PB173 Perl

9 Objekty

Roman Lacko xlacko1@fi.muni.cz

2023-04-19

Obsah

Objekty

Objekty: OOP à la Perl

Trieda ≈ Menný priestor (`package`) s metódami.

Objekt ≈ Hodnota priradená k balíku kľúčovým slovom `bless`.

Metóda ≈ Funkcia, ktorá očakáva *objekt* ako prvý argument.

Atribúty Nemajú priamy ekvivalent, ale môžu sa použiť položky základného dátového typu (hash, pole).

Objekty: Kľúčové slovo `bless`

```
bless REF, CLASSNAME
bless REF
```

Priradí hodnotu odkazovanú `REF` do menného priestoru (*package*) `CLASSNAME`.
Od tohto momentu je na `REF` možné volať metódy.

Vráti svoj prvý argument.

Ako volať `bless`
Bez použitia `CLASSNAME` sa použije názov aktuálneho menného priestoru.
Nepoužívajte, rozbije sa pri dedičnosti (viď príklady).

Objekty: Kľúčové slovo ‹bless›

Efekt ‹bless› na hodnote a premennej
‹bless› síce očakáva referenciu, ale pracuje nad odkazovanou hodnotou. Na premennú ani referenciu efekt nemá.

```
my %hash;
my $ref1 = \%hash;

bless $ref1, 'Class';   # ‹%hash› is now an object
$ref1->method;

my $ref2 = \%hash;      # ‹$ref2 = $ref1› would yield the same result
$ref2->method;          # OK

my $ref1 = undef;       # No effect on ‹%hash› nor remaining references
$ref2->method;          # OK, ‹%hash› is still an object
```

Objekty: Kľúčové slovo ‹bless›

Argument ‹bless›
Argument ‹bless› nemusí byť nutne skalárna premenná, stačí referencia.

```
my @array;
bless \@array, 'Class'; # ‹@array› is now an object too
(\@array)->method;      # OK, but grotesque

my $ref = bless { name => 'John Zoidberg' }, 'Crustacean';
$ref->method;           # OK
```

Objekty: Operátor `->`

Ak vo výraze `X->m` je `X` trieda alebo referencia na objekt, Perl zavolá metódu.

Metóda je funkcia, ktorá dostane ľavý operand `->` ako prvý argument.

Volanie metódy na triede

```
Foo::Bar->method(1, 2);
Foo::Bar::method("Foo::Bar", 1, 2);

my $package = "Foo::Bar";
$package->method(1);
Foo::Bar::method($package, 1);
```

Objekty: Operátor ‹->›

Volanie metódy na objekte

```
my $object = Some::Class->new;  # ≈ bless SOMETHING, "Some::Class";

$object->method(1, 2);
Some::Class::method($object, 1, 2);
```

Objekty: Volanie metódy a dereferencia

Mohlo by sa zdať, že volanie metódy na triede je symbolická dereferencia.
V skutočnosti sa líšia syntakticky:

```
my $pkg = "Something";

$pkg->foo();            # Method call ‹Something::foo("Something")›
$pkg->();               # Symbolic reference call to ‹&Something()›
                        # Cannot use string as a subroutine ref …
```

O symbolickú dereferenciu nejde ani v prípade, že meno metódy je v premennej:

```
my $method = 'foo';
$pkg->$method(1, 2);    # Calls ‹Something::foo("Something", 1, 2)›
```

Objekty: Metódy

Funkcie, ktoré očakávajú objekt objekt ako prvý argument.

Perl nevynucuje meno prvého argumentu, typicky sa však používa konvencia:

`$class`	pre metódy triedy (`Package->method()`)
`$self`	pre metódy objektu (`$object->method()`)
`$ref`	(zriedkavo) ak metóda nerozlišuje objekt a triedu

```
sub Person::new($class, %args) {          # Person->new(...);
    return bless { %args }, $class;
}

sub Person::full_name($self) {            # $person->full_name;
    return join ' ', $self->@{'first_name', 'last_name'};
}
```

Objekty: Konštruktor

Metóda triedy, ktorej úlohou je vytvoriť objekt (`bless`).
Typicky sa volá `new()`, Perl však názov nijak nevynucuje.

```
sub Person::new($class, $name, $surname, $age) {
    return bless {
        name => $name, surname => $surname, age => $age,
    }, $class;
}
```

Ako podkladový dátový dátový typ sa typicky používa hash,
pretože má pomenované „atribúty“ a je jednoducho rozšíriteľný.
Môžeme však použiť aj pole alebo skalár.

