Perl

- Skriptsprache
- Wenig Code für einfache Aufgaben
- Sehr viele String-Funktionen
- Keine Typsicherheit, keine Deklarationen notwendig
- Fast immer mehr als ein Weg, eine Aufgabe zu erledigen

Perl-Skript

```
#!/usr/bin/perl

# die folgende Zeile gibt "Hallo Welt" aus
print "Hallo Welt\n";
```

Skript starten

```
perl versuch.pl

perl -w versuch.pl

./versuch.pl
```

Datentypen

Skalar: $scalar
Array: @array
Hash: %hash
Für Arrays und Hashes keine vorherige Initialisierung mit fester (oder variabler) Größe notwendig
my $var: Geltungsbereich auf aktuellen Block beschränken

Skalare

```
$hallo = "Welt";

$wert = 1;

$doublewert = 6.023e23;

$wert2 = $wert1;
```

Arrays

```
@array = (1, 2, 3, 4);

@array = ("Hallo", " ", "Welt");

@array = (1, "Hallo", 1.5);


$array[0] = "Hallo";

$array[10] = "Welt";


$wert = $array[7]
```

Hashes

```
%hash = ("Hallo" => 1, "Welt" => 2);

%hash = ("Hallo" => "Welt", "Welt" => "weit");


$hash{"Hallo"} = "Welt";

$hash{"Welt"} = 5;


$hash{"Hallo"} = $hash{"Welt"} * 0.43;
```

Besonderheiten

Besondere Variablen:

$0 # *Name des Programms*

@ARGV # *Argumente des Programms*

$_ # *oft aktueller Wert einer (Schleifen–) Anweisung*

Länge eines Arrays:

$length = @array;

Index des letzten Elements eines Arrays:

$last = $#array;

Operatoren

Wie üblich:

+, *, /, -, %

Potenz: **
Konkatenation: .
String-Multiplikation: x
Mit Zuweisung:
+=, *=, /=, -=, .=, x=
Inkrement/Dekrement:
++$a, $a++, --$a, $a--

Operatoren (2)

Logische Operatoren:
&&, ||, !
and, or, not, xor
Vergleich:
==, !=, <, >, <=, >=
$a <=> $b

$a < $b: −1
$a > $b: 1
$a == $b: 0

Bereichsoperator (..)

@array = @array[1 .. 5];

@array = @array[−5 .. −1] # *letzte 5 Elemente*

@array = @array[10 .. $#array] # *10. bis letztes Element*

„magisches“ Autoinkrement:

@alph = (”A” .. ”Z”);

@twosym = (”AA” .. ”ZZ”);

Kontrollstrukturen

```
if (...){
      ...
} elsif (...){
 ...
}else{
 ...
}

unless (...){
 ...
}
```

Kontrollstrukturen (2)

```
while (...){
 ...
}

until (...){
 ...
}

for($i=0;$i<100;$i++){
 ...
}

foreach $value (@array){

}
```

Dateihandles

STDIN, STDOUT, STDERR

open(SESAM, "<file") # *lesen*

open(SESAM, ">file") # *schreiben*

open(SESAM, ">>file") # *anhaengen*

Testen ob Datei existiert:
if (−e /pfad/zur/datei){ ... }

open(SESAM, ">file") or **die** "Could␣not␣open␣file!\n";

Lesen

```
while(<SESAM>){
    print $_;
}

while(<SESAM>){
    print;
}

while(my $line = <SESAM>){
print $line;
}


if( defined( $line = <SESAM> ) ){
    print $line;
}
```

Schreiben

```
open(SESAM,">file"); #oder
open(SESAM,">>file");

print  SESAM "Hallo␣Welt!\n";

print  STDOUT "Hallo␣Welt"\n";
```

Pattern matching

Konkatenation

"Hallo" . " " . "Welt"

$a.$b

"$a␣$b"

"The␣content␣of␣a␣is␣$a"

Substrings

```
substr($str, 10, 15) # ab Pos. 10, L"ange 15
substr($str, 10) # ab Pos. 10, bis Ende
substr($str, -10) # letzte 10 Zeichen

substr($str, 0, 0) = "Hallo" # an Anfang
substr($str, 0, 1) = "Hallo" # erstes Zeichen ersetzt
substr($str, -1) = "Hallo" # letztes Zeichen ersetzt


chomp($str) # entfernt newlines am Ende
```

