Perl によるオブジェクト指向プログラミング

この講義の目的

- 明日以降、Perlの言語自体にはまらない
    - 今日、いろいろやって、なるべくはまってください
    - 疑問があったらどんどん質問してください

目次

Perlプログラミング勘所

質問

- Perlでプログラミングをしたことがありますか?

はじめに

- 事前課題
    - http://github.com/hakobe/Sorter
- 前提
    - はじめてのPerl、続はじめてのPerlに目を通している
    - 一度はPerlでオブジェクト指向プログラミングしたことがある
    - 事前課題でやっているはず

Perlの良いところ

- CPAN
    - やりたいことはすでにモジュール化されてる
    - それCPANでできるよ
- 表現力が高い
    - TMTOWTDI (やりかたはいくつもあるよ！)
- 実際に使われてる
    - はてな/DeNA/NHN/mixi
- 良い開発文化がある
    - http://b.hatena.ne.jp/t/perl

副読サイト

- http://perldoc.jp/ : perldocの日本語訳
- https://metacpan.org/ : CPANの検索

Perlプログラミング勘所

- Perlでおさえておきたい/よくはまるポイントを説明
    - use strict;use warnings;

- データ型
- コンテキスト
- リファレンス
- パッケージ
- サブルーチン

use stirct;use warnings;

- ファイルの先頭には必ず書きましょう

```
 use strict;
use warnings;
```

- デフォルトの振る舞いは互換性のために制限が弱い

use strict; use warnings;を書かないと困ること

```
 my $message = "hello";
print $messsage; # typo!
# => エラーにならない!
$messagge = "bye"; # typo!
# => $messagge がグローバル変数になる！
```

* 細かな振る舞いはperodoc参照

```
 $ perldoc strict
$ perldoc perllexwarn
```

データ型

- スカラ
- 配列
- ハッシュ
- (ファイルハンドル)
- (型グロブ)

スカラ

- 1つの値
- 文字列/数値/リファレンス
- $calar と覚えましょう

```
 my $scalar1 = "test";
my $scalar2 = 1000;
my $scalar3 = \@array; # リファレンス(後述)
```

配列

- @rray と覚えましょう

```
my @array = ('a', 'b', 'c');
```

- Q:@arrayの二番目の要素を取得するには?

配列の操作

```
print $array[1]; # 'b'
```

- スカラを取得するので$でアクセス
- $array と @array とは別の変数

配列の操作

```
 $array[0]; # get
$array[1] = 'hoge'; # set
my $length = scalar(@array); # 長さ
my $last_ids = $#array; # 最後の添字
my @slice = @array[1..4]; # スライス
for my $e (@array) { # 全要素ループ
  print $e;
}
```

* チェックしておこう！ * 関数: push/pop/shift/unshift/map/grep/join/sort/splice * モジュール: List::Util/List::MoreUtils

ハッシュ

- %ash とおぼえましょう

```
 my %hash = (
  perl  => 'larry',
  ruby => 'matz',
);
```

- Q:hashのkey:perlに対応する値を取得するには?

ハッシュの操作

```
 print $hash{perl}; # larry
print $hash{ruby}; # matz
```

- スカラを取得するので$
- $hash と %hash は別の変数
- {} の中は裸の文字列(= '"がない)が許される

ハッシュの操作

```
 $hash{perl}; # get
$hash{perl} = 'larry'; # set
for my $key (keys %hash) { # 全要素
  my $value = $hash{$key};
```

```
}
```

* チェックしておこう！ * 関数: keys/values/delete/exists

データ型まとめ

- スカラ-$ /配列-@ /ハッシュ-%
- $val と @val と %val は別の変数

```
$ perldoc perldata
```

コンテキスト

- Perlといえばコンテキスト
- 代表的ハマリポイント
- 式が評価される場所(= コンテキスト)によって結果が変わる

```
 my @x = (0,1,2);
my ($ans1) = @x;
my $ans2 = @x;
```

* それぞれ何が入っているでしょうか?