Objekty: Atribúty

Perl nemá žiadny priamy mechanizmus na zneprístupnenie častí objektu.
Rovnako nie je možné metódu označiť ako „privátnu“.

```
my $person = Person->new(name => "Jared", age => 19);
say $person->{name};            # Jared
```

Obyklé konvencie:

- K atribútom objektu sa mimo jeho metód nepristupuje.
  Alternatívne: Atribúty začínajúce ‹ _ › sú privátne.
- Metódy začínajúce ‹ _ › sa považujú za privátne.
- Všetko ostatné je verejné.

Objekty: Prístupové metódy

Angl. *accessors*, alebo *getters* a *setters*.

Za predpokladu, že sa rešpektujú konvencie o prístupe k atribútom, tieto metódy majú za cieľ kontrolovať prístup.

Používajú sa typicky dva spôsoby implementácie:

1. Samostatné metódy, napr. `get_x($self)` a `set_x($self, VALUE)`.
   - Pre atribúty „len na čítanie“ `set_x()` nevytvárame.
2. Jedna metóda, ktorá očakáva nepovinný argument na nastavovanie.
   - Prístupové metódy atribútov „len na čítanie“ druhý argument neakceptujú.

Objekty: Prístupové metódy

Ukážka prístupovej metódy (druhý variant):

```
sub Person::age($self, $age = undef) {
    return $self->{age} if !defined $age;       # Getter

    _validate_age($age);
    return $self->{age} = $age;                 # Setter
}
```

Ak by `undef` bola povolená hodnota pre `$age`, musíme zvoliť iný prístup.

```
sub Person::age($self, @age) {
    die "Too many arguments" if @age > 2;
    ...
}
```

Objekty: Deštruktor

Hodnoty v Perli zanikajú, keď počet referencií na nich klesne na 0 (*reference-counting*) na konci rozsahu platnosti poslednej premennej.

Ak je rušená hodnota objekt a má metódu `DESTROY()`, zavolá sa.

Perl garantuje, že `DESTROY()` sa zavolá presne na konci rozsahu premennej. Týmto mechanizmom je možné implementovať RAII.

Typicky sa používa na riadené uvoľnenie zdrojov (zámky, dočasné súbory, ...).

Zdedený deštruktor sa automaticky nezavolá!

Objekty: Deštruktor

```
sub File::Lock::write($class, $handle) {
    flock $handle, LOCK_EX
        or Carp::croak("Cannot lock file: $!");
    return bless \$handle, $class;
}

sub File::Lock::DESTROY($self) {
    flock $self->$*, LOCK_UN;
}

sub do_something_important($file) {
    my $lock = File::Lock->write($file);
    # Only we can write to the file now
    say { $file } dump_data();
}   # <$lock->DESTROY()> gets called here automatically
```

Objekty

V Perli sme zatiaľ videli zabudované objekty:

- `File::stat`
- Súbory — každý otvorený súbor je `IO::File`
- Adresáre — každý otvorený adresár je `IO::Dir`

`IO::File` a `IO::Dir` sú potomkami `IO::Handle`

Dedičnosť

Dedičnosť

Ak trieda obsahuje globálnu (*package*) premennú `@ISA`,
zdedí metódy z tried, ktoré sú v nej vymenované.

```
package Foo;
sub a(@);
sub b(@);

package Bar;
@Bar::ISA = qw{Foo};
# Bar->a and Bar->b are inherited from Foo
```

Dedičnosť

`@ISA` má efekt len na rezolúciu *metód*!

```
package Bar;
@Bar::ISA = qw{Foo};

Bar->a(1);          # Calls inherited <Foo->a(1)>
Bar::a(1);          # Wrong, looks for <Bar::a>
```

Perl dovoľuje dediť z viac než jednej triedy (*multiple inheritance*).
Používajte opatrne alebo vôbec, ak nemusíte.

