Lekcija 5 - OO Perl

Contents

LearningObject

L5: OO Perl

Uvod *
Ovo poglavlje predstavlja uvod u oblast objektno orjentisanog Perl jezika.

OO modules

Cilj

- Upoznavanje sa Perl odulima

Modules

Modules

Moduli

Modul koji opisuje klasu mora da ima poseban podprogram za kreiranje objekta koji se zove konstruktor.

Često se konstruktir zove new, ali se i Create koristi kod Win32 klasa.

Konstruktir kreira novi objekat i vraća reference na njega. Ova referenca je regularna skalarna varijabla,losim što što referencira neke osnovne objekte koje znaju kojim klasama pripadaju.

U programima se ova referenca koristi za manipulaciju objektom.

Metode (engl. method) su podprogrami koji očekuju referencu na objekat kao prvi argument na primer kao:

```
sub in_class {
my $class = shift; # object reference
my ($this, $that) = @_; # params
}
```

Metode se mogu pozivati kao :

PackageName->constructor(*args*)->method_name(*args*);

ili:

```
$object = PackageName->constructor(args);
$object->method_name( args );
```

Objekti imaju poseban skup dostupnih metoda unutar svoji klasa, ali mogu naslediti metode klase svojih roditelja (eng. parent class), ako postoji.

Objekti se uništavaju kada nestane posljednja refrenca. To se može kontrolisati korišćenjem metodaDESTROY.

DESTROY metod treba da se definiše negdje u klasi

Ovaj metod se poziva eksplicitno, biće pozvan kada bude vreme za to.

Reference na objekte sadržane u tekućem objektu biće oslobođene kada se oslobodi tekući objekat.

U većini slučajeva ne potrebe za eksplicitno uništenje objekata, ali ima slučajeva kada je to potrebno, na primer kada više nije potreban socket objekat.

Primer

Modul Shape

Shape.pm

```
package Shape;

sub new {
my $class = shift;
my $self = {
color => 'black',
length => 1,
width => 1,
};
return bless $self, $class;
}

1;
```

Draw.pl

```
use strict;
use warnings;
use feature qw/say/;
use Shape;

# create a new Shape object
my $shape = Shape->new;

# print the shape object attributes

say $shape->{color};
say $shape->{length};
```

```
say $shape->{width};
```

How to install CPAN modules

http://www.cpan.org/modules/INSTALL.html

What are objects?

Cilj

- Upoznavanja sa Perl objektima

What are objects?

What are objects?

Basically you already know what an object is. Trust your instincts. The book you are reading is an object. The knife and fork you eat with are objects. In short, your life is filled with them.

The question that really needs to be asked is, "What are classes?" You see, all object-oriented techniques use classes to do the real work. A *class* is a combination of variables and functions designed to emulate an object. However, when referring to variables in a class, object-oriented folks use the term *properties*; and when referring to functions in a class, the term *method* is used.

I'm not sure why new terminology was developed for object-oriented programming. Because the terms are now commonplace in the object-oriented documentation and products, you need to learn and become comfortable with them in order to work efficiently.

In this chapter, you see how to represent objects in Perl using classes, methods, and properties. In addition, you look at the definitions of some big words such as *abstraction*, *encapsulation*, *inheritance*, and *polymorphism*.

Following are short definitions for these words. The sections that follow expand on these definitions and show some examples of their use.

Abstraction

Information about an object (its properties) can be accessed in a manner that isolates how data is stored from how it is accessed and used.

Encapsulation

The information about an object and functions that manipulate the information (its methods) are stored together.

Inheritance

Classes can inherit properties and methods from one or more parent classes.

Polymorphism

A child class can redefine a method already defined in the parent class.

Object Syntax

Cilj

- Upoznavanja sa sintaksom Perl objekata

Object Syntax

Object Syntax

Perl uses two forms of syntax for invoking methods on objects. For both types of syntax, the object reference or class name is given as the first argument. A method that takes a class name is called a *class method*, and one that takes an object reference is called an *instance method*.

Class methods provide functionality for the entire class, not just for a single object that belongs to the class. Class methods expect a class name as their first argument. Following this explanation, a constructor is an example of a class method:

```
sub new {
my $self = {};
bless $self;
return $self;
}
```

On the other hand, an instance method expects an object reference as its first argument. An instance method will shift the first argument and use this argument as a reference:

```
sub instance_method {
my $self = shift;
my($one, $two, $three) = @_;
# do stuff
}
```

Here is an example of a constructor creating a new object and returning a reference:

```
$tri = new Triangle::Right (side1 => 3, side2 => 4);
```

This example creates a new right-triangle object and references it with $tri. The parameters are given as a hash-style list. This is common for constructors, as they set initial parameters for an object that is probably just a hash. Now that we have an object, we can invoke some method on it. Suppose Triangle::Right defines a ethod, hypot , that returns the length of the hypotenuse for a given right-triangle object. It would be used like this:

```
$h = hypot $tri;
print "The hypotenuse is: $h.\n";
```

In this particular example, there happens to be no additional arguments to the hypot method, but there could have been.