コンテキスト

```
 my @x = (0,1,2);
my ($ans1) = @x; # => 0
my $ans2 = @x; # => 3
```

* 配列への代入の右辺はリストコンテキスト * 0が代入される。1,2 は捨てられる * スカラへの代入の右辺はスカラコンテキスト * 配列はスカラコンテキストで評価すると長さが返る

コンテキストクイズ

- <ここ> のコンテキストはスカラ? リスト?

```
 sort <ここ>;
length <ここ>;
if (<ここ>) { }
for my $i (<ここ>) { }
$obj->method(<ここ>);
my $x = <ここ>;
my ($x) = <ここ>;
my @y = <ここ>;
my %hash = (
  key0 => 'hoge',
  key1 => <ここ>,
);
scalar(<ここ>);
```

```
<ここ>;
```

コンテキストまとめ

- 式/値が評価される場所によって結果がかわる
- コンテキストの決まり方は覚えるしかない
    - 組み込み関数に注意 (length など)
    - 組み込み関数以外もprototypeという機能で実現可能なので注意

```
 $ perldoc perldata
$ perldoc perlsub # Prototype の章
```

リファレンス

- スカラ/配列/ハッシュ などへの参照
    - C++とかの参照/Rubyなどではすべて参照
    - データ構造を作るときに重要

データ構造はまりポイント1

- 行列をつくろう

```
my @matrix = (
  (0, 1, 2, 3),
  (4, 5, 6, 7),
);
```

- どうなるでしょうか...

こうなります

```
my @matrixt =
  (0, 1, 2, 3, 4, 5, 6, 7);
```

* ひー

データ構造はまりポイント2

```
my %entry = (
  body => 'hello!',
  comments => ('good!', 'bad!', 'soso'),
)
```

* どうなるでしょうか...

こうなります

```
my %entry = (
  body => 'hello!',
```

```
  comments => 'good',
  bad => 'soso',
);
```

* ひー

なぜか?

- () の中はリストコンテキスト
- リストコンテキスト内ではリストは展開される

```
 my @matrix = (
  (0, 1, 2, 3),
  (4, 5, 6, 7),
);
```

リファレンスの取得/作成 (配列)

```
 my @x = (1,2,3);
my $ref_x1 = \@x;

# 略記法
$ref_x2 = [1,2,3];

# 組み合わせ
$ref_x3 = [@x];
```

デリファレンス (配列)

```
 my $ref_x = [1,2,3];

my @x = map { $_ * 2 } @$ref_x;

print $ref_x->[0]; # 1

my @new_x = @$ref_x;
print $new_x[0]; # 1
```

リファレンスの取得/作成 (ハッシュ)

```
 my %y = (
  perl => 'larry',
  ruby  => 'matz',
);
my $ref_y1 = \%y;

# 略記法
```

```
$ref_a2 = {
  perl => 'larry',
  ruby => 'matz',
}
```

デリファレンス (ハッシュ)

```
 my $ref_y = {
  perl => 'larry',
  ruby => 'matz',
};

my @keys = keys %$ref_y;

print $ref_y->{perl}; # larry

my %new_f = %$ref_y;
print $new_f{perl}; # larry
```

データ構造の作成

- リファレンスはスカラ値 = リストコンテキストで展開されない

```
 my $matrix = [
  [0, 1, 2, 3],
  [4, 5, 6, 6],
];
```

```
 my $entry = {
  body => 'hello!',
  comments => ['good!', 'bad!', 'soso'],
};
```

複雑なデリファレンス

- 例: リファレンスを返すメソッドの返り値をデリファレンス

```
 my $result = [
    map {
        $_->{bar};
    }
    @{ $foo->return_array_ref }
     # ↑ ブレースを使う
];
```

おすすめ

- 基本的にリファレンス以外使わない
    - ハマリにくい

```
 my @foo = (1, 2, 3);
my %foo = ( a => 1, b => 2, c => 3);
$foo[1], $foo{a}
# ↑ 同じ変数を参照しているように見える……
# が実際は違う変数
```

```
 my $foo = [1, 2, 3];
my $foo = { a => 1, b => 2, c => 3};
# ↑ 同じ変数なので warning がでる
```