Dedičnost: Pragma ‹parent›

```
use parent LIST;
use parent -norequire, LIST;
```

Preferovaný spôsob manipulácie ‹@ISA› v modernom Perli.

Približne ekvivalentné

```
BEGIN {
    require $_ foreach LIST;    # Unless ‹-norequire› is specified.
    push @ISA, LIST;
}
```

Dedičnosť: Prekrytie metód

Ak v mennom priestore vytvoríme funkciu s rovnakým menom,
ako má nejaká funkcia v predkovi, dôjde k *prekrytiu*.

Niekedy však chceme zavolať prekrytú metódu, typicky konštruktor:

```
sub Parent::new($class, @args) {
    return bless [@args], $class;
}

@Child::ISA = qw{Parent};       # ‹parent› pragma would be better
sub Child::new($class, @args) {
    # …?
}

my $obj = Child->new;           # Calls ‹Child::new("Child")›
```

Dedičnosť: Prekrytie metód

Prekryté metódy môžeme zavolať plným menom za ‹->›:

```
sub Child::new($class, @args) {
    my $self = $class->Parent::new(@args);
    # …
}
```

Keďže ‹$x->m› vždy zavolá ‹CLASS::m($x, …)›, tak aj v tomto prípade je toto volanie ekvivalentné

```
Parent::new($class, @args);
```

Dedičnosť: ‹SUPER›

Ak nechceme opakovať meno rodičovkej triedy, môžeme použiť ‹SUPER›.
Táto (pseudo)trieda vyberie správneho rodiča **triedy** (nie objektu).

```
@Child::ISA = qw{Parent};

sub Child::new1($class) {
    return $class->Parent::new;
}

sub Child::new2($class) {
    return $class->SUPER::new;  # Same thing
}
```

Dedičnosť: ‹SUPER›

‹SUPER› má význam len na pravej strane ‹->›.

```
sub Child::cry($self) {
    SUPER::lament($self);   # Regular function call in package ‹SUPER›
    SUPER->howl($self);     # Call on class ‹SUPER›
}
```

Tiež si dajte pozor na ‹::› a ‹->›:

```
sub Child::puke($self, $target) {
    $self::SUPER->barf;     # Sub dereference on ‹$SUPER› in package ‹self›
}
```

Dedičnosť: Späť k ‹bless›

Prečo pri ‹bless› používame ‹$class›?

```
sub Parent::new($class, %args) {
    return bless { %args }, $class;
}
```

Skúsme použiť niečo iné:

```
sub Parent::new($class, %args) {
    return bless { %args };     # Or ‹"Parent"› or ‹__PACKAGE__›
}
```

Pokazí sa niečo v konštruktore potomka?

Dedičnosť: Späť k ‹bless›

1. Zavoláme konštruktor potomka:

```
Child->new(%args);                # ≈ Child::new("Child", %args);
```

2. Potomok pomocou ‹SUPER› zavolá zdedený konštruktor:

```
$class->SUPER::new(%args);        # ≈ Parent::new("Child", %args);
```

3. V rodičovskom konštrukore však zavoláme:

```
bless { %args };                  # ≈ bless { %args }, "Parent";
```

vytvorili sme inštanciu ‹Parent›, nie ‹Child›!

Dedičnosť: Späť k ‹bless›

Ak použijeme ‹$class›, konštruktor predka vytvorí správu inštanciu:

1. Zavoláme konštruktor potomka

```
Child->new(%args);              # ≈ Child::new("Child", %args);
```

2. Potomok pomocou ‹SUPER› zavolá zdedený konštruktor:

```
$class->SUPER::new(%args);      # ≈ Parent::new("Child", %args);
```

3. Rodičovský konštruktor vyrobí inštanciu ‹Child›:

```
bless { %args }, $class;        # ≈ bless { %args }, "Child";
```

Dedičnosť: ‹ref›

Ako zistíme, ku ktorej triede je priradený objekt?

