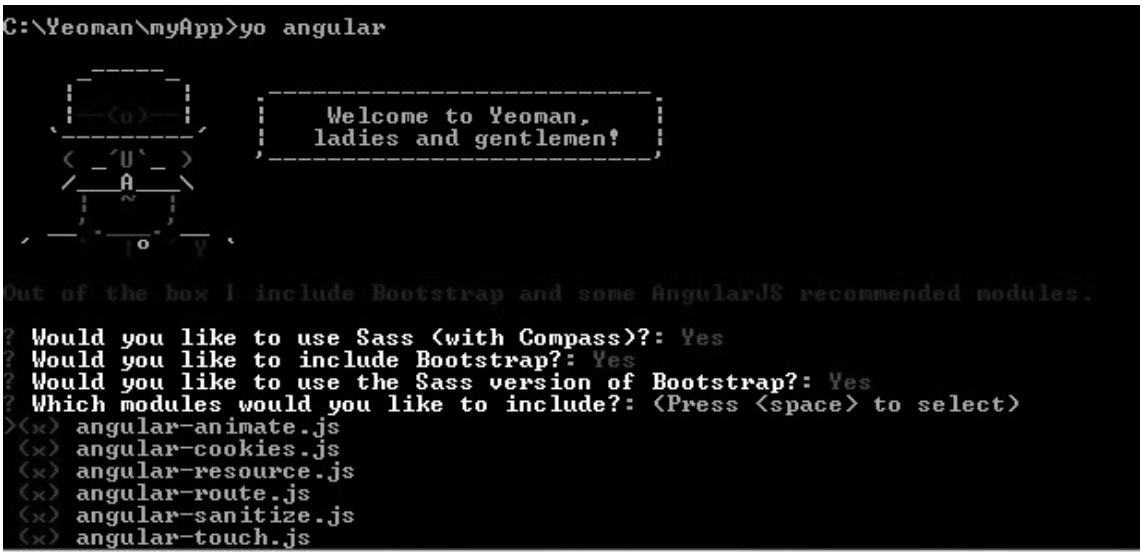


Abbildung 1

bower_components und *bower.json* sind JavaScript- und Web-Abhängigkeiten, die mit *Bower* installiert wurden. *Gruntfile.js*, *package.json* und *node_modules* sind Konfigurationen und Abhängigkeiten, die für *Grunt*-Jobs erforderlich sind. *test* ist ein Test-*Runner* und ermöglicht *Unit*-Tests für das Projekt.

Bower ermöglicht dem Nutzer die Installationen von weiteren Paketen, nach Abhängigkeiten zu suchen und diese zu aktualisieren. Man kann sie in der *bower.json*-Datei hinzufügen oder über das Terminal installieren, ohne die vorhandene *bower.json*-Datei anzupassen.

```
$ bower search <dep>
$ bower install <dep>..<depN>
$ bower update <dep>
```

Mit Hilfe von *Grunt* erstellt man noch einen lokalen *Node-based* HTTP-Server, der unter *localhost:9000* oder *127.0.0.1:9000* erreichbar ist.

```
$ grunt server
```

Dabei erkennt *Livereload* Änderungen in den Projekt-Dateien und lädt sie dann automatisch im Browser. Das *Livereloading* wird mit *Gruntfile.js* konfiguriert. Hier wird die Basis eines neuen Projekts zur Verfügung gestellt und man kann mit der Entwicklung starten.

Schließlich lässt sich das Ganze einfach mit

```
$ grunt test
```

testen und mit

```
$ grunt
```

in eine ausführbare Version umwandeln, die über den Browser abgerufen werden kann (Abb. 2)

Fazit und Ausblick

Yeoman ist ein erstaunliches Tool. Im Hintergrund verwendet es *Grunt* und *Bower*. So ist es möglich, dass ein Projekt mit Hilfe von *Yeoman* gestartet, dann aber nur mit *Grunt* und *Bower* weiterentwickelt wird.

Referenzen

[1] YEOMAN *The web's scaffolding tool for modern webapps*
http://yeoman.io

[2] Grunt *The JavaScript Task Runner*
http://gruntjs.com

[3] Bower *A package manager for the web*
http://bower.io

[4] The jQuery Foundation *jQuery*
http://jquery.com

[5] Joyent *node.js*
http://nodejs.org

[6] Ruby Community *Ruby Programming Lanuage*
https://www.ruby-lang.org

Kurzbiografie

Abderrahmen Dhouibi ist als Consultant und Entwickler für die MATHEMA Software GmbH in Erlangen tätig. Er interessiert sich für objektorientierte Programmierung und beschäftigt sich mit der Java Standard- und Enterprise-Entwicklung.