Reguläre Ausdrücke

\	Escape	\n
|	Alternative	a|b
( )	Gruppierung	(a|b)c
–	Bereiche	A–Za–z
[ ]	Zeichenklasse	[ACGT]
^	Anfang	^(Hallo | Welt)
$	Ende	(the | end)$
.	beliebiges Zeichen	(^.$)

Quantifikatoren

$*$	0 - n mal
$+$	1 - n mal
?	0 - 1 mal
{n}	n mal
{n,}	mindestens n mal
{n,k}	n bis k mal

Normalerweise maximal gematcht, mit nachgestellten ? minimal

Match

m/pattern/
/pattern/

Skalerer Kontext:

$str =~ **m**/pattern/
 # gibt wahr (1) oder falsch ("") zur"uck

if ($str =~ **m**/pattern/){ ... }

Listenkontext:

@array = $str =~ **m**/pattern/g *# alle matches in Liste*

Modifier:
i: case-insensitive
g: globale Suche, progressive Suche

Match (2)

Progressive Suche:

while($str =~ **m**/pattern/ig){ ... }

Besondere Funktionen und Variablen:
pos($str): Position des Zeichens nach aktuellem match
length($str): Länge von $str
$&: aktueller match
$‘: Zeichen davor
$’: Zeichen dahinter
pos($str) == **length**($‘) + **length**($&)

Substitution

```
s///

$str =~ s/pattern/replacement/;

$str =~ s/pattern/replacement/gi;


$str =~ s/pattern/\u$&/g;
($strtemp = $str) =~ s/pattern/replacement/g;

for $_ (@lines) { $_ =~ s/pattern/replacement/g }
for (@lines) { s/pattern/replacement/g }
```

Capturing

```
while( $line =~ m/((.*):(.*)) ){
print "$1 => $2, $3";
}

$line =~ s/(.*)-(.*)/-$2+$1/g;

$line = 1234567890;
while ( $line =~ s/([0-9]+)([0-9]{3})($| )/$1 $2$3/g){print "$line\n"; }

1234567 890
1234 567 890
1 234 567 890

while ( $line =~ s/([0-9]+)([0-9]{3})($| )/$1 $2$3/g){}

1 while ( $line =~ s/([0-9]+)([0-9]{3})($| )/$1 $2$3/g);
```

Übersetzung

tr///

$str =~ **tr**/ list / list /

$str =~ **tr**/A−Z/a−z/;

Achtung: Keine Regulären Ausdrücke, keine Muster, sondern Listen!

Beispiele

```
$str = "ACATGATAGGCGTATA";

if( $str =~ m/((TA){2}|(TA(C|G)+))/){
        print $&."\n";
}
```

Beispiele

```
$str = "ACATGATAGGCGTATA";

if( $str =~ m/((TA){2}|(TA(C|G)+))/){
        print $&."\n";
}
```

Ausgabe:
TAGGCG

Beispiele (2)

```
$str = "ACATGATAGGCGTATA";

if( $str =~ m/((TA){2}|(TA(C|G)+?))/){
        print $&."\n";
}
```

Beispiele (2)

```
$str = "ACATGATAGGCGTATA";

if( $str =~ m/((TA){2}|(TA(C|G)+?))/){
        print $&."\n";
}
```

Ausgabe:
TAG

Beispiele (3)

```
$str = "ACATGATATATA";

if( $str =~ m/((TA){2}|(TA(C|G)+?))/){
        print $&."\n";
}
```

Beispiele (3)

```
$str = "ACATGATATATA";

if( $str =~ m/((TA){2}|(TA(C|G)+?))/){
        print $&."\n";
}
```

Ausgabe:
TATA

Beispiele (4)

```
$str = "ACATGATAGGCGTATA";

while( $str =~ m/((TA){2}|(TA(C|G)+))/g){
        print $&."\n";
}
```

Beispiele (4)

```
$str = "ACATGATAGGCGTATA";

while( $str =~ m/((TA){2}|(TA(C|G)+))/g){
        print $&."\n";
}
```

Ausgabe:
TAGGCG
TATA

Beispiele (5)

```
$str = "ACATGATAGGCGTATA";

$str =~ s/A/T/g;
$str =~ s/C/G/g;
$str =~ s/G/C/g;
$str =~ s/T/A/g;

print $str . "\n";
```

Beispiele (5)

```
$str = "ACATGATAGGCGTATA";

$str =~ s/A/T/g;
$str =~ s/C/G/g;
$str =~ s/G/C/g;
$str =~ s/T/A/g;

print $str . "\n";
```

Ausgabe:
ACAACAAACCCCAAAA

Beispiele (6)

```
$str = "ACATGATAGGCGTATA";

$str =~ tr/ACGT/TGCA/;

print $str . "\n";
```

Beispiele (6)

```
$str = "ACATGATAGGCGTATA";

$str =~ tr/ACGT/TGCA/;

print $str . "\n";
```

Ausgabe:
TGTACTATCCGCATAT

Subroutinen

. . . oder Funktionen oder Prozeduren oder . . .