おすすめ

必要なときだけデリファレンスする

```
 my $list = [1, 2, 3];
push @$list, 4;
```

リファレンスでないリスト/ハッシュを使うと便利

- サブルーチンの引数の処理
- 多値を返すとき

```
 sub hello {
  my ($arg1, $arg2, %other_args) = @_;
  return ($arg1, $arg2);
}
my ($res1, $res2)
  = hello('hey', 'hoy', opt1 => 1, opt2 =>2);
```

リファレンスまとめ

- スカラ/配列/ハッシュへの参照
- 複雑なデータ構造を扱うときに必須
- 記法がちょっと複雑

    ```
     $ perldoc perlreftut
    $ perldoc perlref
    ```

パッケージ

- 名前空間
- モジュールロードのしくみ
- クラス(後述)

```
package Hoge;
```

```
our $PACKAGE_VAL = 10;
# $HOGE::PACKAGE_VAL == 10

sub fuga {
}
# &Hoge::fuga;

1;
```

モジュールロードのしくみ

```
use My::File;
# => My/File.pm がロードされる
```

* @INC(グローバル変数)に設定されたパスを検索

```
use lib 'path/to/your/lib';
$ perl -Ipath/to/your/lib;
```

- path/to/your/lib/My/File.pm をさがしてあれば読み込む

サブルーチン

```
&hello # 定義前に括弧なしで呼ぶにはは&がいる
sub hello {
  my ($name) = @_; # @_内の自分で処理
  return "Hello, $name";
}
hello();
hello; 定義後であれば括弧は不要
```

引数処理イディオム1

```
sub func1 {
  my ($arg1, $arg2, %args) = @_;
  my $opt1 = $args{opt1};
  my $opt2 = $args{opt2};
}
func1('hoge', 'fuga', opt1 => 1, opt2 => 2);
```

引数処理イディオム2

```
sub func2 {
  my $arg1 = shift; # 暗黙的に@_を処理(破壊的)
  my $arg2 = shift;
  my $args = shift;
```

```
  my $opt1 = $args->{opt1};
  my $opt2 = $args->{opt2};
}
```

引数処理イディオム3

```
sub func3 { shift->{arg1} }
```

```
sub func4 { $_[0]->{arg1} } # @_ の第0要素
```

サブルーチンの名前空間

- パッケージに定義される

```
pacakge Greetings;
sub hello { }
1;

# hello は &Greeting::hello(); として定義される
```

- ネストしてもパッケージに定義されるので注意

```
pacakge Greetings;
sub hello {
  sub make_msg { }
  sub print {}
  print (make_msg() );
}
1;

# &Greeting::hello();
# &Greeting::make_msg();
# &Greeting::print();
```

Perlでデバッグ

- use Data::Dumper; を良く使います

```
use Data::Dumper;
warn Dumper($value); # スカラ値がよい
```

* エディタのマクロに登録しておこう！

質問 => 休憩

Perlによるオブジェクト指向プログラミング

質問

- オブジェクト指向プログラミングしたことありますか?
    - Perl以外でも

ですよねー

- 念のためポイントだけおさえておきます

プログラミングにおける抽象化の歴史

- 抽象化とは

> 詳細を捨象し、一度に注目すべき概念を減らすことおよびその仕組み (Wikipediaより)

- 一度に考えないといけないことを減らす
    - = スコープをせばめる
    - 保守性/再利用性を高める

非構造化プログラミング時代

- gotoプログラミング
    - 制御のながれがすべて自由
- Perl で gotoプログラミングした例(fizzbuzz)

```
 my $i = 1;
START:
    goto "END" if $i > 35;

    goto "PRINT_FIZZBUZZ" if $i % 15 == 0;
    goto "PRINT_FIZZ"     if $i %  3 == 0;
    goto "PRINT_BUZZ"     if $i %  5 == 0;
    goto "PRINT_NUM";

PRINT_NUM:_
    print $i;
    goto "PRINT_NL";

PRINT_FIZZ:_
    print "fizz";
    goto "PRINT_NL";

PRINT_BUZZ:_
    print "buzz";
    goto "PRINT_NL";

PRINT_FIZZBUZZ:_
    print "fizzbuzz";
```

```
    goto "PRINT_NL";

PRINT_NL:_
    print "\n";

    $i++;
    goto "START";

END:
```