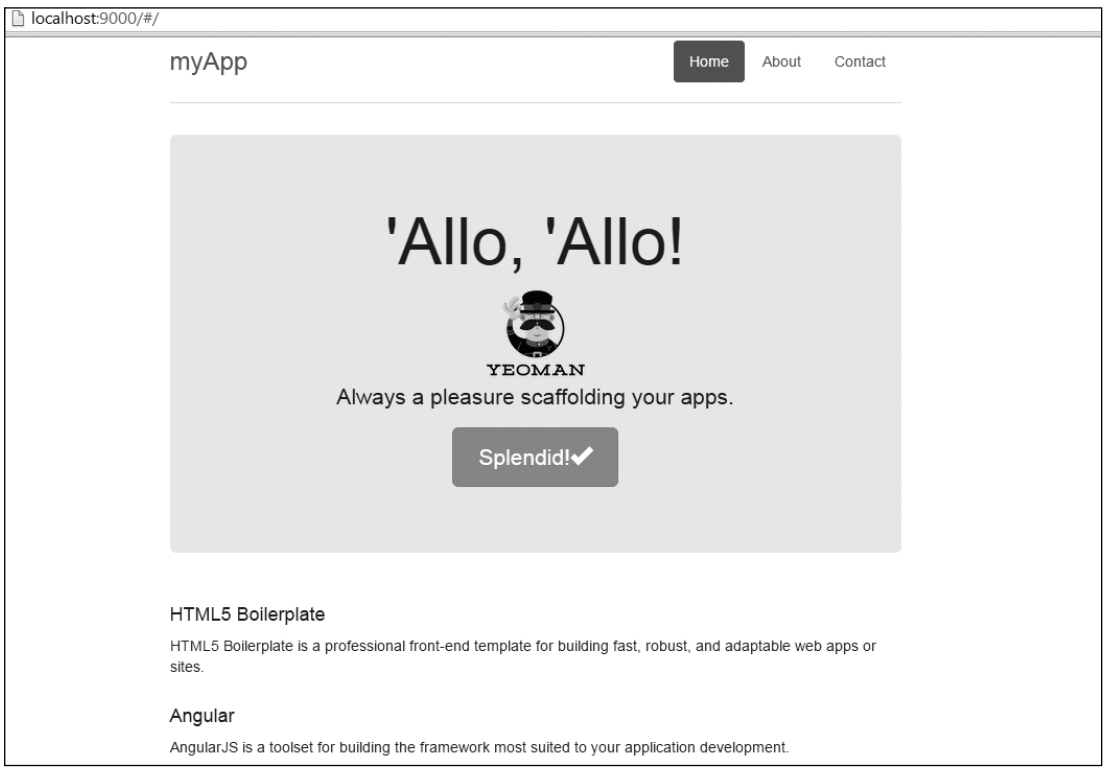


Abbildung 2

Datenbänkchen

WebStorage als eine Alternative zu Cookies

von Zeeshan Ali

WebStorage ermöglicht Websites bzw. Web-Servern, wie *Cookies*, Daten auf der Benutzerseite zu speichern, aber mit höherer Kapazität und einfacher Nutzung.

Cookies sind ein Standard-Mechanismus, der von verschiedenen Browsern und Web-Servern unterstützt wird und mit dem Websites Daten auf der Benutzerseite speichern können. *WebStorage* [1] ist eine Sammlung von Methoden, die auf neuen Standards basieren und dient dem gleichen Zweck. Es wird auch als *DOM Storage* oder *localStorage* bezeichnet. Die Methoden von *WebStorage* können, ebenso wie *Cookies*, mit Hilfe von Skripten der Benutzerseite (z. B. *JavaScript*) verwendet werden. Ursprünglich war *WebStorage* ein Teil von der *HTML5*-Spezifikation und wird deshalb oft als *HTML5 localStorage* oder *HTML5 WebStorage* erwähnt. Jetzt hat es aber eine eigene Spezifikation [2], die vom World Wide Web Consortium (W3C) betreut wird.

Speicherungsmodel

WebStorage speichert Daten als Schlüssel- und Werte-Paare in einem *Assoziative Array*, wobei die Schlüssel die Indizes der *Array*-Elemente sind. Zurzeit können Schlüssel und Werte nur als Zeichenketten (*Strings*) gespeichert werden und zusammengesetzte Objekte müssen erst zur Zeichenkette konvertiert und können nur dann gespeichert werden.

Speicherungsart

WebStorage stellt zwei Speicherungsarten zur Verfügung, die sich in Geltungsbereich (*Scope*) und Lebensdauer unterscheiden.

localStorage

Das ist die langfristige Speicherung von Daten mit folgenden Eigenschaften:

- Daten sind per Website oder Domain gespeichert
- Kein explizites oder automatisches Ablaufdatum
- Es gibt nur einen Datensatz pro Website oder Domain in einem Browser, sogar wenn die Web-Seiten von der gleichen Website in verschiedenen Tabs oder Fenstern eines Browsers offen sind
- Alle Web-Seiten einer Website haben Zugang zu ihrem Datensatz
- Web-Seiten einer Website haben keinen Zugang zu den gespeicherten Daten einer anderen Website
- Daten werden nicht gelöscht, wenn der Browser geschlossen wird
- Daten müssen explizit gelöscht werden

sessionStorage

Dies ist die kurzfristige oder Sitzungs-basierte Speicherung von Daten mit den folgenden Eigenschaften:

- Daten sind pro Fenster oder Tab gespeichert, sogar für die Web-Seiten, die zur gleichen Website gehören und in verschiedenen Fenstern oder Tabs eines Browsers geöffnet sind
- Jedes Fenster bzw. Tab hat seinen eigenen Datensatz
- Daten sind gelöscht, sobald das Fenster bzw. der Tab geschlossen wird

Methode	Beschreibung
localStorage.setItem("name","value")	speichert einen Wert
localStorage.getItem("name","value")	liest einen Wert aus
localStorage.removeItem("name")	löscht einen Wert
localStorage.clear()	löscht alle gespeicherten Werte

Abbildung 1: Die *localStorage*-Methoden

Technische Angaben

Das *window*-Objekt, das ein Fenster oder Tab in einem Browser repräsentiert, enthält folgende zwei Objekte als seine Eigenschaften, wenn ein Browser den *WebStorage*-Mechanismus unterstützt.