Subroutinen

Definition:

```
sub routine{
 ...  # auszufuehrender Code
}
```

Aufruf:

```
routine ();
```

Argumente

- alle Argumente in Array @_ gespeichert
- keine Typsicherheit (Perl!)
- keine feste Anzahl
- Ausweg 1: (teilweise) Prototypen (aber nicht in dieser Vorlesung)
- Ausweg 2: Gute und vollständige Dokumentation der erwarteten Parameter!

Zugriff auf Elemente von @_ wie bei jedem anderen Array ($_[0]).

Tricks mit Argumenten

```
sub setHash{
    my ($key, $value) = @_;
    $hash{$key} = $value;
}

sub peek{
    my $next = shift # oder shift(@_)
    return $next;
}

$val = peek(@array);
$val = peek(1,2,3,4);

sub createHash{
    my %hash = @_;
    return %hash;
}

%here = createHash("a" => 1, "b" => 2);
```

Referenzen

Referenzen

Symbolische Referenzen:

```
$name = "count";
$$name = 1; # Zugriff auf $count
$name = "count2";
@$name = ( 2, 5, 7 ); # Array @count2
$count3 = $$name[2]; # Zugriff auf $count2[2]
```

Harte Referenzen:

```
$name = \$count;
$$name = 1; # Zugriff auf $count
$name = \@count2;
@$name = ( 2, 5, 7); # Array @count2
$count3 = $$name[2]; # Zugriff auf $count2[2]
```

Uns interessieren in erster Linie harte Referenzen.

Harte Referenzen

Erzeugen von Referenzen:

```
$scalarref = \$scalar;
$arrayref = \@array;
$hashref = \%hash;
```

Zugriff:

```
$$scalarref = 5;
@$arrayref = ( 2, 5, 7 );
$$hashref{"bla"} = "blub";

$$arrayref [2] = 3; # entspricht
${ $arrayref }[2] = 3; # und nicht
${ $arrayref [2]} = 3;
```

Pfeil-Operator

Automatische Dereferenzierung vor Zugriff:

$$arrayref [2] = 3 # *entspricht*
$arrayref ->[2] = 3;

$$hashref{"bla"} = "blub"; *#entspricht*
$hashref->{"bla"} = "blub";

Benötigen wir später für Zugriff auf Methoden von Objektreferenzen

Anonymität

Anonyme Arrays:

```
$arrayref = [ 2, 5, 7];
$arrayref->[2] = 3;
```

Anonyme Hashes:

```
$hashref = { "bla" => "blub", "Hallo" => "Welt"};
$hashref->{"bla"} = 3;
```

Zugriff nur noch über Referenz.
Objekte in Perl sind oft Referenzen auf anonyme Hashes.

Pakete

Pakete

- Pakete definieren Namesräume für Variablen
- Mehrere Pakete pro Datei
- Mehrere Dateien für ein Paket
- Ein Paket pro Datei und eine Datei für jedes Paket (für uns am relevantesten)
  $\Rightarrow$ Module
- Durch Pakete Wiederverwenden von Code möglich
  - Klare Trennung zwischen lokal definierten Variablen und denen des Paketes
  - Variablen des Paketes können nicht aus Versehen verändert werden

Deklaration

Deklaration des Beginns eines Paketes mit

package paketname;

Beispiele:

package Color;

package Color::Red;

- Deklaration an beliebiger Stelle im Code
- Geltungsbereich bis
  - zur nächsten Paket-Deklaration
  - Ende des einschließenden Blocks
  - Ende der Datei

main-Paket

Im Paket main liegen alle Variablen, außerhalb anderer Pakete und

- $_
- STDIN
- STDOUT
- STDERR
- ARGV
- ...