非構造化プログラミングの問題

- 制御の流れが分かりにくい
    - プログラム全体を一度に理解していないといけない
    - 大規模なソフトウェアになると保守できなくなる
- コードの再利用は実質的にはできない

`EW Dijkstra(1968). Go to statement considered harmful`

構造化プログラミング

- 手続きを逐次、選択、繰り返しで表現
- サブルーチンにより手続きを抽象化

```
 sub fizzbuzz {
    my $i = 1;

    while ($i < 35) {
        if ($i % 15 == 0) {
            print "fizzbuzz";
        }
        elsif ($i % 3 == 0) {
            print "fizz";
        }
        elsif ($i % 5 == 0) {
            print "buzz";
        }
        else {
            print $i;
        }
        print "\n";
        $i++;
    }
}
fizzbuzz();
```

構造化プログラミングのみを用いる問題

- 手続きとデータがばらばら

```
 open my $fh, '<', $filename;
while (my $line = readline($fh)) {
  print $line;
}
close $fh;
```

* $fhに対してどのような操作ができるのか? * open/close/readline はどんなデータを操作できるのか? * データと手続きそれぞれのスコープが広い

オブジェクト指向プログラミング

- オブジェクトによる抽象化
- オブジェクトとは
    - プログラムの対象となるモノ
    - データ + 手続き
- プログラムはオブジェクト同士の相互作用
- "どう"ではなく"なにがどうする"に着目する

オブジェクト指向プログラミングの良いところ

- 処理の対象(データ)と処理の内容(手続き) が結びついている
    - オブジェクトごとにコードを理解できる
    - 再利用しやすい
- オブジェクトというメタファが人間にとってわかりやすい (こともある)
    - => 設計しやすい

例

```
 use IO::File;
my $file = IO::File->new($filename, 'r');
while (my $line = $file->getline) {
  print $line;
}
$flie->close;
```

* $fileに対してできる操作はすべてメソッドとして定義されている

オブジェクト指向プログラミングの実現

- オブジェクト間の相互作用/メッセージパッシング
- オブジェクト間の相互作用を表現しやすいようにプログラミング言語が支援
    - クラスとインスタンス
    - カプセル化
    - 継承
    - ポリモーフィズム
    - ダイナミックバインディング

本題

Perlにおけるクラスとインスタンス

```
       クラス: [ データ構造定義 + 手続定義 ]
                          ↓ 生成
インスタンス: [ { データ } + 手続への参照 ]
```

|クラス|パッケージ| |メソッド|パッケージに定義された手続き| |オブジェクト|特定のパッケージにbless()されたリファレンス|

クラス定義 (クラス名)

- 課題ででたSorterクラス(簡易版)

```
 # パッケージに手続きを定義
package Sorter; # クラス名
use strict;
use warnings;

# 続く
```

クラス定義 (コンストラクタ/フィールド)

```
 # コンストラクタ
# Sorter->new; のように呼び出す
sub new {
    my ($class) = @_; # クラス名が入る
    # データを用意する
    my $data_structure = {
        values => [],
    };
    # 手続き(= パッケージ)とデータを結びつける
    my $self = bless $data_structure, $class;
    return $self;
}
# 続く
```

クラス定義 (メソッド)

```
 #$sorter->set_values(0,1,2,3) のように呼び出す
sub set_values {
    my ($self, @values) = @_; # $self には$sorterが入る
    $self->{values} = [@values];
    return $self;
}

sub get_values {
    my ($self) = @_;
```

```
    return @{ $self->{values} };
}

sub sort {
    my ($self) = @_;

    $self->{values} = [ sort { $a <=> $b } @{ $self->{values} } ];
}

1; # おまじない
```

クラスの使用

```
use Sorter;

my $sorter = Sorter->new;
$sorter->set_values(5,4,3,2,1);
```