- *localStorage*
- *sessionStorage*

Beide dieser Objekte haben Methoden, damit Werte gespeichert, gelesen und gelöscht werden können. Die Methoden in Abbildung 1 gehören zum *localStorage*-Objekt. Um die Methoden des *sessionStorage*-Objektes zu nutzen, kann einfach *localStorage* durch *sessionStorage* ersetzt werden, z. B.:

```
sessionStorage.setItem("name","value")
```

Alternative Formen dieser *set*- und *get*-Methoden sind:

1. Als eine Eigenschaft z. B. *localStorage.name = "value";*
2. Als ein Element eines *Assoziative Array* z. B. *localStorage["name"] = "value"*;

Implementierungsbeispiele

Die folgenden Beispiele zeigen, wie eine Website *WebStorage* durch JavaScript verwenden kann [3].

1. localStorage-Beispiel

Dieses Beispiel basiert auf einer einfachen Website, die nur eine Web-Seite und ein kleines JavaScript enthält. Die Web-Seite zeigt einige Textzeilen und einen Button, der vom Benutzer angeklickt werden kann. Mit Hilfe von *localStorage* ist ein Zähler implementiert und der Benutzer wird informiert, wie oft er den Button bereits angeklickt hat. Die wichtigen Punkte dieses Beispiels sind:

- Der Zähler wird durch eine Aktualisierung der Web-Seite nicht zurückgesetzt.
- Wenn ein Fenster/Tab geschlossen wird und die Web-Seite dann in einem anderen Fenster/Tab aufgerufen wird, wird der Zähler ebenfalls nicht zurückgesetzt.
- Auch wenn die Web-Seite in mehreren Fenstern/Tabs eines Browsers offen ist, haben alle Instanzen nur einen Zähler oder nur einen gemeinsamen Datensatz.
- Im Falle, dass der Browser das *WebStorage* nicht unterstützt, wird der Benutzer mit einer Textnachricht auf der Web-Seite informiert.

```
<!DOCTYPE html>
<html>
<head>

<script>
function supportsLocalStorage() {
  return ('localStorage' in window) &&
    window['localStorage'] !== null;
}
function clickCounter() {
  if (supportsLocalStorage()) {
    if (localStorage.clickcount)
    {
      localStorage.clickcount = Number(
        localStorage.clickcount
      )+1;
    } else {
      localStorage.clickcount = 1;
    }
    document.getElementById("result").innerHTML =
      "Sie haben den button schon " +
        localStorage.clickcount + " mal geklickt.";
  } else {
    document.getElementById("result").innerHTML =
      "Ihr Browser unterstützt Webstorage nicht...";
  }
}
</script>
</head>

<body>
<p><button onclick="clickCounter()"
  type="button">Click me!</button></p>
<div id="result"></div>
<p>Click the button to see the counter increase.</p>
<p>Close the browser tab (or window), and try again,
  and the counter will continue to count (is not reset).</p>
</body>
</html>
```

2. sessionStorage-Beispiel

In diesem Beispiel wird dieselbe Website wie in dem *localStorage*-Beispiel verwendet, nur wird hier das *localStorage*-Objekt mit dem *sessionStorage*-Objekt ersetzt. Wenn die Web-Seite geladen ist und der Benutzer den Button anklickt, wird er wie im letzten Beispiel auch informiert, wie oft er den Button bereits angeklickt hat. Aber diesmal gibt es einige Unterschiede:

- Der Zähler ist durch Aktualisierung der Web-Seite nicht zurückgesetzt.

- Wenn die Web-Seite in einem anderen Fenster/Tab eines Browsers geöffnet ist, ist der Zähler in dem neuen Fenster/Tab zurückgesetzt und fängt erneut an.
- Wenn die Web-Seite in mehreren Fenstern/Tabs eines Browsers geöffnet ist, hat jedes Fenster/Tab seinen eigenen Datensatz. Das bedeutet, dass der Zähler in einem Fenster/Tab des Browsers komplett unabhängig von den anderen ist.