Das main-Paket enthält alle anderen Pakete und sich selbst:

main::Color::Red # *entspricht*
Color::Red

main::main::main # *entspricht*
main::main # *entspricht*
main

Zugriff

Eine Variable $var, die im Paket Color::Red deklariert wurde, erreicht man (in anderen Paketen) mit

$Color::Red::var

Für Variablen im main-Paket gilt

$main::var # *entspricht*
$::var # *entspricht*
$var

Module

Module

Module sind Pakete, die

- aus genau einer Datei bestehen
- deren Datei auf .pm (für Perl-Modul) endet
- deren Datei genauso heißt, wie das Paket
- :: entspricht Pfad-Trennzeichen

Beispiel:
Datei ./Color/Red.pm

```
package Color::Red;

our $VERSION = 1.0; # Versionsnummer

...
```

Module (2)

Benutzung von Modulen:

use Color :: Red; # *ohne .pm*

- Das Modul Color :: Red muss in einer Datei Color/Red.pm definiert sein,
- das Verzeichnis Color muss im aktuellen Pfad (@INC) von Perl liegen
- *Der Pfad kann über die Umgebungsvariable PERL5LIB geändert werden.*
- das aktuelle Verzeichnis (.) ist (normalerweise) in @INC enthalten

Klassen und Objekte

Klassen

Eine **Klasse**
definiert die Eigenschaften (Instanzvariablen oder Felder) und das Verhalten (Methoden) von (gleichartigen) Objekten.

In Perl sind Klassen als Pakete oder Module umgesetzt. Diese Module definieren Subroutinen, um Objekte einer Klasse zu erzeugen (Konstruktoren) und um auf Objekten der Klasse zu arbeiten (Methoden).

Objekte

Ein **Objekt**
ist eine Instanz (Ausprägung) einer Klasse.

In Perl sind Objekte Datenstrukturen (häufig Hashes aber auch Arrays, ...) auf die eine Referenz verweist. Auǧerdem „wissen“ Objekte zu welcher Klasse sie gehören. Man arbeitet in Perl nur auf der Referenz auf das Objekt und greift mit dem Pfeil-Operator auf Eigenschaften und Methoden zu.

In **Instanzvariablen**
sind Eigenschaften von Objekten gespeichert.

In Perl gibt es keine Instanzvariablen im eigentlichen (üblichen) Sinn. Objekteigenschaften werden stattdessen als Werte innerhalb des Objekt-Hashes oder -Arrays dargestellt.

Methoden und Konstrukoren

Eine **Methode**
ist eine Funktion oder Prozedur, die auf Objekten arbeitet, um Eigenschaften abzufragen oder zu setzen, Objekte zu manipulieren oder Berechnungen auf diesen auszuführen.

In Perl sind Methoden Subroutinen innerhalb einer Klassendefinition. Methoden erwarten als erstes Argument ein Objekt der Klasse in der sie definiert wurden oder die Klasse selbst (oder eine Subklasse).

Ein **Konstruktor**
ist eine spezielle Methode, die ein Objekt (eine Instanz) einer Klasse erzeugt und zurückgibt.

In Perl sind Konstruktoren genau so implementiert. Es gibt keine spezielle Syntax oder spezielle Keywords für Konstruktoren.

Methoden

Erstes Argument jeder Methode ist Referenz auf aufrufendes Objekt oder aufrufende Klasse.
$\Rightarrow$ keine Unterscheidung (syntaktisch) zwischen Klassen- und Instanzmethoden.

```
sub getName{
    my $caller = shift;
    ref( $caller ) or die "Not a class-method\n";
    my $name = $caller->{"name"};
    return $name;
}

$obj->getName();

$numInstances=0;
sub getNumInstances{
    return $numInstances;
}

Klasse->getNumInstances();
```

Objekteigenschaften

Getter- und Setter-Methoden:

```
sub getName{
    $caller = shift;
    ref( $caller ) or die "No object\n";
    return $caller->{"name"};
}

sub setName{
    $caller = shift;
    ref( $caller ) or die "No object\n";
    $caller->{"name"} = shift;
}
```

Objekteigenschaften (2)

Gemeinsame Methode für Lesen und Setzen:

```
sub name{
    $caller = shift;
    ref( $caller ) or die "No object\n";
    if (@_){ $caller->{"name"} = shift; }
    return $caller->{"name"};
}
```

Konstruktoren

Ein **Konstruktor**
ist in Perl eine Methode, die eine Objektreferenz zurückgibt.

```
sub create{
    my $self = {} # Referenz auf anonymes Hash
    bless( $self ,"Color"); # Hash wird zu Objekt
                            # der Klasse Color
    return $self ;
}
```

Die Funktion **bless** erwartet als erstes Argument eine Referenz und als zweites optionales Argument die Klasse zu der das Objekt gehören soll. Wird das zweite Argument weggelassen, wird das Objekt in das aktuelle Paket "gesegnet".

