PB173 Perl

10 Objekty

Roman Lacko <xlacko1@fi.muni.cz>

Obsah

Objekty

Objektovo-orientované programovanie v podaní Perlu

Trieda

≈ Balík (`package`).

Objekt

≈ Hodnota (typicky hash) priradená k balíku kľúčovým slovom `bless`.

Atribúty

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

Metóda

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

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.

Bez použitia `CLASSNAME` použije názov aktuálneho menného priestoru. Nepoužívať, rozbije sa pri dedičnosti (viď príklady).

Objekty: Kľúčové slovo `bless`

`bless` síce očakáva referenciu, ale v skutočnosti pracuje nad odkazovanou hodnotou. Na referenciu ani premennú efekt nemá.

```
my %value;
my $ref1 = \%value;
bless $ref, 'Class';    # ‹$ref1› is now an object.

my $ref2 = $ref1;       # ‹$ref2› refers to the same object as ‹$ref1›.
my $ref3 = \%value;     # Ditto.

my $ref1 = undef;       # ‹$ref1› no longer refers to an object.
                        # ‹$ref2› and ‹$ref3› still do, though.
```

Objekty: Operátor `->`

Ak vo výraze `X->m` je `X` názov balíka alebo objekt, Perl zavolá metódu.

Metóda je obyčajná funkcia, ktorá dostane ľavý operand `->` ako prvý argument. Nasledujúce volania sú teda ekvivalentné:

- Volanie metódy na triede:

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

- Volanie metódy na objekte:

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

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

Objekty: Konštruktor

- Obyčajná metóda, ktorá zavolá `bless`.
- Obvykle názov `new()`, Perl však nijak nevynucuje.
- Metóda triedy, tj. zavolá sa ako `CLASS->new()`.

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

Ako podkladový dátový „typ“ sa typicky používa hash:

- Pomenované „atribúty“.
- Jednoduchá rozšíriteľnosť.

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("John", "Doe", 27);
say $person->{name};            # John
```

Obyklé konvencie ako v Pythone:

- K atribútom objektu mimo triedy sa nepristupuje.
  Alternatívne: Atribúty začínajúce _ sa považujú za privátne.
- Metódy začínajúce _ sa považujú za privátne.
- Všetko ostatné je verejné.

Objekty: Metódy

Obyčajné funkcie, ktoré očakávajú inštanciu ako prvý argument.
(Podobne ako v jazyku Python.)

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

- `$class` pre metódy triedy (`CLASS->method()`).
- `$self` pre metódy objektov (`$obj->method()`).
- `$ref` (zriedkavo) ak metóda nerozlišuje objekt a triedu.

```
sub Person::full_name($self) {
    return join ' ', $self->@{'name', 'surname'};
}

my $rick = Person->new("Rick", "Sanchez", 70);
say $rick->full_name;                    # Rick Sanchez
say Person::full_name($rick);            # (Technically) the same.
```

Objekty: Prístupové metódy

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

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

Používajú sa typicky dva spôsoby implementácie:

- Samostatné metódy, napr. `get_ATTRIBUTE()` a `set_ATTRIBUTE()`.
- Jedna metóda, ktorá očakáva nepovinný argument.

```
sub Person::age($self, $age = undef) {
    return $self->{age} if @_ == 1;     # Getter.

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

Objekty: Deštruktor

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

To platí aj pre objekty. Ak majú metódu `DESTROY( )`, zavolá sa.

Perl garantuje, že sa `DESTROY( )` zavolá na konci rozsahu poslednej premennej.

Týmto mechanizmom je teda možné implementovať RAII.

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

```
sub DESTROY($self) {
    # Release resources here.
}
```

Objekty: Deštruktor

```
sub File::Lock::new($class, $filename) {
    my $self = bless { ... }, $class;
    flock $self->{fh}, LOCK_EX
        or die "$name: lock: $!";
    return $self;
}

sub File::Lock::DESTROY($self) {
    flock $self->{fh}, LOCK_UN
        or warn "$name: unlock: $!";
}

{
    my $lock = File::Lock->new("/tmp/something.lock");
    ...         # Only we now have the lock here.
}   # File::Lock::DESTROY($lock) gets called here.
```

Dedičnosť

Ak trieda obsahuje globálnu (*package*) premennú `@ISA`,
zdedí metódy z vymenovaných tried.