Im Falle, dass der Browser das *WebStorage* nicht unterstützt, wird der Benutzer ebenfalls mit einer Textnachricht auf der Web-Seite informiert.

```
<!DOCTYPE html>
<html>
<head>
<script>
function supportsSessionStorage() {
  return ('sessionStorage' in window) &&
    window['sessionStorage'] !== null;
}
function clickCounter() {
  if(supportsSessionStorage()) {
    if (sessionStorage.getItem("clickcount")) {
      sessionStorage.setItem(
        "clickcount",Number(sessionStorage.clickcount)+1
      );
    } else {
      sessionStorage.setItem("clickcount","1");
    }
    document.getElementById("result").innerHTML =
      "You have clicked the button "
        + sessionStorage.getItem("clickcount")
          + " time(s) in this session.";
  } else {
    document.getElementById("result").innerHTML =
      "Sorry, your browser does not support web storage...";
  }
}
</script>
</head>
<body>
<p><button onclick="clickCounter()" type="button">Click
me!</button></p>
<div id="result"></div>
<p>Click the button to see the counter increase.</p>
<p>Close the browser tab (or window), and try again, and
the counter is reset.</p>
</body>
</html>
```

3. Beispiel eines zsammengesetzten Objekts

Da *WebStorage* nur die Speicherung von Zeichenketten erlaubt, können zusammengesetzte Objekte so nicht gespeichert oder gelesen werden. Diese Objekte müssen zu Zeichenketten konvertiert werden und können erst dann gespeichert werden. Beim Lesen müssen die gespeicherten Zeichenketten noch einmal zum korrekten Typ konvertiert werden. Beispielsweise kann ein *JSON*-Objekt durch *WebStorage* gespeichert und gelesen werden, mit Hilfe der Methoden des globalen JSON-Objektes, das von JavaScript bereitgestellt wird.

```
//Ein Objekt definieren
Var obj = {
"item1": "value1",
"array1": [1, 2, 3]
};
// Objekt speichern
localStorage.setItem("obj", JSON.stringify(obj));

//Objekt abholen
console.log(JSON.parse(localStorage.getItem("obj")));
```

Vorteile von *WebStorage* im Vergleich mit *Cookies*

1. Keine Datenübertragung zum Server

Der Inhalt von *Cookies* wird mit jeder Anfrage im *HTTP*-Header zum Server geschickt, während *WebStorage* eine exklusive benutzerseitige Speicherung von Daten ist. Wenn notwendig, muss der Inhalt von *WebStorage* explizit zum Server, mit *AJAX* oder eine anderen Methode, geschickt werden.

2. Kapazität

WebStorage kann viel mehr Daten speichern als *Cookies*. Die Spezifikation [2] erlaubt eine Größe bis 5 MB pro Website/Domain. In der Praxis gibt es aber Unterschiede zwischen den verschiedenen Browsern, einige erlauben weniger als 5 MB und andere mehr. Es gibt aber einige Websites, die es dem Benutzer ermöglichen, die genaue Kapazität der *localStorage* und *sessionStorage* für einen Browser herauszufinden [4].

3. Kein Ablaufdatum

localStorage hat kein explizites Ablaufdatum und die gespeicherten Daten können nur mit Methoden wie *removeItem()*, *clear()* oder durch Browser-Einstellungen gelöscht werden. Die gespeicherten Daten von *sessionStorage* werden aber sofort nach der Schließung des Fensters/Tabs oder Browsers gelöscht.

Browser-Unterstützung für *WebStorage*

WebStorage wird von den aktuellen Versionen aller Browser unterstützt [5]. Es gibt keine Code-Unterschiede und dasselbe Skript funktioniert in allen Browsern, die *WebStorage* unterstützen.

Fazit

Der *WebStorage*-Mechanismus stellt mehr Kapazität zur Verfügung und die Methoden sind auch leicht zu verwenden. Deshalb sollte *WebStorage* statt *Cookies* zur Speicherung von Daten der Benutzerseite verwendet werden. Allerdings dürfen sensitive Information wie z. B. Usernamen, Passwörter und Kreditkartennummern niemals auf der Benutzerseite mit *WebStorage* oder einer anderen Methode gespeichert werden. Nur falls kleine Werte zum Server gesendet werden, sollten *Cookies* verwendet werden.

Referenzen

[1] W3C *WebStorage*
http://www.w3.org/TR/webstorage

[2] Wikipedia *WebStorage*
http://en.wikipedia.org/wiki/Web_storage

[3] w3schools
http://www.w3schools.com/html/tryit.asp?filename=tryhtml5_webstorage_local_clickcount

[4] *WebStorage Support Test*
http://dev-test.nemikor.com/web-storage/support-test

[5] HTML5 *Funktionen Speicher*
http://www.html5rocks.com/de/features/storage

Kurzbiographie

Zeeshan Ali ist als Software-Entwickler bei der MATHEMA Software GmbH tätig. Nach dem Informatik-Studium hat er erste Erfahrungen im Bereich OOP mit Android-Entwicklung gesammelt. Aktuell beschäftigt er sich mit der Entwicklung und dem Testen von Java Enterprise-Anwendungen. Neben dem Beruf ist er daran interessiert andere Kulturen kennenzulernen und seine Weltanschauung zu erweitern.

Des Programmierers kleine Vergnügen

Der Weg aus der Überlauf-bredouille

von Michael Wiedeking

In dem letzten Vergnügen ging es darum, über einen beliebigen Bereich ganzer Zahlen zu iterieren. Dort blieb leider unerwähnt, dies nur unter Zuhilfenahme von vorzeichenbehafteten Zahlen zu machen, wobei diese einschränkende Bedingung nicht den Schwierigkeitsgrad erhöhen sollte, sondern dem Umstand geschuldet war, dass etwa Java-Entwicklern – höchstbedauerlich – vorzeichenlose Zahlen einfach nicht zur Verfügung stehen. Dieses Vergnügen zeigt, wie man es noch besser machen kann.