Konstruktoren (2)

Standard-Konstruktor mit Argumenten:

```
sub define{
    my $caller = shift;
    my $class = ref( $caller ) || $caller ;
    my $self = { @_ };
    bless( $self , $class );
    return $self ;
}

$red = Color->define(name => "red", rgb => "255,0,0");
```

Vererbung

- Vererbung über spezielles Array @ISA
- Mehrfachvererbung möglich (darum ein Array)

package Mule;
our @ISA = ("Horse", "Donkey");

Suche nach einer Methode (dfs):

1. Suche in aktueller Klasse (Mule)
2. Suche in erster Oberklasse (Horse)
3. Suche der Reihe nach in den Oberklassen von Horse (wenn es sie gibt)
4. Suche in zweiter Oberklasse (Donkey)
5. ...

Methoden der Oberklasse

- durch @ISA werden Oberklassen definiert
- um Methoden der Oberklassen in der aktuellen Klasse zu nutzen: **require**

```
package Mule;
our @ISA = ("Horse", "Donkey");
require Horse;
require Donkey;
```

Gleiche Funktion, aber kürzer (und einfacher):

```
package Mule;
use base ("Horse", "Donkey");
```

Methoden der Oberklasse (2)

Zugriff innerhalb einer Methode (um Code wiederzuverwenden):

```
package Mule;
use base ("Horse", "Donkey");

sub speak{
    my $caller = shift;
     $caller->Horse::speak();
     ...
}
```

Häufig unbekannt, in welcher Oberklasse Methode implementiert ist:

```
package Mule;
use base ("Horse", "Donkey");

sub speak{
    my $caller = shift;
     $caller->SUPER::speak();
     ...
}
```

Interfaces

. . . gibt es in Perl nicht.

Interfaces

. . . gibt es in Perl nicht.
. . . aber können durch Mehrfachvererbung „emuliert“ werden:

- Gemeinsame Oberklasse mehrerer Klasse
- Oberklasse definiert Methoden
  - können („müssen“ hier sogar) überladen werden
  - sind nicht (inhaltlich) implementiert
  - Klassenvertrag durch Dokumentation

Bioperl

. . . endlich . . .

Bioperl

Sammlung von Modulen und Klassen zum

- Zugriff auf Sequenzen
  - aus lokalen Dateien
  - lokalen und entfernten Datenbanken
- Format-Umwandlungen
- Manipulation von Sequenzen
- Suche nach Sequenzähnlichkeiten (BLAST,...)
- Erstellen von Alignments
- ...

Sequenz-Klassen

- Bio::Seq: Standard-Sequenz
- Bio::PrimarySeq: „abgespeckte“ Variante von Bio::Seq
- Bio::Seq::RichSeq: speziell für „ausladende“ Annotation
- Bio::Seq::LargeSeq: speziell für lange ($>$ 100 MB) Sequenzen
- Bio::LiveSeq::: Paket für häufig geänderte Sequenzen

Abstraktion durch Interfaces Bio::PrimarySeqI und Bio::SeqI
$\Rightarrow$ genutzte Implementierung oft nicht relevant

Sequenz-I/O

Klasse SeqIO zum Lesen und Schreiben von

- FastA
- EMBL
- GenBank
- Swissprot
- raw
- ...

Eingelesene Sequenzen je nach Format als Seq, PrimarySeq, RichSeq repräsentiert.

Sequenzdatenbanken

Paket Bio::DB:: enthält Klassen, um Verbindungen zu (Sequenz-) Datenbanken aufzubauen und Sequenzen nach

- Accession number
- Name
- Version
- ID
- ...

abzurufen und weiter zu verwenden.

Bio::Perl

Bio :: Perl enthält Subroutinen, um einfache Operationen auf Sequenzen (auch ohne tieferes Wissen über Objekte) zu ermöglichen.
Subroutinen z.B.