With the arrow (->) notation, the left side of the arrow must be an object reference or a class name, while the right side of the arrow must be a method defined for that object. Any arguments must follow the method inside of parentheses. For example:

$obj->method(*args*) CLASS->method(*args*)

You have to use parentheses because this form can't be used as a list operator, although the first type of method syntax can.

The examples given above would look like this using the arrow syntax:

```
$tri = Triangle::Right->new(side1 => 3, side2 => 4);
```

```
$h = $tri->hypot();

print "The hypotenuse is: $h.\n";
```

The arrow syntax provides a helpful visual relationship between the object and its method, but both forms of syntax do the same thing. Precedence for the arrow syntax is left to right, exactly the same as the dereferencing operator. This allows you to chain together objects and methods if you want to simplify things. You just have to make sure you have an object to the left of the arrow and a method to the right:

```
%sides = (side1 => 3, side2 => 4);

$h = Triangle::Right->new(%sides)->hypot();

print "The hypotenuse is: $h.\n";
```

In this example, you never assign a variable name to the right-triangle object; the reference is passed directly to the hypot method.

Classes

Cilj

- Upoznavanja sa Perl klasama

Classes

Perl klase (egl. Perl classes)

Before looking at specific examples of object-oriented Perl code, you need to see some generic examples. Looking at generic examples while learning the ***standard*** object-oriented terminology will ensure that you have a firm grasp of the concepts. If you had to learn new Perl concepts at the same time as the object concepts, something might be lost because of information overload.

Classes are used to group and describe object types. Character classes have already been looked with regular expressions A class in the object-oriented world is essentially the same thing. Let's create some classes for an inventory system for a pen and pencil vendor. Start with a pen object. How could you describe a pen from an inventory point of view?

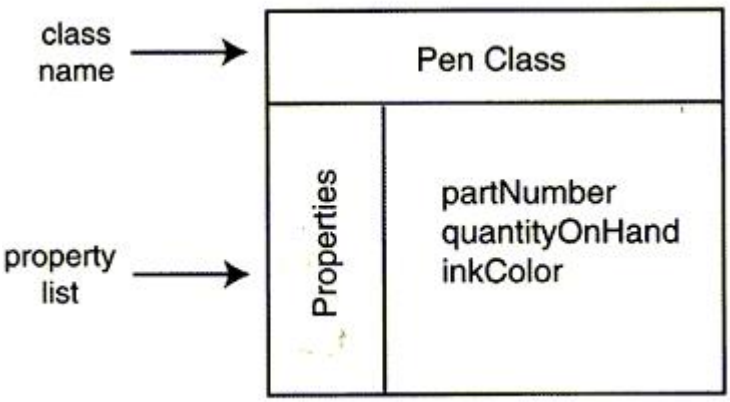
Well, the pen probably has a part number, and you need to know how many of them there are. The color of the pen might also be important. What about the level of ink in the cartridge-is that important? Probably not to an inventory system because all the pens will be new and therefore full.

The thought process embodied in the previous paragraph is called modeling. Modeling is the process of deciding what will go into your objects. In essence, you create a model of the world out of objects.

Note: The terms `object` and `class` are pretty interchangeable. Except that a class might be considered an object described in computer language, whereas an object is just an object.

Objects are somewhat situational dependent. The description of an object, and the class, depends on what needs to be done. If you were attempting to design a school course scheduling program, your objects would be very different than if you were designing a statistics program.

Now back to the inventory system. You were reading about pens and how they had colors and other identifying features. In object talk, these features are called *properties*. Slika 1 shows how the Pen class looks at this stage of the discussion.

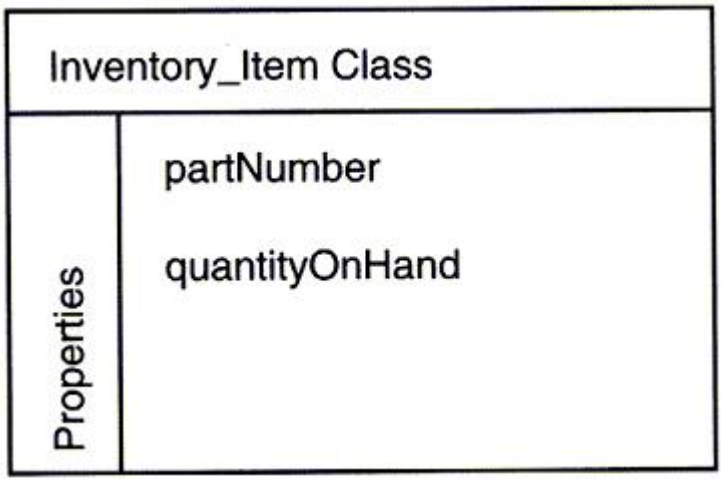


Slika 1 Pen klasa

The Pen Class and its properties

Now that you have a class, it's time to generalize. Some people generalize first. I like to look at the details first and then extract the common information. Of course, usually you'd need several classes before any common features will appear. But because I've already thought this example through, you can cheat a little.