Die beim letzten Vergnügen gefundene Lösung [1] benötigte für beliebige Bereiche nur zwei zusätzliche Instruktionen pro Schleifendurchlauf. Wie ein aufmerksamer Leser[1] bemerkte, müsste das doch auch mit nur einer zusätzlichen Instruktion pro Schleifendurchlauf erledigt werden können, indem einfach bestimmt wird, wie oft die Schleife durchlaufen werden muss, und das dann auch entsprechend oft getan wird.

Recht hat er. Angenommen man möchte von *first* bis *last* (inklusive) alle Zahlen durchlaufen, dann muss die Schleife n Mal

$$n = last - first + 1$$

durchlaufen werden. Sind die beiden Werte *first* und *last* dabei mini- bzw. maximal, dann würden etwa bei einer vorzeichenbehafteten 32-Bit-Zahl die Werte von MIN = -2^{31} bis MAX = $2^{31} - 1$ durchlaufen werden. Gemäß obiger Berechnungsvorschrift wären das also

$$\begin{aligned} n &= (2^{31} - 1) - (-2^{31}) + 1 \\ &= 2^{31} - 1 + 2^{31} + 1 \\ &= 2^{31} + 2^{31} \\ &= 2 \cdot 2^{31} \\ &= 2^{32} \end{aligned}$$

Durchläufe. Leider lässt sich diese Zahl nicht in 32 Bit darstellen und würde wegen der Überläufe 0 ergeben. Dennoch muss bei *first* ≤ *last* die Schleife mindestens einmal durchlaufen werden. Also wird nur noch ein Zähler benötigt, mit dem man $2^{32} - 1$ Durchläufe zählen kann, und das lässt sich in einer 32-Bit-Zahl unterbringen – wenn man denn eine vorzeichenlose Zahl zur Verfügung hätte.

```
if (first <= last) {
  unsigned n = last - first + 1;
  int i = first;
  do {
    …
    i += 1;
    n = n - 1;
  } while (n != 0);
}
```

Aber einen vorzeichenlosen Wert (mal abgesehen davon, dass obiger *unsigned*-Wert auch zu klein wäre) gibt es doch nicht. Macht aber nix. Hier kann man sich tatsächlich auch einer vorzeichenbehafteten Ganzzahl bedienen.

In einem vergangenen Vergnügen [2] wurde bereits gezeigt, dass sich bei den arithmetischen Operationen Addition, Subtraktion und Multiplikation die vorzeichenlose Variante exakt so verhält wie die vorzeichenbehaftete; nur eben, dass die unterschiedliche Interpretation zu verschiedenem Überlaufverhalten führt. Das also n durch die Berechnung von *last* − *first* + 1 auch negativ sein kann, tut dem Abzählen als solches keinen Abbruch.

Ist n positiv, so läuft ja ohnehin alles korrekt. Hat es aber einen Überlauf gegeben und wird n dadurch negativ, so ist das ja nur eine Sache der Interpretation. Dekrementiert man nun in jedem Schleifendurchlauf n, so wird n zunächst immer kleiner. Beim vorzeichenbehafteten MIN = -2^{31} gibt es dann den Überlauf und man landet nach dem Dekrement beim vorzeichenbehafteten MAX = $2^{31} - 1$. Und ab da läuft ja, weil positiv, wieder alles korrekt.

[1] David Artizada aus Erlangen.

Der größtmögliche Wert für n ist 2^{32}, was als 0 dargestellt wird. Da aber vor dem Test dekrementiert wird, wird die 0 erst dann erreicht, wenn alle möglichen Zahlen durchlaufen worden sind. Damit wurde n insgesamt 2^{32} Mal dekrementiert – voilà! Ergibt die Berechnung von n einen um k kleineren Wert, dann wird analog auch die Schleife k Mal weniger durchlaufen. Und wie schon erwähnt wird der Fall, dass die Schleife gar nicht durchlaufen werden soll, durch den Test $first \le last$ abgefangen.

Leider ist es damit nicht getan, denn dieser Ansatz soll ja auch dann noch funktionieren, wenn die Schrittweite Δ von i einen anderen Wert als 1 hat. An obigem Vorgehen muss man nichts ändern, nur dass jetzt die Anzahl der Schleifendurchläufe und das Inkrement etwas anders berechnet werden müssen. Das Inkrement wird immer noch trivial mit

$$i \mathrel{+}= \Delta;$$

berechnet (mit $\Delta \ge 1$). Die Anzahl der Durchläufe hängt dementsprechend von Δ ab. Ist $first \le last$, dann wird die Schleife mindestens einmal durchlaufen. Dann wird die Schleife noch so oft durchlaufen, solange

$$first + n \cdot \Delta \le last$$

ist (wenn das ohne Einschränkungen durch die Wortbreite korrekt ausgerechnet wird). Die korrekte Anzahl erhält man dementsprechend, wenn

$$n = \left\lfloor \frac{last - first + \Delta}{\Delta} \right\rfloor$$

ist, wobei der Quotient nach unten gerundet wird.