- get_sequence
- new_sequence
- revcom
- translate
- write_sequence

Dokumentation

www.bioperl.org

- BioPerl Tutorial
- weitere (speziellere) Tutorials
- HowTos
- Deobfuscator (ähnlich Javadoc)
  - Liste aller verfügbaren Klassen
  - Liste aller Methoden (inkl. ererbten) pro Klasse
  - Dokumentation/Kommentare zu Klassen, Methoden

Bio::Seq

Konstruktor für neue Sequenzen:

Bio::Seq->new(<args>);

Argumente:

- -seq: die Sequenz
- -alphabet: dna, rna oder protein
- -description: Beschreibung zur Sequenz
- -accession_number: die Nummer der Sequenz

...und weitere mögliche Argumente.
Beispiel:

$seq = Bio::Seq->new(-seq => 'ACGT', -alphabet => 'dna');

Bio::Seq (2)

Methoden:

- seq(): Sequenz als String
- **length**(): Länge der Sequenz
- subseq(start ,end): Subsequenz von start (ab 1) bis end (inklusive) (String)
- revcom(): Das reverse Komplement als Bio::Seq-Objekt
- translate (): Proteinsequenz (Bio::Seq) aus Translation, optionale Argumente:
  - −complete: Translatierte Sequenz inklusive Start-, Stop-Codon (0 oder 1)
  - −orf: Translationsstart ab erstem Start-Codon (0 oder 1)

  Beispiel: $prot = $seq−>translate(−complete => 1, −orf => 1);

Bio::SeqIO

Konstruktor:

Bio::SeqIO->new(<args>)

Argumente:

- -file: Pfad zur Datei, wie bei **open**()
- -**format**: Dateiformat:
  - Genbank
  - Fasta
  - EMBL
  - ...

Bio::SeqIO (2)

Konstruktor für Filehandle:

Bio::SeqIO->newFh(<args>)

- Argumente wie bei Bio::SeqIO->new().
- Erzeugt Filehandle.

Beispiel:

```
$in = Bio::SeqIO->newFh(-file => 'file.fasta', -format => 'Fasta');
while(my $seq = <$in>){
    print $seq->seq()."\n";
}
```

Bio::SeqIO (3)

Methoden (Objekte mit Bio::SeqIO->new() erzeugt):

- next_seq(): Gibt nächste Sequenz zurück (Bio::Seq)
- write_seq(Bio::Seq): Schreibt Sequenz (Bio::Seq) in angegebene Datei

Beispiele:

```
$in = Bio::SeqIO->new(-file => "Beispiel.fasta", -format => "Fasta");
$seq = $in->next_seq();

$seq2 = Bio::Seq->new(-seq => 'ACGT', -alphabet => 'dna');
$out = Bio::SeqIO->new(-file => ">out.fasta", -format => "Fasta");
$out->write_seq($seq2);
```

Bio::Tools:SeqStats

Klasse für Sequenzstatistiken.
Konstruktor:

Bio :: Tools :: SeqStats−>new(−seq => $seq)

Methoden:

- count_monomers(): Anzahl der Monomere (Hash-Referenz)
- count_codons(): Anzahl der Codons (Hash-Referenz)
- get_mol_wt(): Molekulares Gewicht der Sequenz (Einzelstrang)
  (Array-Referenz, [0]: Untere Schranke, [1]: Obere Schranke)

Bio::DB::GenBank

- Zugriff auf Sequenzen der NCBI GenBank
  `http://www.ncbi.nlm.nih.gov/Genbank/index.html`
- Hier beispielhaft für Zugriffe auf Sequenzdatenbanken

Konstruktor:

$gb = Bio::DB::GenBank->new();

Erstellt neuen „Zugang“ zur NCBI-Datenbank, holt noch keine Daten

Bio::DB::GenBank (2)

Zugriff auf einzelne Sequenzen

- mit bekanntem Namen (ID)
- mit bekannter GI-Nummer
- mit bekannter Accession-Number
- ...

Methoden:

```
$gb->get_Seq_by_id(<id>);
$gb->get_Seq_by_gi(<gi>);
$gb->get_Seq_by_acc(<accession>);
 ...
```

. . . geben Bio::SeqI-Objekte zurück

Bio::DB::Query::GenBank

- Klasse für Anfragen an die NCBI GenBank
- wiederum exemplarisch

Konstruktor:

$query = Bio::DB::Query::GenBank->new();

Argumente:

- -db : 'protein' oder 'nucleotide'
- -query : die Anfrage (wie man sie im Suchfeld bei NCBI eingeben würde)
- -mindate : minimales Einstelldatum
- -maxdate : maximales Einstelldatum
  Alternativ:
- -ids : Liste von GIs, überschreibt query

Query kann in Bio::DB::GenBank::get_Stream_by_query() benutzt werden.

