Object Oriented Programming with Perl

Yet Another Perl Conference

Amsterdam, August 2 – 4, 2001

Object Oriented Programming with Perl

Preface

Object Oriented Programming with Perl

Object Oriented
Programming with Perl

Johan Vromans

Squirrel Consultancy

<JVromans@Squirrel.NL>

Object Oriented Programming with Perl

This document contains selected sheets from the training “Object Oriented Programming with Perl”.

For the complete materials, please contact Squirrel Consultancy, `<info@squirrel.nl>`.

Object Oriented Programming with Perl

Part I

Object Oriented Programming

It's all about objects, that are instances of classes that supply methods, either directly or via inheritance from base classes. Derived classes can use polymorphism to tailor inherited methods to their specific needs, and abstraction keeps implementation details out of the way.

Despite popular belief, object oriented programming is not something new. The idea of programming in terms of objects goes back to the mid-1960s, when Ole Dahl and Kirsten Nygaart in Norway created the language Simula for simulating physical processes.

Classes

Classes are the heart of the objects system.

Name space with data and routines (behavior).

Usually related to one specific kind of problem.

Objects

An *object* is a data structure that belongs to (is an *instance* of) a class. Every object knows which class it belongs to.

Objects' behavior is supplied by the class.

The object data are called *attributes*.

Methods

The class defines *methods*: behaviors that apply to the class and its instances.

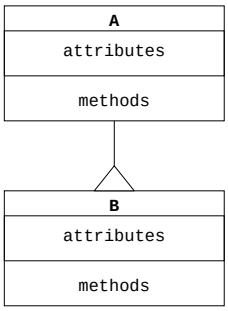
If a method 'does something' related to the object data, it is called an *instance method*.

Otherwise, it is called a *class method*.

A special kind of method are *constructor methods*: these methods create new objects.

Inheritance

A class may *inherit* methods from other classes.
If class B inherits methods from class A, then class A is a *parent* class (*base* class, *superclass*, *generalization*) of class B. Class B is *derived from* (is a *subclass* of, a *specialization* of) class A. Class B has an is-a relation with class A.



Polymorphism

Objects can use inheritance for methods, but they can also supply their own specific implementation by *overriding* methods from the base class. They can even use the base class methods when rolling their own.

Encapsulation

Application programs do not need to know what's inside the object.

The object *encapsulates* its inner guts.

All access to an object should go through methods alone.

Methods that exclusively set and get attribute values are called *attribute accessors*.

Abstraction

Since all access goes through methods, the underlying details are abstracted out.

The *interface* is what is important to the user of the objects, not how it is implemented.

Object Oriented Programming

Why?

Object Oriented Programming with Perl

Part II

Classes

A class is simply a package.

It provides the name space to hold data and subroutines.

Often, a module is used to hold one or more classes.

Objects

An object is simply a referent.

A reference pointing to the referent is used to represent the object.

The association between a package (class) and the referent (object) is made using Perl's builtin `bless` function.

Usually, the class defines one or more methods that return a new object: constructor methods.

Often, one of the constructor methods is called `new` so you can say:

```
$myobj = new MyClass::;
```


Methods

A method is simply a subroutine.

Method invocation is a subroutine call with a twist.

Two ways to invoke a method:

- as a dereference
- using *indirect object* syntax

Method Invocation as a Dereference

```
invocant->method(args, ...)
```

The invocant can be the name of the class (for class methods), or an instance of the class.

The subroutine gets the invocant as its first argument, before the supplied arguments.

Class names are best specified with two trailing colons, e.g. `MyClass::`. This prevents them from being taken for the name of a subroutine.

Method Invocation using Indirect Object Syntax

```
method invocant (args, ...)
method invocant args, ...
```

The subroutine gets the invocant as its first argument, before the supplied arguments.

The indirect object syntax suffers from syntactic ambiguity.