Die Addition mit Δ im Zähler stellt sicher, dass beispielsweise bei $\Delta = 3$, $first = 0$ und $last = 3$ die Schleife auch wirklich zweimal durchlaufen wird und nicht nur einmal.

Damit mein Vergnügen vom letzten Mal [1] nun doch nicht in ein all zu schlechtes Bild gerückt wird, sei hier erwähnt, dass obige Division natürlich vorzeichenlos funktionieren muss. Aber auch das ist kein Problem, da wir uns schon im Februarvergnügen aus diesem Jahr [3] ausführlich damit beschäftigt haben.

So verbleibt nur der unschöne Fall, dass $first$ und $last$ wieder mini- bzw. maximal sind und $\Delta > 1$ ist. Denn dann ergibt der Zähler

$$\mathrm{MAX} - \mathrm{MIN} + \Delta = 2^{32} - 1 + \Delta$$

was als $\Delta - 1$ dargestellt wird. Die Division wird also ungeachtet der Tatsache, dass $first \le last$ ist,

$$n = \left\lfloor \frac{\Delta - 1}{\Delta} \right\rfloor = 0$$

liefern. Wegen dieses speziellen Falls ist es also – weil wir wissen, dass bei $first \le last$ die Schleife sicher durchlaufen wird – besser

$$n = \left\lfloor \frac{last - first}{\Delta} \right\rfloor + 1$$

zu berechnen. Und damit ist auch dieser Sonderfall eliminiert.

Zuletzt bleibt nur noch zu erwähnen, dass diese Methode für fast alle Fälle die bessere sein sollte, da ja nun wirklich pro Schleifendurchlauf nur eine einzige Instruktion mehr als im Standardfall

```
for (i = 0; i < n; i++)
```

benötigt wird – also eine ganze Instruktion weniger als die Lösung im letzten Vergnügen benötigte. Für ganz Eilige könnte sich bei kleinem n allerdings die teure Division der neuen Lösung negativ bemerkbar machen, so dass in dieser Zeit die „alte“ Schleife schon mehrfach durchlaufen sein könnte, bevor die Division ihr Ergebnis liefert.

Ungeachtet dessen zeigt sich hier wieder einmal, dass es – wenn es wirklich zeitkritisch ist – nicht genügt, Vorzeichenloses zu emulieren, sondern dass es nötig ist, die gegebenen vorzeichenbehafteten Mittel korrekt auszureizen.

Referenzen

[1] Wiedeking, Michael
Des Programmierers kleine Vergnügen – Schleifenüberlaufdilemma,
KaffeeKlatsch, Jahrgang 7, Nr. 10, S. 15f, Bookware, Oktober 2014
http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2014-10.pdf

[2] Wiedeking, Michael
Des Programmierers kleine Vergnügen – Ohne Vorzeichen,
KaffeeKlatsch, Jahrgang 7, Nr. 1, S. 20f, Bookware, Januar 2014
http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2014-01.pdf

[3] Wiedeking, Michael
Des Programmierers kleine Vergnügen – Geteiltes Vergnügen,
KaffeeKlatsch, Jahrgang 7, Nr. 2, S. 12f, Bookware, Februar 2014
http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2014-02.pdf

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt“ Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

Kaffeesatz

Superwichtig

von Michael Wiedeking

Man könnte ja meinen, dass der Computer die einflussreichste technische Errungenschaft des letzten Jahrhunderts ist. Sicherlich ist er auch nicht mehr aus unserem Leben wegzudenken, ist er doch praktisch allgegenwärtig. Aber trotzdem steht der Computer höchstens auf Platz 2.

Zu berechnen gab es ja schon immer etwas, und mit Hilfe des Computers sind inzwischen Berechnungen möglich, die früher einfach nie zu bewältigen gewesen wären. So hat er beispielsweise den rechnenden Büroangestellten gleichen Namens vollständig vom Arbeitsmarkt verdrängt.

Ein solcher „Computer" wird übrigens erstmalig 1613 in dem Buch *The yong mans gleanings* von Richard Braithwait namentlich erwähnt [1]:

> „*I haue read the truest computer of Times, and the best Arithmetician that euer breathed, and he reduceth thy dayes into a short number.*"

Aber der Computer führt ja inzwischen nicht nur Berechnungen durch, sondern wird überall zur Messung, Kontrolle und Steuerung eingesetzt. So werden etwa durch den Einsatz in der Landwirtschaft im Zusammenhang mit GPS millimetergenau Furchen gezogen, die den Ertrag auf einem Feld maximieren. Er wird im Haus zur Steuerung der Heizung eingesetzt, um dieses mit dem geringstmöglichen Energieeinsatz aufzuwärmen. Oder er startet den Automotor so präzise, dass dieser an jeder roten Ampel abgeschaltet werden kann.

Seit man den Computer mit sich rumschleppen kann, hat er die Kommunikationsmöglichkeiten noch einmal um ein Vielfaches verändert. Über das Internet ist man nicht nur überall erreichbar, man hat auch von überall den Zugriff auf einen unglaublichen Fundus von Wissen. Nachrichten aus den fernsten Regionen der Welt erreichen einen sofort und auch die entferntesten Verwandten rücken damit in die (möglicherweise unerwünscht) unmittelbare Nähe.

