Renée Bäcker

Operatoren überladen

Eigentlich alle Objektorientierten Sprachen kennen den Begriff des Überladens. In den meisten Sprachen - wie z.B. Java - werden Methoden überladen, damit je nach Anzahl oder Art der Parameter die richtige Methode aufgerufen wird. Perl bietet weder Methoden-Signaturen noch das Überladen von Methoden. Deshalb ist der Begriff frei für etwas anderes: Operatoren überladen.

In Perl können alle Built-in Operatoren für eine Klasse mit einer neuen Bedeutung versehen werden, aber wozu das Ganze?

Als Beispiel mal eine kleine Klasse `Magazin`:

```
package Magazin;

use Moose;
use Moose::Util::TypeConstraints;

has title => (
  isa => 'Str', is => 'ro',
);

has date => (
  isa => 'DateTime', is => 'ro',
  coerce => 1,
);

has published => (
  isa => 'Bool', is => 'ro',
);

no Moose;
1;
```

Die Definition von des Typs *DateTime* und der Umwandlung von Strings in ein `DateTime`-Objekt wurde hier weggelassen, weil das ein Moose-Thema ist (siehe auch Ausgabe 15 von $foo).

Mit dieser Klasse können sehr einfache Objekte erstellt werden, die einen Titel haben und ein Veröffentlichungsdatum:

```
my $m1 = Magazin->new(
    title => 'Winter 2012',
    date  => '01.11.2012',
);
```

So weit, so gut. Mit diesem Objekt kann man wie gewohnt arbeiten:

```
use feature 'say';
say $m1->title;
say $m1->date;
```

In manchen Situationen wäre es aber schön, wenn man nicht immer die Methoden aufrufen müsste, sondern automatisch der richtige Wert geliefert wird

```
say sprintf
  "Die übernächste Ausgabe nach
     $m1 ist %s",
     $m1 + 2;
```

Wenn man das so ausführt sieht die Ausgabe ungefähr so aus:

```
Die übernächste Ausgabe nach
     Magazin=HASH(0x35a8b90) ist
     56265618
```

Nicht unbedingt das was man haben will. Jetzt kommt das Überladen ins Spiel.

Das Modul `overload` ist Teil des Perl-Kerns (seit 5.002). Mit diesem Modul kann man festlegen, wie Operatoren überladen werden. Für das oben gezeigte Beispiel kommt das Stringifizieren des Objektes in Frage:

```
use overload '""' => 'print_title';
```

Ab sofort wird immer die anonyme Subroutine ausgeführt wenn das Objekt stringifiziert wird. Diese Codestücke

```
print "$m1";
print $m1 . '';
# und ähnliches
```

werden zu `$m1->print_title()` konvertiert.

Die Methode `print_title` ist ganz einfach:

```
sub print_title { $_[0]->title }
```

Ein kleiner Einschub: Man kann hier nicht einfach den Accessor `title` nehmen, weil *title* ein read-only Attribut ist und den überladenden Funktionen immer drei Werte übergeben werden.

Numerische Operatoren überladen

Genauso wie beim Stringifizieren kann man auch numerische Operatoren überladen.

Nicht-Kommutative Operatoren

Man muss auch immer zwischen kommutativen und nicht-kommutativen Operatoren unterscheiden. Es sollte noch aus der Grundschule bekannt sein, dass *2 + 3* das gleiche ist wie *3 + 2*. Aber *2 / 3* ist nicht das gleiche wie *3 / 2*. Aus diesem Grund übergibt Perl - wie oben schon erwähnt - 3 Werte an die Subroutine: Das Objekt, den anderen Operanden und *swap*. *swap* ist immer dann gesetzt, wenn das Objekt eigentlich der rechte Operand ist, z.B. bei `2 - $obj`.

```
package Number;

use Moose;

use overload '-' => \-

has value => ( isa => 'Num', is => 'ro' );

sub minus {
    my ($self,$r,$swap) = @_;

    my $l = $self->value;

    ($l,$r) = ($r,$l) if $swap;
    my $result = $l - $r;
    return $result;
}

1;


my $obj = Number->new(5);
say $obj - 2;
```

Hier sind die übergebenen Werte `$obj`, 2 und ''.

```
say 2 - $obj;
say 2 + $obj;
```

Hier wird als dritter Parameter 1 übergeben. Da - nicht kommutativ ist, muss man in der Methode das berücksichtigen und die Operanden tauschen. Ansonsten würde man das falsche Ergebnis bekommen.

Welche Operatoren können überladen werden?

Zu den wichtigsten Operatoren, die überladen werden können, zählen:

- Arithmetische Operatoren: +, +=, -, -=, *, *=, /, /=
- Vergleiche: <, <=, >, >=, ==, !=, <=>, cmp
- Inkrement, Dekrement: ++, --

Auch andere Sachen, die keinen direkten Operator haben, können überladen werden:

- bool
- ""
- 0+

Die komplette Liste gibt es unter `perldoc overload`.