Operátor ‹ref› pre posvätené referencie vráti meno triedy:

```
my $obj = {};
say ref $obj;                    # HASH

bless $obj, 'Some::Package';
say ref $obj;                    # Some::Package
```

Čo ak chceme rozlíšiť obyčajné referencie od objektov?

Dedičnosť: ‹Scalar::Util›

Čo ak chceme rozlíšiť obyčajné referencie od objektov?

- Môžeme použiť funkciu ‹blessed()› z modulu:

```
Scalar::Util::blessed($ref);
```

  Ak je hodnota objekt, vráti meno triedy objektu.
  Vráti ‹undef› pre čokoľvek iné, vrátane bežných referencií.

- Pre úplnosť, máme aj „silnejšie ‹ref›“:

```
Scalar::Util::reftype($ref);
```

  Ak je hodnota bežná referencia, správa sa ako ‹ref›.
  Pre objekt vráti názov podkladového dátového typu.

Dedičnosť: ‹builtin›

Táto vlastnosť je experimentálna od ‹v5.36›

Niektoré funkcie ‹Scalar::Util› sú dostupné v module ‹builtin›.

```
use experimental 'builtin';
use builtin qw{blessed reftype};

say blessed($object);  # ≈ ‹Scalar::Util::blessed›
say reftype($object);  # ≈ ‹Scalar::Util::reftype›
```

Dedičnosť: `UNIVERSAL`, `isa`

Ako zistíme, či je objekt inštanciou nejakej triedy?

```
blessed($object) eq "Some::Class";
```

Čo ak `$object` je inštanciou `Some::Subclass` (podtrieda `Some::Class`)?

```
say blessed($object);                    # Some::Subclass
say blessed($object) eq "Some::Class";   # Nope
```

Dedičnosť: `UNIVERSAL`, `isa`

Každý menný priestor (a teda aj trieda) v Perli má prapredka `UNIVERSAL`.

Z neho dedí metódy napr.

`UNIVERSAL::can($ns, $method)`
: Zistí, či trieda `$ns` pozná metódu `$method`.

`UNIVERSAL::isa($ref, $class)`
: Zistí, či trieda alebo objekt `$ref` dedí z `$class`.

V oboch prípadoch sa testujú aj predkovia v `@ISA`.

Používajte vždy ako `$ref->can("foo")` alebo `$ref->isa("Class")`, nikdy nie `UNIVERSAL::isa($ref, "foo")`.

Dedičnosť: ‹UNIVERSAL›, ‹isa›

Ako zistíme, či je objekt inštanciou nejakej triedy?

```
$object->isa("Some::Class");
```

Čo ak by ‹$object› náhodou nebol objekt, ale obyčajná referencia?

```
Can't call method "isa" on unblessed reference at …
```

Dedičnosť: ‹UNIVERSAL›, ‹isa›

Ako zistíme, či je objekt inštanciou nejakej triedy?

Správne riešenie do Perl ‹v5.32›:

```
blessed($object) && $object->isa("Some::Class");
```

Experimentálny operátor ‹isa› od Perl ‹v5.32› (stabilný od ‹v5.36›):

```
# use experimental 'isa';       # Perl v5.32 to v5.34

use v5.36;
$object isa Some::Class
```

Preťažovanie operátorov

Preťažovanie operátorov

Preťažovanie operátorov (*operator overloading*) umožňuje zmeniť správanie operátorov pre objekty.

```
use overload HASH;
```

Pragma, ktorá v triede preťaží operátory uvedené v kľúčoch.
Metóda, ktorá operátor implementuje, je uvedená ako hodnota.

Všetky metódy pre operátory sa očakávajú so signatúrou:

```
sub NAME($lhs, $rhs, $swap);
```

`$rhs`	Pravá strana; `undef` pre unárne operácie.
`$swap`	Indikuje, či došlo k výmene operandov.