Und nicht zuletzt die Roboter leisten seit Jahrzehnten unschätzbare Dienste. Egal ob sie Autos bauen oder Platinen bestücken; sie sind ebenfalls aus unserem Alltag nicht mehr wegzudenken. Unbemerkt leisten sie zudem noch ungewürdigt ihren Dienst in Autos, um dieses in der Spur zu halten, es in der Not abzubremsen oder einzuparken. Und es wird wahrscheinlich nicht mehr all zu lange dauern, dann werden sie uns chauffieren, uns pflegen und den Haushalt machen.

Dennoch, wenn man dem Ökonomen Ha-Joon Chang glaubt, gibt es eine Erfindung, die unser Leben deutlich mehr beeinflusst hat: Die Waschmaschine. Die Erfindung von Waschmaschine und Bügeleisen hat die Zeit für das Wäschewaschen um mehr als 80 % reduziert (von 4 Stunden auf 41 Minuten) und das des Bügelns auf mehr als 60 % (von 4,5 auf 1,75 Stunden) [2]. Und erst damit war es möglich, dass der Anteil Frauen, die außerhalb ihres Haushaltes arbeiten konnten, von wenigen Prozent auf über 80 % ansteigen konnte.

Daraus folgte, dass nicht nur Söhne gefördert wurden, sondern auch die Töchter. Damit wurden diese unabhängiger, konnten gänzlich entweder ohne einen versorgenden Mann auskommen oder etwa glaubhaft mit Trennung drohen. Darüber hinaus wurde man unabhängiger von der Mithilfe der Kinder, was deren Anzahl reduzierte. Und jetzt sind wir gesellschaftlich da, wo wir sind – auch ohne Internet und Computer.

Aber machen wir uns nichts vor: Auch die Waschmaschine wird inzwischen äußerst effizient mit Hilfe eines Computers gesteuert.

Referenzen

[1] Wikipedia *Computer*, http://en.wikipedia.org/wiki/Computer

[2] Chang, Ha-Joon *Warum die Welt stärker von Waschmaschinen als vom Internet verändert wurde,* Geo – Das Reportage Magazin, S. 86f, Gruner + Jahr, April 2013

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt" Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch? Dann geben Sie uns doch bitte Bescheid.

Bookware, Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0, Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
http://www.jug-bb.de
Kontakt: Herr Ralph Bergmann (orga@jug-bb.de)

Java UserGroup Bremen
http://www.jugbremen.de
Kontakt: Rabea Gransberger (rgransberger@gmx.de)

JUG DA
Java User Group Darmstadt
http://www.jug-da.de
Kontakt: jug-da-orga@googlegroups.com

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxony.de
Kontakt: Herr Falk Hartmann
(falk.hartmann@jugsaxony.org)

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
http://www.rheinjug.de
Kontakt: Herr Heiko Sippel (info@rheinjug.de)

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
http://www.ruhrjug.de
Kontakt: Herr Heiko Sippel (heiko.sippel@ruhrjug.de)

JUGF
Java User Group Frankfurt
http://www.jugf.de
Kontakt: Herr Alexander Culum
(alexander.culum@web.de)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o Stefan Koospal
http://www.java.de (office@java.de)

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
http://jug-karlsruhe.de
(jugkarlsruhe@gmail.com)

JUGC
Java User Group Köln
http://www.jugcologne.org
Kontakt: Herr Michael Hüttermann
(michael@huettermann.net)

jugm
Java User Group München
http://www.jugm.de
Kontakt: Herr Andreas Haug (ah@jugm.de)

JUG Münster
Java User Group für Münster und das Münsterland
http://www.jug-muenster.de
Kontakt: Herr Thomas Kruse (tkjugi@sforce.org)

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
http://www.jug-n.de
Kontakt: Frau Natalia Wilhelm
(info@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V.
c/o Dr. Michael Paus
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)
Rainer Anglett (ra@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
http://www.jugs.ch (info@jugs.ch)

.NET User Groups

Deutschland

.NET User Group Bonn
.NET User Group "Bonn-to-Code.Net"
http://www.bonn-to-code.net (mail@bonn-to-code.net)
Kontakt: Herr Roland Weigelt

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Udo Neßhöver, Frau Ulrike Stirnweiß
(info@dodnedder.de)

.NET UserGroup Frankfurt
http://www.dotnet-usergroup.de

.NET User Group Hannover
http://www.dnug-hannover.de
Kontakt: (dnug@indisoftware.de)

INdotNET
Ingolstädter .NET Developers Group
http://www.indot.net
Kontakt: Herr Gregor Biswanger
(gregor.biswanger@web-enliven.de)

DNUG-Köln
DotNetUserGroup Köln
http://www.dnug-koeln.de
Kontakt: Herr Albert Weinert (info@der-albert.com)

.NET User Group Leipzig
http://www.dotnet-leipzig.de
Kontakt: Herr Alexander Groß (agross@dotnet-leipzig.de)
Herr Torsten Weber (tweber@dotnet-leipzig.de)

.NET Developers Group München
http://www.munichdot.net
Kontakt: Hardy Erlinger (hardy_erlinger@hotmail.com)

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Frau Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET Developers Group Stuttgart
Tieto Deutschland GmbH
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Herr Michael Niethammer

.NET Developer-Group Ulm
c/o artiso solutions GmbH
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

.NET User Group Austria
c/o Global Knowledge Network GmbH,
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx
Kontakt: Herr Christian Nagel (ug@christiannagel.com)

Software Craftsmanship Communities

Deutschland, Schweiz, Österreich

Softwerkskammer – Mehrere regionale Gruppen und Themengruppen unter einem Dach
http://www.softwerkskammer.org
Kontakt: Nicole Rauch (nicole.m@gmx.de)

MATHEMA

meet the experts of enterprise infrastructure

HTML5, CSS3 und JavaScript
9. – 12. März 2015, 21. – 24. September 2015
1.650,– € (zzgl. 19 % MwSt.)