It's pretty obvious that all inventory items will need a part number and that each will have its own quantity-on-hand value. Therefore, you can create a more general class than Pen. Let's call it `Inventory_item`. Slika2 shows this new class.



Slika 2 `Inventory_item` class

Because some of Pen's properties are now also in Inventory_item, you need some mechanism or technique to avoid repetition of information. This is done by deriving the Pen class from Inventory_item. In other words, Inventory_item becomes the parent of Pen..

The relationship between Inventory_item and Pen

You may not have noticed, but you have just used the concept of *inheritance*. The `Pen` class inherits two of its properties from the `Inventory_item` class. Inheritance is really no more complicated than that. The child class has the properties of itself plus whatever the parent class has.

You haven't seen methods or functions used in classes yet. This was deliberate. Methods are inherited in the same way that data is. However, there are a couple of tricky aspects of using methods that are better left for later. Perhaps even until you start looking at Perl code.

Note Even though you won't read about methods at this point in the chapter, there is something important that you need to know about inheritance and methods. First, methods are inherited just like properties. Second, using inherited methods helps to create your program more quickly because you are using functionality that is already working. Therefore, at least in theory, your programs should be easier to create.

Example

```
#!/usr/bin/perl

# Number.pm, a number as an object

package Number; # This is the " Class"

sub new # constructor, this method makes an object that belongs to class Number

{

my $class = shift; # $_[0] contains the class name

my $number = shift; # $_[1] contains the value of our number

# it is given by the user as an argument

my $self = {}; # the internal structure we'll use to represent

# the data in our class is a hash reference

bless( $self, $class ); # make $self an object of class $class

$self->{num} = $number; # give $self->{num} the supplied value

# $self->{num} is our internal number

return $self; # a constructor always returns an blessed()

# object

}

sub add # add a number to our object's number

{

my $self = shift; # $_[0] now contains the object on which the method was called (executed on)

my $add = shift; # number to add to our number

$self->{num} += $add; # add

return $self->{num};

# by returning our new number after each operation we could see

# its value easily, or we could use the dump() method which could

# show us the number without modifying its value.

}

sub subtract # subtract from our number

{

my $self = shift; # our object's internal data structure, as above

my $sub = shift;
```

```
$self->{num} -= $sub;
return $self->{num};
}

sub change # assign new value to our number
{
my $self = shift;
my $newnum = shift;
$self->{num} = $newnum;
return self->{num};
}

sub dump # return our number
{
my $self = shift;
return $self->{num};
}

1; # this 1; is neccessary for our class to work
```

Example

```
package Person;
sub new
{
my $class = shift;
my $self = {
_firstName => shift,
_lastName => shift,
_ssn => shift,
};

# Print all the values just for clarification.
print "First Name is $self->{_firstName}\n";
print "Last Name is $self->{_lastName}\n";
print "SSN is $self->{_ssn}\n";
bless $self, $class;
return $self;
}
$object = new Person( "Paul", "John", 23234345)
```

Example

In general, bless associates an object with a class.

```
package MyClass;
my $object = { };
bless $object, "MyClass";
```

Example

Person.pl

```
#!/usr/bin/perl
package Person;

sub new
{
my $class = shift;
my $self = {
_firstName => shift,
_lastName => shift,
_ssn => shift,
};
# Print all the values just for clarification.
print "First Name is $self->{_firstName}\n";
print "Last Name is $self->{_lastName}\n";
print "SSN is $self->{_ssn}\n";
bless $self, $class;
return $self;
}
sub setFirstName {
my ( $self, $firstName ) = @_;
$self->{_firstName} = $firstName if defined($firstName);
return $self->{_firstName};
}

sub getFirstName {
my( $self ) = @_;
return $self->{_firstName};
}
1;
```

test.pl

```
#!/usr/bin/perl
use Person;

$object = new Person( "Nenad", "Zoran", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Petar." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "Before Setting First Name is : $firstName\n";
```

Display

```
First Name is Nenad
Last Name is Zoran
SSN is 23234345
Before Setting First Name is : Nenad
Before Setting First Name is : Petar.
```

Objects in Perl

Cilj

- Upoznavanja sa Perl objektima

Objects in Perl

Objects in Perl

References will play a large role in the rest of the chapter and are critical to understanding how classes are used. You specifically need to remember that the `{ }` notation indicates an anonymous hash. Armed with this knowledge and the object-oriented terminology from the first part of this chapter, you are ready to look at real Perl objects.

The following listing shows how the `inventory_item` class could be defined in Perl:

Start a new class called `Inventory_item`.

The `package` keyword is used to introduce new classes and namespaces.

Define the `new()` function. This function is responsible for constructing a new object. The first parameter to the `new()` function is the class name (`Inventory_item`).

The `bless()` function is used to change the data type of the anonymous hash to `$class` or `Inventory_item`.

Because this is the last statement in the method, its value will be returned as the value of the function -- using the return statement to explicitly return a value would clutter the code in this situation.

An anonymous hash is used to hold the properties for the class. For the moment, their values are undefined.

Switch to the package called `main`. This is the default place for variables and code to go (technically, this is called a namespace).

If no classes are defined in your script, then this line is not needed.