Preťažovanie operátorov

Perl sa snaží implementáciu operátorov čo najviac uľahčiť:

- Prvý argument je vždy objekt.
  V nepriaznivej situácii, napr. `7 + $x`, operandy prehodí a zavolá funkciu s pravdivou hodnotou `$swap`.
- Niektoré operátory zvládne vygenerovať sám, napríklad
  - `A - B` umožní vygenerovať `A -= B`, `A--` aj `--A`
  - `A <=> B` umožní vygenerovať všetky relačné operátory

Preťažovanie operátorov

```
package Complex;

use overload '-' => \&_minus;

sub _minus($lhs, $rhs, $swap) { ... };

my $a = Complex->new(0, 1);
my $b = Complex->new(1, 0);

$a - $b;        # _minus($a, $b, '')
$a -  7;        # _minus($a,  7, '')
10 - $a;        # _minus($a, 10,  1)
$a -= 14;       # _minus($a, 14, undef)
$a--;           # _minus($a,  1, undef)
```

Preťažovanie operátorov

Je možné preťažiť aj niektoré operácie, ktoré nemajú priamo operátor, napr.

- `""` Interpolácia hodnoty do reťazca.
  Treba použiť jednoduché úvodzovky, napr. `use overload '""' => ...`
- `0+` Použitie hodnoty v číslenom kontexte.
- `bool` Použitie hodnoty v logickom kontexte.

Preťažovanie operátorov: Priradenie

Majme nasledujúci kód:

```
sub magic($a) {
    my $b = $a;
    $b++;
    return $a + $b;
}

say magic(2);           # 5
```

Čo sa stane, ak namiesto `5` do funkcie pošleme objekt s aritmetickými operátormi?

Preťažovanie operátorov: Priradenie

Zavolajme funkciu znova, tentokrát s objektom:

```
sub magic($a) {
    my $b = $a;          # ‹$a› and ‹$b› are references to the same ‹Number›
    $b++;
    return $a + $b;
}

say magic(Number->new(2));
```

Vypíše program 5 alebo 6?

Preťažovanie operátorov: Kopírovací konštruktor

Predchádzajúca ukážka bude fungovať ako s číslom a vypíše 5.

Prečo? **Pretože inak to nedáva zmysel.**

Perl sa snaží, aby sa objekty s aritmetickými operátormi dali používať rovnako ako čísla bez toho, aby programátor musel upravovať kód.

To dosiahne **kopírovacím konštruktorom**:
Ak má objekt preťažené aritmetické operácie, Perl tesne pred vykonaním mutátora (napr. `$a += $b`) *transparentne* vyrobí kópiu objektu na ľavej strane.

Preťažovanie operátorov: Kopírovací konštruktor

Kopírovací konštruktor si Perl odvodí sám.
Dá sa však preťažiť kľúčom `=>` pre `overload`.

Kopírovací konštruktor sa používa len pre aritmetické mutátory.
Toto **nie je** operátor priradenia, `$a = $b` preťažiť nejde.

Perl hovorí preťažovaniu matematických operátorov „mathemagic“.

Špeciality

Pripomenutie z minula

‹AUTOLOAD› je špeciálna funkcia v module, ktorá sa zavolá, ak sa v module volá neexistujúca funkcia.

Triedy môžu ‹AUTOLOAD› využiť na generovanie prístupových metód alebo na *data-driven* generovanie funkcionality.

```
sub Person::AUTOLOAD($self) {
    my $attribute = $Person::AUTOLOAD =~ s/.*:://r;

    die "No such attribute: Person::$attribute"
        if !exists $self->{$attribute};

    return $self->{$attribute};
}
```

Ak definujete metódu `‹AUTOLOAD›`, definujte aj `‹DESTROY›`!

V opačnom prípade Perl zavolá `‹AUTOLOAD›` na konci života objektu.

K operátoru `->`

Videli sme, že prekrytú metódu vieme zavolať upresnením menného priestoru prekrytej metódy, napr. `$object->Parent::method`.