コンストラクタ

- コンストラクタは自分で定義する
- オブジェクトも自分で作る
- new()
    - リファレンス を パッケージ(クラス) で bless して返す
- blessはデータと手続きを結びつける操作

```
my $self = bless { field => 1 }, "Sorter";
```

クラスメソッドとインスタンスメソッド

- 文法的違いはない
- 定義時: 第一引数を$classとみなすか$selfとみなすか
- 呼出時: クラスから呼び出すかインスタンスから呼び出すか

```
Class->method($arg1, $arg2);
&Class::method("Class", $arg1, $arg2);

$object->method($arg1, $arg2);
&Class::method($object, $arg1, $arg2);
```

フィールド

- 1インスタンスに付き1データ(のリファレンス)
- 複数のデータをもちたい場合はハッシュをbless する

```
my $self = bless {
    filed1 => $obj1,
```

```
    field2 => [],
    field3 => {},
}, $class;
```

カプセル化

- 可視性の指定(public/privateなど) はない
    - すべてが public
- 命名規則などでゆるく隠蔽する

```
sub public_method {
  my $self = shift;
}

sub _private_method {
  my $self = shift;
}
```

* 完全に隠蔽する方法もある(クロージャを使う)

継承

- use base を使う

```
package Me;
use base "Father";
1;
```

* 親クラスのメソッド * SUPER

```
sub new {
  my ($class) = @_;
  my $self = $class->SUPER::new();
  return $self;
}
```

多重継承

- Mixiinをやりたいときなどにつかう
- 乱用しない

```
package Me;
use base qw(Father Mother); # 左 => 右の順
1;
```

* メソッドの検索アルゴリズム * Class::C3 * Next

オブジェクト指向のまとめ

- 手作り感あふれるオブジェクト指向
    - package に手続きを定義
    - blessでデータと結びつける
    - コンストラクタは自分でつくる、オブジェクトも自分で作る
    - オブジェクト志向風によびだせるような糖衣
- オブジェクト指向に必要な機能はそろっている

UNIVERSAL

- JavaでいうObjectのようなもの
- UNIVERSALに定義するとどのオブジェクトからも呼べる
- isa()

```
 my $dog = Dog->new();
$dog->isa('Dog');    # true
$dog->isa('Animal'); # true
$dog->isa('Man');    # false
```

* can()

```
 my $bark = $dog->can('bark');
$man->$bark();
```

AUTOLOAD

*呼び出されたメソッドがMy::Classクラスに見つからない場合、 * My::Class::AUTOLOADメソッドを探す * 親クラスのAUTOLOADメソッドを探す * UNIVERSAL::AUTOLOADを探す * なかったらエラー

- AUTOLOADメソッドで未定義のメソッド呼び出しを補足
- Ruby の method_missing

AUTOLOAD

- フィールドを動的に定義できたりする
- 想像できない振る舞いを作り出し得るのでなるべく使わない
    - こういう仕組みがあることは理解しておく

```
 package Foo;

sub new { bless {}, shift }

our $AUTOLOAD;
sub AUTOLOAD {
    my $method = $AUTOLOAD;
    return if $method =~ /DESTROY$/;
    $method =~ s/.*::///;
    {
        no strict 'refs';
        *{$AUTOLOAD} = sub {
            my $self = shift;
```

```
            sprintf "%s method was called!", $method;
        };
    }
    goto &$AUTOLOAD;
}

1;
```

演算子のオーバーロード

- 想像できない振る舞いを作り出し得るのでなるべく使わない
    - こういう仕組みがあることは理解しておく
- URI

```
 my $uri = URI->new('http://exapmle.com/');
... do something ...
print "URI is $uri";
```

- DateTime

```
 $new_dt = $dt + $duration_obj;
$new_dt = $dt - $duration_obj;
$duration_obj = $dt - $new_dt;
for my $dt (sort @dts) {           # sort内で使われる<=>がoverloadされている
    ...
}
```