Assign an instance of the `Inventory_item` class to the `$item` variable.

The Perl code for defining the `Inventory_item` Class is:

```
package Inventory_item;

sub new {
my($class) = shift;
bless {
"PART_NUM" => undef,
"QTY_ON_HAND" => undef
}, $class;
}

package main;

$item = Inventory_item->new();
```

The first line, `package Inventory_item`; says two things, depending on if you are thinking in terms of objects or in terms of Perl. When considering objects, it begins the definition of a class. When considering Perl, it means that a specific namespace will be used.

A namespace is used to keep one set of names from interfering with another. For example, you can have a variable named bar and a function called `bar`, and the names will not conflict because variables and functions each have their own namespace.

The `package` keyword lets you create your own namespace. This lets you create more than one function called `new()` as long as each is in its own package or namespace. If you need to refer to a specific function in a specific namespace, you can use `Inventory_item->new`,`Inventory_item::new`, or `Inventory_item'new`. Which notation you use will probably depend on your background.

Object oriented folks will probably want to use the -> notation.

The second line, sub new, starts the definition of a function. It has become accepted practice in the object-oriented world to construct new objects with the `new()` method. This is called the class *constructor*. This might be a good time to emphasize that the class definition is a template. It's only when the `new()` function is called that an object is created or *instantiated*. Instantiation means that memory is allocated from your computer's memory pool and devoted to the use of this specific object.

The `new()` function normally returns a reference to an anonymous hash. Therefore, the`new()` function should never be called unless you are assigning its return value to a variable. If you don't store the reference into a scalar variable for later use, you'll never be able to access the

anonymous hash inside the object. For all intents and purposes, the anonymous hash *is* the object.

Note Not all objects are represented by hashes. If you need an object to emulate a gas tank, perhaps an anonymous scalar would be sufficient to hold the number of gallons of gas left in the tank. However, you'll see that working with hashes is quite easy once you learn how. Hashes give you tremendous flexibility to solve programming problems.

There is nothing magic about the new function name. You could call the function that creates new objects `create()` or `build()` or anything else, but don't. The standard is `new()`, and everyone who reads your program or uses your classes will look for a `new()` function. If they don't find one, confusion might set in. There are so few standards in the programming business. When they exist, it's usually a good idea to follow them.

The `bless()` function on the third line changes the data type of its first parameter to the string value of its second parameter. In the situation shown here, the data type is changed to the name of the package, `Inventory_item`. Using `bless()` to change the data type of a reference causes the`ref()` function to return the new data type.

Note The `bless()` function is used without using parentheses to surround the parameters.

Embedded inside the `bless()` function call is the creation of an anonymous hash that holds the properties of the class. The hash definition is repeated here for your convenience:

{

"PART_NUM" => undef,

"QTY_ON_HAND" => undef

};

Nothing significant is happening here that you haven't seen before. Each entry in the hash is a different property of the class. For the moment, I have assigned the undefined value to the value part of the entries. Soon you'll see how to properly initialize them.

After the new() function is defined, there is another package statement: package main;

There is no object-oriented way to interpret this statement. It simply tells Perl to switch back to using the main namespace. Don't be fooled into thinking that there is a main class somewhere. There isn't.

A note of Caution: While you could create a main class by defining the new() function after the package main; statement, things might get to be confusing, so don't do it!

The last statement in the file is really the first line that gets executed. Everything else in the script has been class and method definitions. $item = Inventory_item->new();

By now, you've probably guessed what this statement does. It assigns a reference to the anonymous hash to $item. You can dereference $item in order to determine the value of the entries in the hash. If you use the ref() function to determine the data type of $item, you find that its value isInventory_item.

Here are some key items to remember about objects in Perl:

All objects are anonymous hashes: While not strictly true, perhaps it should be. Also, most of the examples in this book follow this rule. This means that most of the `new()` methods you see return a reference to a hash.

`bless()` changes the data type of the anonymous hash: The data type is changed to the name of the class.

The anonymous hash itself is blessed: This means that references to the hash are not blessed. This concept is probably a little unclear. I had trouble figuring it out myself. The next section clarifies this point and uses an example.

Objects can belong to only one class at a time: You can use the `bless()` function to change the ownership at any time. However, don't do this unless you have a good reason.

The **–>** operator is used to call a method associated with a class: There are two different ways to invoke or call class methods:

$item = new Inventory_item;

or

$item = Inventory_item->new();

Both of these techniques are equivalent, but the -> style is preferred by object-oriented developers.

Hashes and Passing the Reference

Cilj

- Upoznavanja sa heševima i metodom slanja referenci

Hashes and Passing the Reference

Hashes and Passing the Reference

The `ref()` function returns either the undefined value or a string indicating the parameter's data type (`SCALAR`, `ARRAY`, `HASH`, `CODE`, or `REF`). When classes are used, these data types don't provide enough information.