Entwicklung mobiler Anwendungen mit iOS
20. – 22. April 2015, 5. – 7. Oktober 2015
1.250,– € (zzgl. 19 % MwSt.)

Fortgeschrittenes Programmieren mit Java
Ausgewählte Pakete der Java Standard Edition
4. – 6. Mai 2015, 19. – 21. Oktober 2015
1.350,– € (zzgl. 19 % MwSt.)

Weiterführende Programmierung unter C#
Umstieg auf C# 4.5/5 und Einführung in fortgeschrittene Konzepte
11. – 13. Mai 2015, 26. – 28. Oktober 2015
1.350,– € (zzgl. 19 % MwSt.)

AngularJS
18. – 19. Mai 2015, 7. – 8. Dezember 2015
950,– € (zzgl. 19% MwSt.)

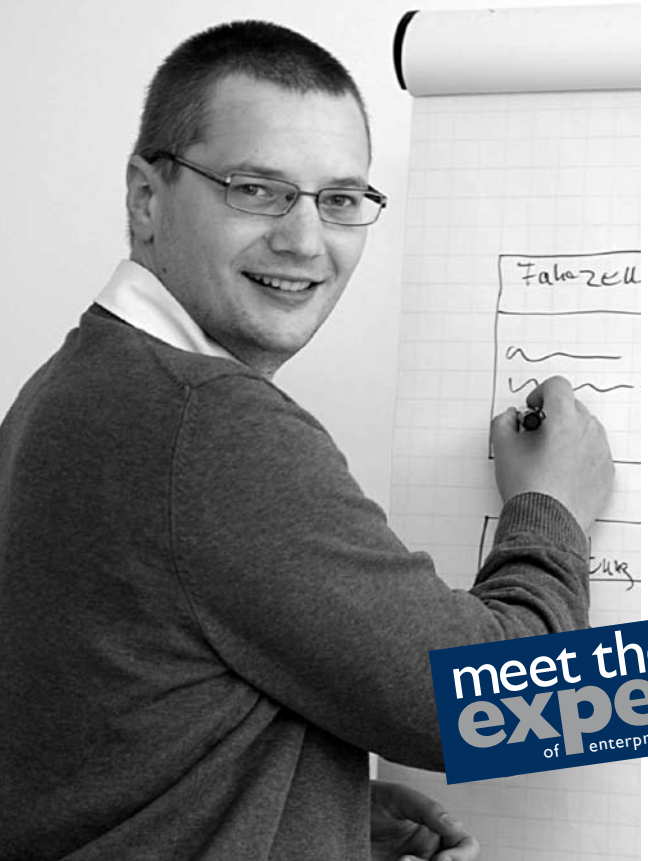
Anwendungsentwicklung mit der Java Enterprise Edition
22. – 26. Juni 2015, 30. Nov – 4. Dez. 2015
2.150,– € (zzgl. 19 % MwSt.)

Lesen bildet. **Training macht fit.**

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

MATHEMA

training



18. – 19. Mai 2015, 7. – 8. Dezember 2015,
950,– € (zzgl. 19% MwSt.)

AngularJS

Trainer: Moritz Herrmann

Das JavaScript Framework AngularJS ermöglicht Ihnen die Entwicklung von modernen Web-Applikationen basierend auf dem MVVM-Prinzip. Aufbauend auf unseren umfangreichen Praxiserfahrungen mit AngularJS vermitteln wir Ihnen in diesem Kurs alle wichtigen Grundlagen, Komponenten und Konzepte des Frameworks. Durch viele Beispiele und Übungen können Sie Ihr neu erworbenes Wissen sofort praktisch anwenden und vertiefen.

Selbstverständlich führen wir das Training auch gerne in Ihrem Haus durch. Sprechen Sie uns an und wir entwickeln ein auf Sie persönlich zugeschnittenes Angebot.

Weitere Informationen und Kursanmeldung unter:
http://www.mathema.de/training/katalog/angularjs

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de | Ansprechpartnerin:
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de | Madeline Hörner

Das Allerletzte

```
int pi_value() {
  return 3.14;
}
```

Dies ist kein Scherz!
Dieser Code wurde tatsächlich als
„Beispiel für eine C-Funktion“ in dem Lehrbuch (sic!)
„Pro iOS Apps Performance Optimization“
gefunden.

Ist Ihnen auch schon einmal ein Exemplar dieser
Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im Dezember.

Herbstcampus

Wissenstransfer par excellence

31. August – 3. September 2015
in Nürnberg