クラスビルダー

- Perlのオブジェクト指向は手作り感満載
    - newは自分でつくる
    - フィールドのアクセサも自分で定義
- たいへんなので自動化されている

Class::Accessor::Fast

- コンストラクタ/フィールドのアクセサを自動的に定義

before

```
 package Foo;

sub new {
    bless {
        foo => undef,
        bar => undef,
        baz => undef,
    }, shift
```

```
}

sub foo {
    my $self = shift;
    $self->{foo} = $_[0] if defined $_[0];
    $self->{foo};
}

sub bar {
    my $self = shift;
    $self->{bar} = $_[0] if defined $_[0];
    $self->{bar};
}

sub baz{
    my $self = shift;
    $self->{baz} = $_[0] if defined $_[0];
    $self->{baz};
}

1;
```

after

```
 package Foo;

use base qw(Class::Accessor::Fast);
__PACKAGE__->mk_accessors(qw(foo bar baz));

1;
```

Class::Data::Inheritable

- 継承可能なクラス変数を作成する

```
 use base qw(Class::Data::Inheritable);
__PACKAGE__->mk_classdata(dsn => 'dbi:mysql:dbname=foo');
1;
```

よくあるクラス

```
 package UsualClass;
use base qw(Class::Accessor::Fast Class::Data::Inheritable);
__PACKAGE__->mk_classdata(hoge => "classdata");
__PACKAGE__->mk_accessors(qw(foo bar baz));
```

```
# メソッド定義とか

1;
```

他のクラスビルダー

Class::Accessor::Lite

```
 package Foo;

use Class::Accessor::Lite (
    new => 1,
    rw  => [ qw(foo bar baz) ],
);
```

* 継承ツリーを汚さない * おすすめ

Moose

- モダンなオブジェクト指向を実現するモジュール
- 柔軟かつ安全なアクセサの生成
- 型の採用
- Roleによるインタフェイス指向設計
- プロジェクトの複雑性をあげるのであまりつかわない

Mouse

- Mooseの軽量版
    - Mooseはモジュール読み込み時のコストが高い
    - 機能は一部制限

オブジェクト指向でプログラムを書くコツ

- 登場人物を考える = オブジェクト
- 登場人物がそれぞれどのような責務を持つべきかを考える。
- 責務にあわせてスコープを限定するように書く。
    - 「カプセル化で継承でポリモーフィズムが……」とか考えても意味ない
    - よりよい、わかりやすく問題をモデリングするための手段

責務とは？

- オブジェクトの利用者、メソッドの呼び出し元との約束
    - 責任のないことはやらなくていい
    - 責任のないことはやっちゃだめ
- 責務を綺麗に切り分けることで、綺麗に設計できる

オブジェクト指向のまとめ

- 手作り感あふれるオブジェクト指向
    - package に手続きを定義
    - blessでデータ(リファレンス)と結びつける
    - コンストラクタは自分でつくる
    - オブジェクト志向風によびだせるような糖衣

テストを書こう

テスト重要

- プログラムを変更する二つの方法 [レガシーコード改善ガイドより]
    - 編集して祈る
    - テストを書いて保護してから変更する

なぜテストを書くのか

- テストがないと、プログラムが正しく動いているかどうかを証明できない
- 大規模プロジェクトでは致命的
    - 昔書いたコードは今もうごいているのか?
    - 新しくコードと古いコードの整合性はとれているのか?
    - 正しい仕様/意図が何だったのかわからなくなっていないか?
- Perlのような型のない動的言語では特に重要
- 祈らずテストを書こう！

何をテストするのか

- 正常系
- 異常系
- 境界
- 100% の カバーは難しい
    - 命令網羅(C0)/分岐網羅(C1)/条件網羅(C2)
    - C2 とかはたいへん
- 必要/危険だと思われるところから書き、少しづつ充実する

PerlでTest

- Test::More
- Test::Fatal
- Test::Class

Test::More

```
 use Test::More;

subtest 'topic' => sub {
 use_ok    'Foo::Bar';
 isa_ok    Foo::Bar->new, 'Foo::Bar';
 ok        $something_to_be_bool;
 is        $something_to_be_count, 5;
 is_deeply $something_to_be_complicated, {
     foo => 'foo',
     bar => [qw(bar baz)],
 };
```

```
};

done_testing;
```