This is why the `bless()` function was added to the language. It lets you change the data type of any variable. You can change the data type to any string value you like. Most often, the data type is changed to reflect the class name. It is important to understand that the variable itself will have its data type changed. The following lines of code should make this clear (`bless.pl`):

`bless.pl`

```
$foo = { };

$fooRef = $foo;

print("data of \$foo is " . ref($foo) . "\n");

print("data of \$fooRef is " . ref($fooRef) . "\n");

bless($foo, "Bar");

print("data of \$foo is " . ref($foo) . "\n");

print("data of \$fooRef is " . ref($fooRef) . "\n");
```

This program displays the following:

```
data of $foo is HASH

data of $fooRef is HASH

data of $foo is Bar

data of $fooRef is Bar
```

After the data type is changed, the `ref($fooRef)` function call returns `Bar` instead of the old value of `HASH`. This can happen only if the variable itself has been altered. This example also shows that the `bless()` function works outside the object-oriented world.

Initializing Properties

Cilj

- Upoznavanja sa metodom inicijalizacije karakteristika (engl. properties)

Initializing Properties

Initializing Properties

You now know how to instantiate a new class by using a `new()` function and how to create class properties (the class information) with undefined values. Let's look at how to give those properties some real values. You need to start by looking at the `new()` function from the first example `init.pl`:

`init.pl`

```
sub new {
my($class) = shift;
bless {
"PART_NUM" => undef,
"QTY_ON_HAND" => undef
}, $class;
}
```

The `new()` function is a *static* method. Static methods are not associated with any specific object. This makes sense because the `new()` function is designed to create objects. It can't be associated with an object that doesn't exist yet, can it?

The first argument to a static method is always the class name. Perl takes the name of the class from in front of the -> operator and adds it to the beginning of the parameter array, which is passed to the `new()` function.

If you want to pass two values into the `new()` function to initialize the class properties, you can modify the method to look for additional arguments as in the following `init2.pl`:

`init2.pl`

```
sub new {
my($class) = shift;
my($partNum) = shift;
my($qty) = shift;
bless {
"PART_NUM" => $partNum,
"QTY_ON_HAND" => $qty
}, $class;
}
```

Each parameter you expect to see gets shifted out of the parameter array into a scalar variable. Then the scalar variable is used to initialize the anonymous hash.

You invoke this updated version of `new()` by using this line of code:

```
$item = Inventory_item->new("AW-30", 1200);
```

While this style of parameter passing is very serviceable, Perl provides for the use of another technique: passing named parameters.

Using Named Parameters in Constructors

Cilj

- Upoznavanja sa korišćenjem imenovanih parametara u konstruktorima

Using Named Parameters in Constructors

The concept of using named parameters has been quickly accepted in new computer languages. I was first introduced to it while working with the scripting language for Microsoft Word. Rather than explain the technique in words, let's look at an example in code, as shown below:

Start a definition of the *`Inventory_item`* class.

Define the constructor for the class.

Get the name of the class from the parameter array.

Assign the rest of the parameters to the `%params` hash.

Bless the anonymous hash with the class name. Use `%params` to initialize the class properties.

Start the `main` namespace.

Call the constructor for the `Inventory_item` class.

Assign the object reference to `$item`

Print the two property values to verify that the property initialization worked.

The code listing `invent.pl` is as follows:

```
package Inventory_item;
sub new {
my($class) = shift;
my(%params) = @_;
bless {
"PART_NUM" => $params{"PART_NUM"}, "QTY_ON_HAND" => $params{"QTY_ON_HAND"}
}, $class;
}
package main;
$item = Inventory_item->new( "PART_NUM" => "12A-34", "QTY_ON_HAND" => 34);
print("The part number is " . %{$item}->{'PART_NUM'}"\n");
print("The quantity is " . %{$item}->{'QTY_ON_HAND'} "\n");
```

One key statement to understand is the line in which the `new()` function is called:

$item = Inventory_item->new("PART_NUM" => "12A-34", "QTY_ON_HAND" => 34);

This looks like an associative array is being passed as the parameter to `new()`, but looks are deceiving in this case. The `=>` operator does exactly the same thing as the comma operator. Therefore, the preceding statement is identical to the following:

$item = Inventory_item->new("PART_NUM", "12A-34", "QTY_ON_HAND", 34);

Also, a four-element array is being passed to `new()`.

The second line of the `new()` function, `my(%params) = @_;` does something very interesting. It takes the four-element array and turns it into a hash with two entries. One entry is for `PART_NUM`, and the other is for `QTY_ON_HAND`.

This conversion (array into hash) lets you access the parameters by name using `%params`. The initialization of the anonymous hash-inside the`bless()` function-takes advantage of this by using expressions such as `$params{"PART_NUM"}`.

I feel that this technique helps to create self-documenting code. When looking at the script, you always know which property is being referred to. In addition, you can also use this technique to partially initialize the anonymous hash. For example,

$item = Inventory_item->new("QTY_ON_HAND" => 34);

gives a value only to the `QTY_ON_HAND` property; the `PART_NUM` property will remain undefined. You can use this technique with any type of function, not just constructors.

