Perl-Praxis

Objektorientiertes Programmieren

Jörn Clausen
joern@TechFak.Uni-Bielefeld.DE

Übersicht

- Objektorientierung in Perl
- Klassen, Objekte, Methoden
- Attribute

Warum OO?

- Datenkapselung / *information hiding*
- *name clashes*
- exportierte Namen:

```
use Matrix; use Music;
@flipped = transpose(@matrix);
@higher  = transpose(8, @song);
```

- explizite Modul-Namen:

```
@flipped = Matrix::transpose(@matrix);
@higher  = Music::transpose(8, @song);
```

- objekt-orientiert:

```
$flipped = $matrix->transpose;
$higher  = $song->transpose(8);
```

OO in Perl

- nachträgliche Erweiterung (ähnlich C/C++)
- Erweiterung vorhandener Konstrukte:

Klasse	Modul
Methode	Subroutine
Objekt	Referenz auf Variable

- Objekt „weiß“, zu welchem Modul es gehört
- OO in Perl weniger strikt als anderen Sprachen (C++, Java)

Konstruktoren

- Konstruktor kann beliebigen Namen haben
- Klasse kann mehrere Konstruktoren haben

```
package Shouter;
$VERSION='1.0';

sub new {
  my $obj;
  bless(\$obj);
  return(\$obj);
}
```

- Instanziierung:

```
use Shouter;
$speaker = Shouter->new;
```

```
Shouter=SCALAR(0x140137460)
```

Aufgaben

- Erhält der Konstruktor new irgendwelche Parameter?
  Wenn ja: Welche Daten werden übergeben?

- Die Methode new erhält einen Parameter. Er enthält den Namen der Klasse, in die-
  sem Fall also "Shouter".

Konstruktoren, cont.

- typisches Idiom:

```
sub new {
  my ($class) = @_;
  my $obj;
  bless(\$obj, $class);
  return(\$obj);
}
```

- für Vererbung wichtig
- alternative Syntax für Methodenaufruf:

```
$speaker = new Shouter;
```

parametrisierte Konstruktoren

- Parameter bei der Instanziierung übergeben:

```
$lucy = Shouter->new('Lucy');
$sgt  = Shouter->new('Sgt. Hartman');
```

- „normale“ Parameterübergabe (nach dem Klassennamen):

```
sub new {
  my ($class, $name) = @_;
  my $obj = $name;
  return(bless(\$obj, $class));
}
```

- referenzierte Variable `$obj` bisher unbenutzt
- jetzt Container für Attribut „Name“ des Objekts

Methoden

- Methoden verwenden:

```
$lucy->hello;
$sgt->hello;
```

- Methode definieren:

```
sub hello {
  my ($self) = @_;
  print "$$self says hello\n";
}
```

- `$self` ist das Objekt selbst
- `->` übergibt „Ding“ links von sich als erstes Argument

parametrisierte Methoden

- Paramter an Methode übergeben:

```
$lucy->say('you stupid beagle');
$sgt->say('move it, private pyle!');
```

- Definition:

```
sub say {
  my ($self, $text) = @_;
  print "$$self says: ",uc($text),"\n";
}
```

Destruktoren

- implizite Destruktion bei verlassen des Scopes
- automatische *garbage collection*
- Zerstörung erzwingen:

```
undef($lucy);
```

- eigener Destruktor aber möglich:

```
sub DESTROY {
  my ($self) = @_;
  print "$$self says good bye\n";
}
```

- Datei schließen, Verbindung zu Datenbank abbauen, . . .

Aufgaben

- Protokolliere in der Klasse `Shouter`, wieviele Objekte gerade instanziiert sind. Über die Klassenmethode `count` soll diese Zahl ermittelt werden können.

- Objekte mitzählen:

```
package Shouter;

$VERSION = '1.0';

my $count = 0;

sub new {
    my ($class, $name) = @_;
    my $obj = $name;
    $count++;
    return(bless($obj, $class));
}

sub count {
    return($count);
}

sub DESTROY {
    my ($self) = @_;
    print "$self says good bye\n";
    $count--;
}
```

Attribute

- einzelnes Attribut in Objekt-Variable ablegen
- mehrere Attribute: Hash-Referenz als Objekt

```
$juser = Student->new('Joe User', 'juser', '12345', 'NWI');
```

- Konstruktor:

```
sub new {
  my ($class, $name, $account, $matnum, $subject) = @_;
  my $student = { name    => $name,
                  account => $account,
                  matnum  => $matnum,
                  subject => $subject };
  return(bless($student, $class));
}
```

- beachte: `$student` ist bereits Referenz

Zugriff auf Attribute

- unschön, häßlich, gefährlich, . . . , falsch:

```
$juser = Student->new('Joe User', 'juser', '12345', 'NWI');
$subject = $juser->{subject};
$juser->{subject} = 'MGS';
```

- `->` kein Methodenaufruf sondern Hash-Zugriff
- Verletzung des Prinzips der Datenkapselung
- `perlmodlib(1)`:

  Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.

korrekter Zugriff auf Attribute

- über Methoden:

```
sub getsubject {
  my ($self) = @_;
  return($self->{subject});
}

sub setsubject {
  my ($self, $newsubject) = @_;
  $self->{subject} = $newsubject;
}
```

- Aufruf:

```
$subject = $juser->getsubject;
$juser->setsubject('MGS');
```

getter-setter-Methoden

- Zusammenfassung von `get...` und `set...`:

```
sub subject {
  my ($self, $newsubject) = @_;
  $self->{subject} = $newsubject if defined($newsubject);
  return($self->{subject});
}
```

- Aufruf:

```
$subject = $juser->subject;
$juser->subject('MGS');
```

- Rückgabe des gesetzten Attributs zur Kontrolle

```
$newsubject = $juser->subject('MGS');
```

Aufgaben

- Eine komplexe Zahl $z \in \mathbb{C}$ läßt sich als Paar (α, β) von zwei reellen Zahlen $\alpha, \beta \in \mathbb{R}$ darstellen. Die Länge einer komplexen Zahl ist definiert als

$$|z| = \sqrt{\alpha^2 + \beta^2}$$

  Implementiere eine Klasse CP, um komplexe Zahlen zu verarbeiten. Definieren die folgenden Methoden:

```
$z   = CP->new(3,4); # 3+4i
$re  = $z->Re;       # realer Anteil, 3
$im  = $z->Im;       # imaginaerer Anteil, 4
$len = $z->length;   # Laenge, 5
```

- Komplexe Zahlen:

```
package CP;
$VERSION = '1.0';

sub new {
    my ($class, $re, $im) = @_;
    my $c = { re => $re,
              im => $im };
    return(bless($c, $class));
}

sub Re {
    my ($self) = @_;
    return($self->{re});
}

sub Im {
    my ($self) = @_;
    return($self->{im});
}

sub length {
    my ($self) = @_;
    my $re = $self->Re;  # $self->{re}
    my $im = $self->Im;  # $self->{im}
    return(sqrt($re**2 + $im**2));
}
```

Aufgaben, cont.

- Addition und Multiplikation von komplexen Zahlen sind folgendermaßen definiert:

$$\begin{aligned}(\alpha_1, \beta_1) + (\alpha_2, \beta_2) &= (\alpha_1 + \alpha_2, \beta_1 + \beta_2) \\ (\alpha_1, \beta_1) \cdot (\alpha_2, \beta_2) &= (\alpha_1\alpha_2 - \beta_1\beta_2, \alpha_1\beta_2 + \alpha_2\beta_1)\end{aligned}$$

  Implementiere entsprechende Methoden:

```
$s = $z->add($y);
$p = $z->mul($y);
$v = $s->add($p)->mul($z);
```

  Beachte, daß sowohl der übergebene Paramter $y als auch die Rückgabewerte $s bzw. $p Objekte vom Typ CP sind.

- Addition und Multiplikation:

```
sub add {
    my ($self, $other) = @_;
    return(CP->new($self->re + $other->re, $self->im + $other->im));
}

sub mul {
    my ($self, $other) = @_;
    my ($a1,$b1) = ($self->re, $self->im);
    my ($a2,$b2) = ($other->re, $other->im);
    return(CP->new($a1*$a2 - $b1*$b2, $a1*$b2 + $a2*$b1));
}
```

Projekt

- Implementiere eine einfache Stundenplanverwaltung mit den beiden Klassen `Student` und `Course`. Die folgenden Attribute sollen verwaltet werden:

`Student`

`name`	Joe User
`account`	juser
`matnum`	12345
`subject`	NWI
`grade`	GS

`Course`

`title`	Perl-Praxis
`time`	2
`type`	Ü
`grade`	GS

Projekt, cont.

- Die folgenden Methoden sollen implementiert werden:

```
$juser = Student->new('Joe User', 'juser', 12345,
                      'NWI', 'GS');
$perl = Course->new('Perl-Praxis', 2, 'Ü', 'GS');

$juser->enroll($perl);    # Berechtigung pruefen
$juser->totaltime;        # Gesamt-SWS
$juser->timetable;        # tabellarischer Stundenplan
$perl->as_text;           # Informationen zu Veranstaltung

foreach $course ($juser->courses) {
  print $course->as_text;
}
```


- An irgendeiner Stelle muß protokolliert werden, in welche Kurse sich ein Student erfolgreich eingeschrieben hat. Dazu legt man ein weiteres Attribut an, das auf eine Liste von Course-Objekten zeigt. Im Konstruktor kann man dieses Attribut mit einer leeren Liste initialisieren:

```
my $stud = { name => $name,
             acc => $acc,
             ...
             courses => [] };
```

In der enroll-Methode wird das jeweilige Kursobjekt dann in dieser Liste abgelegt:

```
push(@{self->{courses}}, $course);
```