Method Invocation using Subroutine Call

```
method(invocant,args,...)
```

The method name must be fully qualified, since object oriented packages do not normally export subroutines.

We'll see later why this way of invocation is a bad idea.

Inheritance

When a method is invoked on an invocant, the invocant may either provide the method itself (by defining the appropriate subroutine), or inherit the method from one of its base classes.

Perl decides at invocation time which subroutine to call.

Therefore, method invocation is slightly slower than straight subroutine calls.

Also, no prototype checking is possible.

Inheritance

A class can register its base classes by storing their names in a package array `@ISA`.

For example:

```
package Vowel;
use Letter;
our @ISA = qw(Letter);
```

But it's better (and easier) to use the `base` pragma:

```
package Vowel;
use base qw(Letter);
```


UNIVERSAL

A special pre-defined class `UNIVERSAL` is the ultimate ancestor from which all classes implicitly derive.

It provides methods to query whether a class derives from some other class, and if it can perform a certain method.

Methods can be added to class `UNIVERSAL`, but beware! Some other class may not expect to find it.

Polymorphism

Derived classes can add, and override, any methods of the base classes.

When a method is called, perl first looks in the class the object belongs to.

If not found, it searches the class hierarchy.

Looking for Methods

When a method is called, perl first looks in the class the object belongs to.

If not found, it runs a left-to-right, recursive, depth-first search on all base classes.

If still not found, looks in class `UNIVERSAL`.

If still not found, the whole procedure is repeated but this time for a subroutine named `AUTOLOAD`.

If no subroutine could be found, perl raises an exception.

Encapsulation

Perl provides several ways to hide private data from unwanted access.

Abstraction

Perl makes it possible to have all data access go through methods, leaving only the abstracted *interface* to the user of the objects.

Perl does, however, not *enforce* this.

There are ways to make unwanted access hard, if not impossible.

Object Destruction

Perl uses reference counting to keep track of allocated objects.

When an object goes out of scope or otherwise ceases to exist, it will be garbage collected – eventually.

If the object can perform a `DESTROY` method, this is called by Perl when the object is logically removed from the object system.

If it cannot, no special action is taken.

Object Oriented Programming with Perl

Part III

HTML::Head example

Let's set up a simple class that implements HTML heading objects, like `<h1>...</h1>`, `<h2>...</h2>`, and so on.

What we need:

- a constructor to create a heading object,
- a way to specify the level and the text, and
- a method to publish the object, i.e., create HTML text from it.

HTML::Head example – The Class

A class is simply a package.

```
package HTML::Head;

use strict;

1;
```


HTML::Head example – The Constructor

A constructor is simply a method that returns a blessed reference.

A method is simply a subroutine that gets its invocant passed as its first argument.

```
# $obj = HTML::Head::->new;

sub new {
    bless { };
}
```

The object is implemented using a hash.

HTML::Head example – The Attributes

Attributes are the data of the instance. They are stored in the hash.

HTML::Head has two attributes: the *text* and the header *level*.

```
# $obj->set_level(2);

sub set_level {
    my ($self,$level) = @_;
    $self->{level} = $level;
}

# $lvl = $obj->get_level;

sub get_level {
    my $self = shift;
    $self->{level};
}
```


HTML::Head example – The Attributes

For boolean attributes, it is conventional to use set_*attr* and is_*attr*.

```
$obj->set_bold(1);              # on
$obj->set_bold(0);              # off
if ( $obj->is_bold ) { ... }
```

Please supply a sensible default value:

```
$obj->set_bold;                 # on, not off!
```


HTML::Head example – The Constructor (revisited)

If the attributes are stored in a hash, it is convenient and easy to initialize them at construction time:

```
# $obj = HTML::Head::->new(level=>2, text=>"Heading");

sub new {
    shift;              # ignore invocant
    bless { @_ };
}
```

This is much too simple, we need to do some integrity checking here.