Inheritance in Perl Style

Cilj

- Upoznavanja sa nasleđivanjem u Perlu

Inheritance in Perl Style

Inheritance in Perl Style

Object-oriented programming sometimes involves inheritance. Inheritance simply means allowing one class called the Child to inherit methods and attributes from another, called the Parent, so you don't have to write the same code again and again. For example, we can have a class Employee which inherits from Person. This is referred to as an "isa" relationship because an employee is a person. Perl has a special variable, @ISA, to help with this. **@ISA governs (method) inheritance.**

Example

Employee.pm

#!/usr/bin/perl

package Employee;

use Person;

use strict;

our @ISA = qw(Person); # inherits from Person

main.pl

#!/usr/bin/perl

use Employee;

```
$object = new Employee( "Nenad", "Zoran", 23234345);

# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Rade." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";
```

Display

This will produce following result

```
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
```

Before Setting First Name is : Mohd.You already know that inheritance means that properties and methods of a parent class will be available to child classes. This section shows you can use inheritance in Perl.

First, a little diversion. You may not have realized it yet, but each package can have its own set of variables that won't interfere with another package's set. So if the variable `$first` was defined in package A, you could also define `$first` in package B without a conflict arising.

`inher.pl`

```
package A;
$first = "package A";
package B;
$first = "package B";
package main;
print("$A::first\n");
print("$B::first\n");
```

Displays

```
package A
package B
```

Notice that the :: is being used as a scope resolution operator in this example. The -> notation will not work; also, it's okay that -> can't be used because we're not really dealing with objects in this example, just different namespaces.

You're probably wondering what this diversion has to do with inheritance, right? Well, inheritance is accomplished by placing the names of parent classes into a special array called `@ISA`. The elements of `@ISA` are searched left to right for any missing methods. In

addition, the `UNIVERSAL` class is invisibly tacked on to the end of the search list. For example `universal.pl`,

```
package UNIVERSAL;
sub AUTOLOAD {
die("[Error: Missing Function] $AUTOLOAD @_\n");
}
package A;
sub foo {
print("Inside A::foo\n");
}
package B;
@ISA = (A);
package main;
B->foo();
B->bar();
```

Displays:

```
Inside A::foo
[Error: Missing Function] B::bar B
```

Let's start with the nearly empty class `B`. This class has no properties or methods; it just has a parent: the `A` class. When Perl executes `B->foo()`, the first line in the main package, it first looks in `B`. When the `foo()` function is not found, it looks to the `@ISA` array. The first element in the array is`A`, so Perl looks at the `A` class. Because `A` does have a `foo()` method, that method is executed.

When a method can't be found by looking at each element of the `@ISA` array, the `UNIVERSAL` class is checked. The second line of the main package, `B->bar()`, tries to use a function that is not defined in either the base class `B` or the parent class `A`. Therefore, as a last-ditch effort, Perl looks in the `UNIVERSAL` class. The `bar()` function is not there, but a special function called `AUTOLOAD()` is.

The `AUTOLOAD()` function is normally used to automatically load undefined functions. Its normal use is a little beyond the scope of this book. However, in this example, I have changed it into an error reporting tool. Instead of loading undefined functions, it now causes the script to end (via the `die()` function) and displays an error message indicating which method is undefined and which class Perl was looking in. Notice that the message ends with a newline to prevent Perl from printing the script name and line number where the script death took place. In this case, the information would be meaningless because the line number would be inside the `AUTOLOAD()` function.

The next listing below shows how to call the constructor of the parent class. This example shows how to explicitly call the parent's constructor. In the next section, you learn how to use the `@ISA` array to generically call methods in the parent classes. However, because constructors are frequently used to initialize properties, I feel that they should always be called explicitly, which causes less confusion when calling constructors from more than one parent.

This example also shows how to inherit the properties of a parent class. By calling the parent class constructor function, you can initialize an anonymous hash that can be used by the base class for adding additional properties.

It operates as follows:

Start a definition of the *Inventory_item* class.

Define the constructor for the class.

Get the name of the class from the parameter array.

Assign the rest of the parameters to the *%params* hash.

Bless the anonymous hash with the class name.

Use *%params* to initialize the class properties.

Start a definition of the *Pen* class.

Initialize the *@ISA* array to define the parent classes.

Define the constructor for the class.

Get the name of the class from the parameter array.

Assign the rest of the parameters to the *%params* hash.

Call the constructor for the parent class, *Inventory_item*, and assign the resulting object reference to *$self*.

Create an entry in the anonymous hash for the *INK_COLOR* key.

Bless the anonymous hash so that *ref()* will return *Pen* and return a reference to the anonymous hash.

Start the *main* namespace.

Call the constructor for the *Pen* class. Assign the object reference to *$item*.

Note that an array with property-value pairs is passed to the constructor.

Print the three property values to verify that the property initialization worked.