Auto-Generierung

Da schon die hier gezeigte Liste an überladbaren Operatoren relativ lang ist und es ziemlich schwer ist, alle Operatoren zu überladen werden viele Sachen auch automatisch generiert. Das heißt, dass es reicht <=> zu überladen und man bekommt automatisch auch ==, !=, <, <=, > und >= überladen.

Es gibt ein Minimum an Operatoren, die man überladen sollte, wenn man wirklich alles abgedeckt haben möchte:

1. + - * / % ** << >> x
2. <=> cmp
3. & | ^ ~
4. atan2 cos sin exp log sqrt int
5. "" 0+ bool
6. ~~

Bei der Auto-Generierung kann Perl auf verschiedene Werte zurückgreifen. So kann *bool* aus *0+* oder *""* generiert werden. Wenn beides vorhanden ist, wird *0+* herangezogen, weil das für diesen Fall eine höhere Priorität hat. In der Dokumentation `perldoc overload` sind mehrere Tabellen zu finden in denen steht, welche Operation aus welchen Operatoren hergeleitet werden kann. Da das aber je nach überladenen Operatoren etwas andere Ergebnisse liefern kann, sollte man aber prüfen, ob die automatisch generierten Operationen auch genau das tun, was man von ihnen erwartet.

fallback

Dieses eben beschriebene Verhalten der Auto-Generierung kann man aber auch abschalten, in dem man *fallback* auf einen definierten aber unwahren Wert setzt. Wenn *fallback* `undef` ist, ist das Auto-Generieren aktiv. Setzt man einen wahren Wert, so versucht Perl es mit der Auto-Generierung kann aber auch zu dem Verhalten zurückkehren wie es ohne `overload` wäre.

```
package Fallback;

use Moose;

use overload 'x' => 'get_value';

BEGIN {
    if ( !defined $ARGV[0] ) {
        overload->import(
            '""' => 'get_value'
        );
    }

    no warnings 'uninitialized';
    overload->import(
        'fallback' => $ARGV[0]
    );
}

has value => (
    isa => 'Num',
    is  => 'ro',
);

sub get_value { shift->value }

no Moose;

1;
```

Um zu verdeutlichen, wofür *fallback* gut ist, werden hier drei Aufrufe gezeigt. Wenn *!* nicht umgesetzt ist, kann es aus *bool*, *0+* oder *""* generiert werden.

```
$ perl -MFallback -E 'say !Fallback->new' 0
Operation "!": no method found, argument in
  overloaded package Fallback at -e line 1.
```

In diesem Fall ist nur *x* überladen mit der Funktion `get_value`. *fallback* ist auf `0` gesetzt, da das ein definierter aber unwahrer Wert ist, ist das Auto-Generieren ausgeschaltet. Da keine Methode für *!* implementiert ist, wird eine Fehlermeldung geschmissen.

```
$ perl -MFallback -E 'say !Fallback->new' 1

$
```

Auch hier ist nur *x* überladen. *fallback* ist auf `1` gesetzt, also ein definierter und wahrer Wert. Damit ist es Perl erlaubt auf das "Standardverhalten" zurückzugreifen wenn die Operation nicht überladen ist und die Operation nicht durch Auto-Generierung möglich ist. Also wird hier einfach nur überprüft, ob das Objekt existiert.

```
$ perl -MFallback -E 'say !Fallback->new'
1
$
```

Da hier kein Parameter übergeben wird, wird noch *""* überladen und *fallback* ist undefiniert. Dadurch versucht Perl durch Auto-Generierung die Operation zu ermöglichen wenn diese nicht überladen ist. *!* ist nicht überladen, kann aber aus *""* generiert werden. Also wird bei *!* die Funktion `get_value` ausgeführt.

nomethod

Es macht meistens keinen Sinn, alle Operatoren überladen zu wollen. Was sollte das Multiplizieren bei der Magazin-Klasse machen? Da gibt es keinen sinnvollen Anwendungsfall. Es macht auch keinen Sinn, diese Operation zu erlauben. Für diese Fälle kann man bei `overload` festlegen, was gemacht werden soll, wenn ein Operator verwendet wird, der nicht überladen ist. Dazu gibt es `nomethod`.

```
use Carp;
use overload (
    '+' => \&add,
    'nomethod' => sub {
        croak 'no valid operation';
    };
);
```

Damit bricht das Skript ab, wenn z.B. "*" verwendet:

```
$ perl -MMagazin -e 'Magazin->new * 3'
no valid operation at -e line 1
```

Man könnte damit auch eine "catch all" Funktion erstellen:

```
use overload (
  'nomethod' => 'catch_all',
);
```

Dann bekommt die Funktion neben den beiden Operanden auch noch *swap* und die Operation übergeben.

```
$ perl -MMagazin -e
  '$m = Magazin->new; 3 + $m'
```

Resultiert im Funktionsaufruf `catch_all( $m, 3, 1, '+' )`.