Perl však nijak nevynucuje, že metóda na pravej strane pochádza od predka:

```
@Cat::ISA = qw{Animal};
sub Cat::meow($self) { ... };

@Human::ISA = qw{Animal};
sub Human::imitate_cat($self) {
    say $self isa Cat;  # Prints nothing (false)
    $self->Cat::meow;   # Calls <Cat::meow($self)>, no problem
}
```

K operátoru ‹->›

Syntax ‹Class->method› nie je nednoznačná, ak ‹Class› existuje ako funkcia.

Obvyklé Perl konvencie pomenúvajú funkcie ‹underscore_case› a triedy ‹CamelCase›. Dodržujte ich!

```
sub Class() { "Troll" }

Class->method;          # ‹Troll::method›
Class()->method;        # ‹Troll::method›
'Class'->method;        # ‹Class::method›
Class::->method;        # ‹Class::method›
```

Nepriame volanie metód

Táto vlastnosť je od `v5.36` vo východzom stave vypnutá

```
use v5.36;
use feature qw{indirect};

my $song = new File::MP3 "Highway to Hell.mp3"; # ‹File::MP3->new("…");›
play $song;                                     # ‹$song->play›
```

Táto syntax sa dá nájsť v starších kódoch.
V modernom Perli by sa nemala používať, pretože

- je ťažšia na čítanie a pochopenie,
- môže mať nejednoznačný význam (čo ak je `File::MP3` funkcia?).

Globálna deštrukcia

Na konci programu beží *Globálna destrukcia*, ktorá ukončí život všetkých globálnych objektov.

Ak objekt `A` drží referenciu na `B`, pri globálnej deštrukcii nie je garantované, v akom poradí sa zrušia (špeciálne ak sú odkazy cyklické).

Globálna deštrukcia

Takýto stav môžeme ošetriť minimálne dvoma spôsobmi:

1. Overíme, že referencia je stále platná:

```
sub A::DESTROY($self) {
    return unless defined $self->{component};
    # ...
}
```

2. Deštruktor preskočíme, ak beží globálna deštrukcia

```
sub B::DESTROY($self) {
    return if ${^GLOBAL_PHASE} eq 'DESTRUCT';
    # ...
}
```

OOP: Moduly

Perl OOP často slúži ako základ zložitejším OOP prístupom.
Tieto sú implementované v (neštandardných) moduloch.

‹`Class::Tiny`› — Miniatúrny modul na generovanie prísupových metód.

```
package Person;
use Class::Tiny qw{name age brainslug};

my $person = Person->new(name => 'Phillip J. Fry', age => 25);
say $person->name;      # Getter
$person->brainslug(1);  # Setter
```

`Object::Pad` — Vlastná syntax pre objekty a metódy

```
class Person {
    field $nick: param;

    method speak($what) {
        say "$nick says: $what";
    }
}

my $knight = Person->new(nick => 'Knight');
$knight->speak('Ni!');          # Knight says: Ni!
```

‹Moose› — Ťažkotonážny objektový systém

```
package Person;
use Moose;
extends 'Creature';              # Inherits ‹Creature::shriek›

subtype 'NonEmptyStr' as 'Str'
    where { length $_ > 0 };

has 'name' => (isa => 'NonEmptyStr', is => 'rw');

after 'shriek' => sub ($self) {
    say "@{[ $self->name ]} has shrieked";
};
```

Dedičnosť (inheritance) *Vertikálny* vzťah tried

Rysy (traits) *Horizontálny* vzťah tried

Rysy môžu byť v závislosti od typového systému:

- štrukturálne — prekladač overí, že trieda má všetky požadované metódy v mieste, kde sa použije (napr. `concept` v C++)
- nominálne — trieda musí deklarovať, že spĺňa rys (napr. `trait` v Rust)

`Role::Tiny` — Nominálne rysy pre Perl

```
package Identifiable;
use Role::Tiny;

sub id($) { ... };

package Person;
use Role::Tiny::With;

with 'Identifiable';
sub id($self) { ... };

my $person = Person->new(…);
say Role::Tiny::does_role($person, 'Identifiable');     # 1
```