Test::Fatal

```
 use Test::Fatal;

ok( exception{ $foo->method }, '例外が発生する');
like( exception { $foo->method },  qr/division by zero/, '0除算エラーが発生する');
isa_ok( exception { $foo->method }, 'Some::Exception::Class', '例外クラスがthrowされる);
```

Test::Class

- メソッドにテストコードをわける
- xUnit系
- Sorterのテストをみてね！

その他

Test::Name::FromLine

- 名前のついていないテストを見やすく表示してくれる

Test::Most

- 便利なテストモジュールを一気にuseしてくれる

テストの実行

- テストコードは t ディレクトリに.t拡張子をつけて保存
  - t/hoge.t
- proveコマンド(Test::Moreに付属)で実行する

```
 $ prove -lvr t
t/hoge.t ..
ok 1 - L8: is Hoge::hey(10), 100;
1..1
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.02 usr  0.01 sys +  0.03 cusr  0.01 csys =  0.07 CPU)
Result: PASS
```

テストを書くコツ

- まず、こういう振る舞いで有るべきというテストを書く

```
is_deeply( [numsort(2,3,4,0,1)], [0,1,2,3,4], 'ランダムな数列をsortすると昇順に並ぶ'
);
```

* 次に境界条件での振る舞いを検証するテストを書く

```
is_deeply( [numsort()], [], '空配列をsortしたら空になる' );
is_deeply( [numsort(100)], [100], '1要素ならそのまま' );
```

* 例外条件についても確かめる

```
ok( exception { [numsort('hoge')] },'文字をわたすと例外発生' );
```

Test::Hatena

- はてな社内で標準化中のテストフレームワーク
- Test::More/Test::Exception 他、便利な関数
- 単体テスト/結合テスト/副作用のあるテストに対応

リファクタリング

- リファクタリングとは？
  - プログラムの振舞を変えずに実装を変更すること
- テストがなければ、外部機能の変更がないことを証明できない。
  - テストがなければリファクタリングではない
- レガシーなコードに対してはどうする？
  - まずは、テストを書ける状態にしよう。
- テストを書いてリファクタリングし、常に綺麗で保守しやすいコードを書きましょう

ヒント

ドキュメントを引きましょう

- perldoc perltoc 便利！
  - 定義済み変数　$_ @_ $@
  - perldoc perlvars を見るべし
- 関数
  - perldoc -f open
  - http://perldoc.jp/ : perldocの日本語訳
- CPANモジュール
- https://metacpan.org/

良い本を読みましょう

- はじめてのPerl
- 続・はじめてのPerl
- Perlベストプラクティス
  - Perl::Critic
- モダンPerl入門

プロジェクトのコードを書く心構え

- コードが読まれるものであることを意識する
    - あとから誰が読んでもわかりやすく書く
    - 暗黙のルールを知る => コードを読みまくる
- テストを書いて意図を伝える

まとめ

- Perlプログラミング勘所
- Perlによるオブジェクト指向プログラミング
- テストを書こう
- ヒント
- 今日かいて今日はまろう
    - と言っても時間はあまりないので無理せず

課題

課題1

この講義をふまえて、事前課題で作成したSorterクラスを改良・完成させてください。

以下再掲

以下のようなインターフェースをもつ、整数値を昇順にソートするソート器クラスをPerlを用いて実装してください。余裕がある人は、アルゴリズム別のSorterのサブクラスを実装してみてください。

```
 my $sorter = Sorter->new;
$sorter->set_values(5,4,3,2,1);
$sorter->sort;
$sorter->get_values # (1,2,3,4,5) が返ってくる
```

実装に際しては、以下の条件を守ってください。

- Perl組み込みのsort関数やソートを行うためのCPANモジュール を利用しない
- アルゴリズムはクイックソートを用いる

課題2

オブジェクト指向連結リスト My::List を作成してください。(C言語で良く使われる連結リストデータ構造を、オブジェクト指向Perlで書いた物、と思ってください。)

リストの各要素には任意のデータ (スカラー、オブジェクト、リファレンス etc) を保存できるものとします。連結リストはオブジェクト指向インタフェースを持ちます。リストの要素を先頭から手繰る(たぐる)のにイテレータを用意してください。