HTML::Head example – Other Methods

The publish method produces an HTML string for the instance.

```
sub publish {
    my $self = shift;
    "<h" . $self->get_level . ">" .
      $self->get_text .
        "</h" .  $self->get_level . ">";
}
```

Isn't it a bit overdone to use `get_level` and `get_text` from within the class?

HTML::Head example – Array implementation

Any referent may be used as the object implementation. Hashes are nice, but there's nothing wrong with a plain array.

Let's change the internal implementation from hash to array.

The constructor:

```
sub new {
    shift;              # ignore invocant
    my $obj = [ ];
    my %init = @_;
    $obj->[0] = delete $init{level};
    $obj->[1] = delete $init{text};
    bless $obj;
}
```


Constants

In situations like this, the `constant` pragma may be helpful.

```
use constant ATTR_LEVEL => 0;
use constant ATTR_TEXT  => 1;
```


HTML::Head example – The 'text' attribute

A straightforward change: replace the '`->{text}`' with '`->[ATTR_TEXT]`'.

```
# $obj->set_text("Heading");

sub set_text {
    my ($self, $text) = @_;
    $self->[ATTR_TEXT] = $text;
}

# $text = $obj->get_text;

sub get_text {
    my $self = shift;
    $self->[ATTR_TEXT];
}
```


HTML::Head example – Other Methods

The publish method:

```
sub publish {
    my $self = shift;
    "<h" . $self->get_level . ">" .
      $self->get_text .
        "</h" .  $self->get_level . ">";
}
```

Because we used `get_level` and `get_text`, the other methods do not need to change when the implementation changes.

In general, it is good practice that only the constructor and attribute get/set methods know about the inner details.

Hash versus Array

Arrays are slightly faster, but hard to maintain.

When inheritance gets into play, it becomes even harder.

Hashes are slightly slower, but more flexible.

Perl provides the best of both worlds: the pseudo-hash.

Special pragma's: `fields` and `base`.

HTML::Head example – Pseudo-Hash implementation

The `fields` pragma is used to define the fields that we're going to use.

```
package HTML::Head;

use strict;
use fields qw(level text);
```


HTML::Head example – The Constructor

The `fields::new` method creates a suitable object.

```
# $obj = HTML::Head::->new;

sub new {
    my $invocant = shift;
    my HTML::Head $obj = fields::new($invocant);
    my %init = @_;
    $obj->{$_} = $init{$_} foreach keys %init;
    $obj;
}
```

Illegal values for attribute names are now trapped at run time.

HTML::Head example – The Attributes

```
# $obj->set_level(2);

sub set_level {
    my HTML::Head $self = shift;
    my $level = shift;
    $self->{level} = $level;
}

# $lvl = $obj->get_level;

sub get_level {
    my HTML::Head $self = shift;
    $self->{level};
}
```


HTML::Head example – Other Methods

Again, no changes for the other methods.

```
sub publish {
    my $self = shift;
    "<h" . $self->get_level . ">" .
      $self->get_text .
        "</h" .  $self->get_level . ">";
}
```


Inheritance

As stated earlier, a class can either provide the method itself (by defining the appropriate subroutine), or inherit the method from one of its base classes.

For example, assume we have a base class `HTML::Head`, and two derived classes, `HTML::Head1` and `HTML::Head2`.

`HTML::Head` handles storing the text, while the derived classes handle the publishing.

HTML::Head example with Derived Classes

```
package HTML::Head;
use strict;

sub new {
    shift;              # ignore invocant
    bless { @_ };
}

sub get_text { ... }
sub set_text { ... }

package HTML::Head1;

use base qw(HTML::Head);

sub publish { "<h1>@{[shift->get_text]}</h1>" }
```


Inheritance and Constructors

The `bless` function blesses a reference in the current package. So it will always become an instance of the class that declared the constructor.

But what if the derived class inherits the constructor?