The Perl code to perform the above is `invent2.pl`:

`invent2.pl`:

```
package Inventory_item;
sub new {
my($class) = shift;
my(%params) = @_;
bless {
"PART_NUM" => $params{"PART_NUM"},
"QTY_ON_HAND" => $params{"QTY_ON_HAND"}
}, $class;
}
package Pen;
@ISA = (Inventory_item);
sub new {
my($class) = shift;
my(%params) = @_;
```

```
my($self) = Inventory_item->new(@_);
$self->{"INK_COLOR"} = $params{"INK_COLOR"};
return(bless($self, $class));
}
package main;
$pen = Pen->new(
"PART_NUM" => "12A-34",
"QTY_ON_HAND" => 34,
"INK_COLOR" => "blue");
print("The part number is " . %{$pen}->{'PART_NUM'} . "\n");
print("The quantity is " . %{$pen}->{'QTY_ON_HAND'} . "\n");
print("The ink color is " . %{$pen}->{'INK_COLOR'} . "\n");
```

Display

```
The part number is 12A-34
The quantity is 34
The ink color is blue
```

You should be familiar with all the aspects of this script by now. The line `my($self) = Inventory_item->new(@_);` is used to get a reference to an anonymous hash. This hash becomes the object for the base class.

To understand that calling the parent constructor creates the object that becomes the object for the base class, you must remember that an object *is* the anonymous hash. Because the parent constructor creates the anonymous hash, the base class needs a reference only to that hash in order to add its own properties. This reference is stored in the `$self` variable.

You may also see the variable name **`$this`** used to hold the reference in some scripts. Both `$self` and `$this` are acceptable in the object-oriented world.

Linkovi

Primer 1 - http://www.tutorialspoint.com/perl/perl_oo_perl.htm

Primer 2 - http://ods.com.ua/win/eng/program/Perl5Unleashed/ch5.phtml

Primer 3 - http://wwwacs.gantep.edu.tr/docs/perl-ebook/ch19.htm

Polymorphism

Cilj

- Upoznavanja sa OO karakteristikom polimorfizma

Polymorphism

Polymorphism

Polymorphism, although a big word, is a simple concept. It means that methods defined in the base class will override methods defined in the parent classes. The following small example clarifies this concept `polymorph.pl`:

```
package A;
sub foo {
print("Inside A::foo\n");
}
package B;
@ISA = (A);
sub foo {
print("Inside B::foo\n");
}
package main;
B->foo();
```

Display

```
Inside B::foo
```

The `foo()` defined in class `B` overrides the definition that was inherited from class `A`.

Polymorphism is mainly used to add or extend the functionality of an existing class without reprogramming the whole class.

The prgram below uses polymorphism to override the `qtyChange()` function inherited from `Inventory_item`. In addition, it shows how to call a method in a parent class when the specific parent class name (also known as the *`SUPER`* class) is unknown.

The Perl program is as follows (`polymorph2.pl`):

`polymorph2.pl`

```
package Inventory_item;
sub new {
my($class) = shift;
my(%params) = @_;
bless {
"PART_NUM" => $params{"PART_NUM"},
"QTY_ON_HAND" => $params{"QTY_ON_HAND"}
}, $class;
}
sub qtyChange {
my($self) = shift;
my($delta) = $_[0] ? $_[0] : 1;
$self->{"QTY_ON_HAND"} += $delta;
}
```

```
package Pen;

@ISA = ("Inventory_item");
@PARENT::ISA = @ISA;
sub new {
my($class) = shift;
my(%params) = @_;
my($self) = $class->PARENT::new(@_);
$self->{"INK_COLOR"} = $params{"INK_COLOR"};
return($self);
}

sub qtyChange {
my($self) = shift;
my($delta) = $_[0] ? $_[0] : 100;
$self->PARENT::qtyChange($delta);
}

package main;
$pen = Pen->new(
"PART_NUM"=>"12A-34",
"QTY_ON_HAND"=>340,
"INK_COLOR" => "blue");

print("The data type is " . ref($pen) . "\n");
print("The part number is " . %{$pen}->{'PART_NUM'} . "\n");
print("The quantity is " . %{$pen}->{'QTY_ON_HAND'} . "\n");

print("The ink color is " . %{$pen}->{'INK_COLOR'} . "\n");
$pen->qtyChange();
print("\n");

print("The quantity is " . %{$pen}->{'QTY_ON_HAND'} . "\n");
```

Display

```
The data type is Pen
The part number is 12A-34
The quantity is 340
The ink color is blue
The quantity is 440
```

The first interesting line in the preceding example is `my($delta) = $_[0] ? $_[0] : 1;`. This line checks to see if a parameter was passed to `Inventory_item::qtychange()` and if

not, assigns a value of 1 to `$delta`. This line of code uses the `ternary` operator to determine if `$_[0]` has a value or not. A zero is used as the subscript because the class reference was shifted out of the parameter array and into `$self`.

The next interesting line is `@PARENT::ISA = @ISA;`. This assignment lets you refer to a method defined in the parent class. Perl searches the parent hierarchy (the `@ISA` array) until a definition is found for the requested function.

The `Pen::new()` function uses the `@PARENT::ISA` to find the parent constructor using this line: `my($self) = $class->PARENT::new(@_);`. I don't really recommend calling parent constructors in this manner because the constructor that gets called will depend on the order of classes in the`@ISA` array. Having code that is dependent on an array keeping a specific order is a recipe for disaster; you might forget about the dependency and spend hours trying to find the problem.