Bio::DB::Query::GenBank (2)

Methoden:

$query->count(); *#Anzahl der Ergebnisse der Anfrage*
$query->ids(); *#Liste von GIs als Ergebnis der Anfrage*

Die GIs können in den Methoden von Bio::DB::GenBank benutzt werden, z.B.

```
$query = Bio::DB::Query::GenBank->new(-db => 'protein',
                -query => 'coli chromosome');
@ids = $query->ids();
$gb->Bio::DB::GenBank->new();
for $id (@ids){
    $gb->get_Seq_by_gi($i);
}
```

Bio::DB::GenBank (revisited)

Zugriff auf Mengen von Sequenzen

```
$gb->get_Stream_by_query(<query>); #Query-Objekte von letzter Folie
$gb->get_Stream_by_gi([<gi1>,<gi2>,...]);
$gb->get_Stream_by_acc([<acc1>,<acc2>,...]);
 ...
```

. . . geben Bio::SeqIO-Objekte zurück,
Zugriff auf einzelne Sequenzen mit next_seq()

Bio::SeqFeatureI

- Interface für Sequenzfeatures (Coding Sequences, repetitive Elemente,...)
- Objekte werden von Bio::Seq::get_SeqFeatures() zurückgegeben

Methoden:

```
$f->primary_tag(); #Tag des Features (z.B. 'CDS', 'exon', ...)
$f->display_id(); #Name des Features zur Anzeige
$f->seq(); #Sequenz(abschnitt) fuer dieses Feature (Bio::PrimarySeq)
$f->spliced_seq() #ggf. gespleisste Sequenz, sonst wahrsch. Originalseq.
$f->location() #Ort auf der Sequenz (Bio::LocationI, naechste Folie)
$f->get_SeqFeatures() #Liste von Subfeatures (wenn vorhanden)
$f->strand() #1 fuer den aktuellen und -1 fuer den Gegenstrang
$f->annotation()#Annotationen (uebernaechste Folie)
```

Bio::LocationI

- Interface für Start- und Endpositionen auf einer Sequenz
- Objekte werden von Bio::SeqFeatureI::location() zurückgegeben

Methoden:

$loc->start() *#Starposition auf der Sequenz*
$loc->end() *#Endposition auf der Sequenz*

...und weitere Methoden in abgeleiteten Klassen.
Methoden in abgeleiteten Klassen können deutlich mehr Eigenschaften beinhalten!

Bio::AnnotationCollectionI

- Annotationen zu einem Feature
- Objekte von Bio::SeqFeatureI::annotation() zurückgegeben

Methoden:

$ac->get_num_of_annotations() *#Anzahl der Annotationen*
$ac->get_all_annotation_keys() *#Schluessel (Namen) aller Annotationen*
$ac->get_Annotations(<key>) *#Liste aller Annotation zu einem Schluessel*

Annotationsobjekte → nächste Folie

Bio::AnnotationI

- Interface für Annotationen (Genzugehörigkeit, Anmerkungen, ähnliche Gene,...)
- Objekte von Bio :: AnnotationCollectionI :: get_Annotations() und Bio :: SeqFeatureI :: get_Annotations() zurückgegeben

Methoden:

$annot->tagname() *#Tag der Annotation (entspricht key)*
$annot->as_text() *#textuelle Beschreibung des Wertes*

Bio::Tools::Run::RemoteBlast

Klasse, mit der man

- NCBI BLAST-Anfragen schicken
- die BLAST-Ergebnisse abholen

kann.
Die Anfragen werden über das Netz an den dortigen BLAST-Server geschickt und dort auch bearbeitet.
Achtung: Skripte möglichst selten testen, um den BLAST-Server nicht unnötig zu belasten.