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

package Bar;
our @ISA = qw(Foo);
# ‹Bar->a› and ‹Bar->b› are inherited from ‹Foo›
```

`@ISA` má efekt len na rezolúciu *metód*, `Bar::a` fungovať nebude!

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, SUPER

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) {
    # Create an instance of ‹Parent›.
}

@Child::ISA = qw(Parent); # Not using ‹parent› to keep the example short.
sub Child::new($class, @args) {
    # …?
}

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

Dedičnosť: Prekrytie metód, SUPER

Prekryté metódy môžeme zavolať plným menom za `->`:

```
package 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);
```

Namiesto konkrétnej triedy však môžeme použiť aj (pseudo)triedu SUPER:

```
$class->SUPER::new(@args);      # ‹Parent::new($class, @args)›
```

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 }, __PACKAGE__;    # Or ‹"Parent"›, or nothing.
}
```

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

   Ups, 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`, `blessed`

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
```

Problematické, ak chceme rozlíšiť obyčajné referencie od objektov.

```
use Scalar::Util qw(blessed);
blessed($scalar);
```

- Funkcia `Scalar::Util::blessed` vráti meno triedy objektu.
- Vráti `undef` pre ostatné hodnoty, vrátane bežných referencií.

Dedičnosť: `UNIVERSAL`, `isa`

Ako zistíme, či je objekt inštanciou nejakej triedy?

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

Čo ak `$object` je inštanciou podtriedy `Some::Class`?

```
say ref $object;                          # Some::Subclass
say @Some::Subclass::ISA;                 # Some::Class

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

Dedičnosť: `UNIVERSAL`, `isa`

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

Z neho dedí metódy napr.

`UNIVERSAL::can($ns, $function)`
: Zistí, či menný priestor `$ns` pozná funkciu `$function`.

`UNIVERSAL::isa($ref, $class)`
: Zistí, či trieda alebo objekt `$ref` je inštanciou triedy `$class`.

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

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

Dedičnosť: `UNIVERSAL`, `isa`

Takže znova...
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`

Takže znova...
Ako zistíme, či je objekt inštanciou nejakej triedy?

Typické správne riešenie:

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

Perl od verzie 5.32 má experimentálny operátor `isa`:

```
use experimental 'isa';

$object isa Some::Class;
```

Preťažovanie operátorov

V Perli je možné zmeniť implementáciu niektorých 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);
```

- `$rhs` môže byť `undef` pre unárne operácie.
- `$swap` hovorí operácii, či došlo k prehodeniu operandov.

Preťažovanie operátorov

Perl sa snaží implementáciu operátorov čo najviac uľahčiť:

- Ak je len pravý operand objekt (napr. `7 + $x`), tak operandy prehodí a `$swap` nastaví na 1. Prvý argument bude teda vždy objekt.
- Niektoré operátory zvládne vygenerovať sám, napr.
  - Z `A - B` vygeneruje `A -= B`, `A--` aj `--A`.
  - Z `A <=> B` vygeneruje všetky relačné operátory.

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

- `""` pre interpoláciu hodnoty do reťazca.
  (doslova `use overload '""' => ...`)
- `0+` pre použitie v kontexte čísla.
- `bool` pre použitie v kontexte logickej hodnoty.

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)
```

Odbočka: Kopírovací konštruktor

Majme nasledujúci kód:

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

say magic(5);           # 11
```

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

Odbočka: Kopírovací konštruktor

Zavolajme funkciu znova, tentokrát s objektom:

```
sub magic($a) {
    my $b = $a;          # ‹$a› and ‹$b› are the same reference…
    $b++;                # … so we would change ‹$a› here as well…
    return $a + $b;      # … and we would get ‹$b› + ‹$b›.
}

say magic(Number->new(5));      # 12 (… OR IS IT?)
```

Takéto správanie by spôsobilo, že preťažené aritmetické operátory by boli nepoužiteľné, alebo by funkcie museli rozlišovať bežné skaláry a objekty.

Odbočka: Kopírovací konštruktor

V skutočnosti bude predchádzajúca ukážka fungovať.

Perl sa totiž znova snaží dávať zmysel tak, aby sa objekty s aritmetickými operátormi mohli používať ako čísla bez toho, aby programátor musel tieto hodnoty rozlišovať.

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.

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, tj. `$a = $b` preťažiť nejde.