My::List のプログラムインタフェースは以下のようになるでしょう。例は、リストに "Hello" "World" "2008" という 3 つの文字列を保存して、イテレータでそれらを取り出し出力するコードです。

```
 my $list = My::List->new;

$list->append("Hello");
$list->append("World");
$list->append(2008);
```

```
my $iter = $list->iterator;
while ($iter->has_next) {
    print $iter->next->value;
}
```

リストの各要素は My::List 内に配列として保持するのではなく、ハッシュベースなリスト要素のオブジェクトをリファレンスで繋いだ連結リストとして保持する、という点に注意してください

- 連結リストがどのようなデータ構造か分からない場合は『定本 Cプログラマのためのアルゴリズムとデータ構造 (SOFTBANK BOOKS)』 P.50 - 66 を参照してください。(なお、連結リストなどはアルゴリズムとデータ構造の基礎ですので、その知識がない場合は本書籍を通読することをお薦めします。)
- イテレータについては『増補改訂版Java言語で学ぶデザインパターン入門』の第一章を参照してください。

両書籍とも、会社の本棚にあります。

課題3(オプション)

twitterもどきをつくろう。

以下のようなことができる小鳥オブジェクトを実装してください。

- 小鳥はつぶやくことができる
- 小鳥は他の小鳥をfollowできる
- 小鳥はfollowしている小鳥のつぶやきを見ることができる

いろいろな設計方法が考えられます。すっきりしたかっこいいのを考えてみましょう。 余裕があれば、mentions(@記法)やunfolllow や block、 lists などの機能も実装してみてください。

プログラムのインターフェースは自由です。例えば以下のようなインターフェースが考えられます。下の例では、SmallBirdクラスしか存在していませんが、つぶやき全体を管理するクラスがあっても良いかもしれません。Observerパターンを使ってみてもよいでしょう。

```
 use Bird;
my $b1 = Bird->new( name => 'jkondo');
my $b2 = Bird->new( name => 'reikon');
my $b3 = Bird->new( name => 'onishi');

$b1->follow($b2);
$b1->follow($b3);

$b3->follow($b1);

$b1->tweet('きょうはあついですね！');
$b2->tweet('しなもんのお散歩中です');
$b3->tweet('烏丸御池なう！');

my $b1_timelines = $b1->friends_timeline;
print $b1_timelines->[0]->message; # onishi: 烏丸御池なう!
print $b1_timelines->[1]->message; # reikon: しなもんのお散歩中です

my $b3_timelines = $b3->friends_timeline;
print $b3_timelines->[0]->message; # jkondo: 今日はあついですね！
```

課題に取り組む際の注意点

- できるだけテストスクリプトを書く
  - 少くとも動かして試してみることができないと、採点できません
  - 課題の本質的なところさえ実装すれば、CPANモジュールで楽をするのはアリ
  - 何が本質なのかを見極めるのも課題のうち
- 余裕があったら機能追加してみましょう
- 課題については以下の通り、ディレクトリを作成してコミットしてください。
  - pushするのを忘れずに!

```
intern/username/100802/exercise1
                      /exercise2
                      /exercise3
               /100803/exercise1
                      /exercise2
                      /   ...
                      /   ...
```

Perlの慣習として、以下のディレクトリ構成でコミットするといろいろ良いです。

```
.
|-- lib
|   `-- MyClass
|       `-- Foo.pm
`-- t
    `-- 00_base.t
```

* ライブラリを置くディレクトリはlib * テストファイルを置くディレクトリはt * テストスクリプトは prove -Ilib t で実行できます * あるいはテストスクリプト単体を perl -Ilib t/00_base.t で実行します。

課題の採点基準

各課題の満点 = 課題1: 4点 + 課題2: 4点 + 課題3: 2点 = 10点

- 講義および教科書の理解度が課題の実装に反映されているかどうか。
- テスト用スクリプトが実際に動作するかどうか。
- 設計・コードがきれいに書けているかどうか。
- 独自に機能追加が行われているかどうか。
- 課題1と課題2をじっくりやったあと3に挑戦してください

やること

- git clone
- 課題