Bio::Tools::Run::RemoteBlast (2)

Konstruktor:

$blast = Bio::Tools::Run::RemoteBlast->new();

Argumente:

- -prog: BLAST-Version, 'blastn', 'blastp', ...
- -data: Datenbank, 'swissprot', ...
- -expect: E-Wert (Parameter von BLAST), je höher, desto höher auch die Wahrscheinlichkeit, eine entsprechende Sequenzähnlichkeit auch zufällig zu finden
- -readmethod: 'SearchIO' oder 'BPlite'

SearchIO ist die modernere Variante, die mehr Möglichkeiten bietet, BPlite ist einfacher (sowohl bzgl. der Möglichkeiten als auch der Bedienung)

Bio::Tools::Run::RemoteBlast (3)

Methoden:

```
$blast->submit_blast($seq) #$seq ist Bio::SeqI-Objekt
$blast->submit_blast(@seqs) #mehrere Sequenzen
$blast->submit_blast($file) #FastA-File

@rids = $blast->each_rid() #rid: RemoteBlast ID

$blast->retrieve_blast($rid)
```

retrieve_blast () gibt zurück

- -1 bei Fehler
- 0 wenn noch nicht fertig
- ein Bio::SearchIO- oder ein Bio::Tools::BPlite-Objekt

Bio::SearchIO

Moderne Klasse für das Parsen von BLAST-Ergebnissen Konstruktor (nur für lokale Files, sonst über
Bio :: Tools :: Run::RemoteBlast:: retrieve_blast ()):

Bio :: SearchIO−>new();

Argumente:

- -file: Ergebnisfile von BLAST
- -format: 'blast', 'blastxml', 'fasta', ...

Zwei wichtige Methoden:

$searchio−>result_count() # *Anzahl der Ergebnisse*
$searchio−>next_result() # *naechstes Ergebnis*

next_result () gibt Objekt vom Typ Bio::Search::Result::ResultI zurück

Bio::Search::Result::ResultI

Interface für einzelnes BLAST-Ergebnis.
Methoden:

$result->algorithm() # *verwendeter Algorithmus (blastn, blastp, ...)*
$result->num_hits() # *Anzahl Treffer*
$result->next_hit() # *naechster Treffer*

next_hit () gibt ein Objekt des Interfaces Bio::Search::Hit::HitI zurück

Bio::Search::Hit::HitI

Interface für einzelne Hits. Methoden:

$hit->name(); *#Name als Skalar*
$hit->description(); *#Beschreibung (Skalar)*
$hit->**length**(); *#Laenge (Skalar)*
$hit->significance (); *#Signifikanz als E-Wert*
$hit->next_hsp(); *#Naechstes High Scoring Pair*

High Scoring Pairs vom Typ Bio::Search::HSP::HSPI

Bio::Tools::BPlite

Ältere und einfachere Alternative zu Bio::SearchIO. Methoden:

```
$lite->database(); # verwendete Datenbank
$lite->nextSbjct(); # naechstes 'Subject'
$lite->next_feature(); # naechtes High Scoring Pair
```

High Scoring Pair als Objekt vom Typ Bio::Tools::BPlite::HSP,
nextSbjct() gibt Objekt vom Typ Bio::Tools::BPlite::Sbjct zurück.

Bio::Tools::BPlite::Sbjct

wenige Methoden:

$sbjct->name(); # *Name des Treffers*
$sbjct->nextHSP(); # *Naechstes High Scoring Pair*

name() liefert z.B. für blastp den Namen des gefundenen Proteins.
nextHSP() liefert Objekt vom Typ Bio::Tools::BPlite::HSP.

Bio::Restriction::EnzymeCollection

Klasse für eine Menge von Restriktionsenzymen.
Konstruktor:

$ec = Bio::Restriction::EnzymeCollection->new();

Legt neue Menge mit allen Enzymen an, die BioPerl kennt.
Methoden:

$ec->blunt_enzyme(); *#Liste der Enzyme die glatt schneiden*
$ec->each_enzyme(); *#Liste aller Enzyme*
$ec->get_enzyme($name); *#Enzym mit bestimmtem Namen*

get_enzyme() liefert ein Bio::Restriction::Enzyme-Objekt zurück

Bio::Restriction::Analysis

Klasse für die Analyse von Restriktionsschnittstellen.
Konstruktor:

$an = Bio:: Restriction :: Analysis->new();

Argumente:

- -seq: Bio::Seqlnfo-Objekt das geschnitten werden soll
- -enzymes: (optional) Bio::Restriction::EnzymeCollection-Objekt

Methoden:

$an->fragments($enz)

Berechnet die Fragmente fuer ein bestimmtes Enzym,
Rueckgabe ist Liste von Skalaren (Strings)

$an->sizes($enz) *#Groesse der Fragmente*

The End