However, I thought you should see how it works. Because the `$class` variable (which is equal to `Pen`) is used to locate the parent constructor, the hash will be blessed with the name of the base `Pen` class-one small advantage of this technique. This is shown by the program's output. This technique avoids having to call the `bless()` function in the base class constructor.

By now, you must be wondering where polymorphism fits into this example. Well, the simple fact that both the `Pen` and `Inventory_item` classes have the `qtyChange()` method means that polymorphism is being used.

While

the `Inventory_item::qtyChange()`

method defaults to changing the quantity by one, the `Pen::qtyChange()` method defaults to changing the quantity by 100.

Because the `Pen::qtyChange()` method simply modifies the behavior of `Inventory_item::qtyChange()`, it does not need to know any details about how the quantity is actually changed.

This capability to change functionality without knowing the details is a sign that abstraction is taking place.

Tip The `Inventory_item::qtychange()` notation refers to the `qtyChange()` function in the `Inventory_item` class, and`Pen::qtyChange()` refers to the `qtyChange()` function in the `Pen` class. This notation lets you uniquely identify any method in your script.

L4: OO Perl

Zaključak*
Nakon studiranja sadržaja ovog poglavlja, studenti će steći osnovna znanja iz oblasti objektno orjentisanog Perl jezika.

LearningObject

Perl programski jezik

Perl programski jezik

Kreiranje klasa

U perlu su klase paketi, dok su metode podrutine. Perl nema klase kao sintaksni konstrukt, ali omogućava njihovo definisanje i instanciranje pomoću paketa. Počećemo sa definicijom jednostavne klase Animal koja ima polje "text" i metodu "speak", koja ispisuje taj tekst.

use v5.10.0;

use warnings;

use strict;

package Animal;

sub new {

my $class = $_[0];

my $self = {text => "I am an abstract animal.\n"};

return bless($self, $class);

}

Definisali smo konstruktor (new)

Prvi parametar pri instanciranju je ime klase

Self je hash objekat koji sami kreiramo

Funkcija bless vezuje referencu za određenu klasu, čineći je referencom na instancu te klase

Ovo se može napisati jednostavnije, obzirom da je @_ podrazumevani argument za većinu funkcija:

sub new {

return bless({text => "I am an abstract animal.\n"}, shift);

}

Dodavanje metoda

Na klasu Animal dodaćemo metodu "speak", koja štampa polje "text".

sub speak {

my $self = shift;

print $self->{"text"}

}

Objašnjenje:

za instance metode (metode koje se pozivaju nad instancom objekta), prvi argument je referenca na sam objekat (u Javi bi to bila this referenca), koji se uzima iz liste argumenata funkcijom shift.

$self je referenca (u ovom slučaju referenca na hash koji je vezan za Animal klasu)

referenca $self se mora de-referencirati da bi se pristupilo njenim vrednostima, što se može učiniti operatorom '->' .

$self->{"text"} dereferencira $self, pristupa hash-u sa ključem "text"

vrednost poslednje izvršenog izraza je povratna vrednost metode, u ovom slučaju vrednost polja text.

Nasleđivanje

Za nasleđivanje se u Perl-u koristi klasna promenljiva @ISA, koja je array parent klasa.

```
package Duck;
our @ISA = qw(Animal); # qw(Animal) je ekvavilentno ("Animal")

sub new {
return bless({text => "I am a a duck.\n"}, shift);
}
```

Klasa Duck je sada pod-klasa klase Animal. U podklasi je moguće redefinisati metode iz roditeljske klase.

Finalizacija modula

Da bi Perl učitao modul, potrebno je da poslednja vrednost u modulu bude true. Ovo se obično radi tako što se na poslednju liniju postavi "1;". Perl moduli se čuvaju kao tekstualni fajlovi sa ekstenzijom .pm.

Instanciranje klasa i pozivanje metoda

Komandom "use ImeModula;", učitavate perl modul u trenutni kontekst. Ovo je slično import i require pozivima u Python-u i Ruby-u.

```
# Instanciranje objekata:
use Objects;

my $animal = Animal->new;
$animal->speak; # Ispisuje "I am an abstract animal.\n"

$animal = Duck.new;
$animal->speak; # Ispisuje "I am a a duck.\n"
```

Instanca klase se vraća iz "new" funkcije kao referenca, pa je potrebno dereferencirati je radi poziva metoda ili pristupa poljima. Sama referenca je sklar, pa se čuva u promenljivama prefiksovanim znakom $.

Perl - Klase i objekti

Perl - Klase i objekti

1) Kreirati klase Animal, Cow, Horse i Sheep. Napraviti kompletan Perl program koji ilustrue karakteristike nasleđivanja na osnovu kreiranih klasa.

2) Napisati Vector2D klasu koja podržava operacije vektorskog sabiranja, oduzimanja, skaliranja (množenje i deljenje skalarom) i uzimanja dužine i jediničnog vektora. Svaka vektorska operacija treba da napravi novu instancu vektor klase (tj sabiranje dva vektora vraća novi vektor). Korišćenje overloadovanih operatora je opciono.