Objects created by the inherited constructor would end up being instances of the base class, not of the derived class.

```
$h1 = HTML::Head1::->new;
```

This calls `HTML::Head::new` to create an `HTML::Head` object, and will not produce an `HTML::Head1` object.

Inheritance and Constructors (cont'd)

Remember, a method always gets its invocant as its first argument.

```
$h1 = HTML::Head1::->new;
```

This calls `HTML::Head::new("HTML::Head1")`.

`bless`' Second Argument

`bless` takes a second argument, the name of the class to bless the object into.

We can use this to create an inheritable constructor.

```
sub new {
    my $invocant = shift;
    ...
    bless $obj, $invocant;
}
```


Instance Methods as Constructors

While we're at it: there is no reason to restrict object construction to classes.

```
my HTML::Head $h1 = HTML::Head::->new;
my HTML::Head $h2 = $h1->new;
```

`ref` returns the package name of an object.

```
sub new {
    my $invocant = shift;
    ...
    bless $obj, ref($invocant) || $invocant;
}
```


Factory Methods

Methods that create objects are called *factory methods*.

A constructor is a special case of factory method: one that creates a new instance for its class.

To distinguish factory methods from constructors, give them a catchy name, like `create_`... or `make_`... .

Attribute Accessor Methods

Writing all those get_*attr* and set_*attr* methods gets boring soon.

One thing to try is combine the set and get methods:

```
# $obj->level(2);
# $lvl = $obj->level;

sub level {
    my $self = shift;
    $self->{level} = shift if @_;
    $self->{level};
}
```


Using Closures for Accessor Methods

For repetitive and otherwise boring tasks, use Perl.

```
foreach my $fld ( qw(level text) ) {
    no strict 'refs';
    *{$fld} = sub {
        my ($self) = shift;
        $self->{$fld} = shift if @_;
        $self->{$fld};
    }
}
```


AUTOLOAD Accessor Methods

We can skip the definition of the accessor methods, and leave it to AUTOLOAD to catch them.

```
sub AUTOLOAD {
    my $self = shift;
    croak "objects only, please" unless ref($self);
    my $name = our $AUTOLOAD;
    return if $name =~ /::DESTROY$/;
    if ( $name =~ /^HTML::Head::(.*)/ and
         $1 eq "text" || $1 eq "level" ) {
        $self->{$1} = shift if @_;
        return $self->{$1};
    }
    croak "undefined method $name called";
}
```


Class Data and Methods

Classes can have data, too.

These can be stored nice privately in lexical variables.

To allow derived classes to use the data, accessor methods are needed, just as with instance data.

These methods do not need to pay attention to their first argument, since they do not relate to any particular instance.

Class Data and Methods (cont'd)

Assume, in our HTML::Head examples, that we want to keep track of the number of allocated instances.

```
my $num_instances;
```

In the constructor:

```
$num_instances++;
```


Class Data and Methods (cont'd)

If we need the actual number of live instances, not just a tally, we need a destructor method:

```
sub DESTROY {
    $num_instances--;
}
```


Class Data and Methods (cont'd)

To access the class data from outside, or from a derived class, define an accessor method:

```
sub instances {
    shift;              # ignore invocant
    $num_instances;
}
```

No matter where and how invoked, this method will always return the value of the lexical.

Usually this is exactly what is desired.

Privacy of Instance Data

Instance data in hashes and arrays is always vulnerable to unwanted access.

Anyone having a reference can dereference it with `$obj->{priv_data}`.

A common technique to obtain real private instance data is by storing the data in a lexical hash, using the instance reference as the key.

Privacy of Instance Data (cont'd)

```
package Secure;
my %instance_data;

sub new {
    ...
    my $self = bless {}, $class;
    $instance_data{$self} = { attr => ... };
    $self;
}

sub get_attr {
    my $self = shift;
    $instance_data{$self}{attr};
}
```


Using Separate Hashes

Using separate hashes for each attribute, typing errors can be caught. It is slightly faster as well.

```
my %text;
my %level;

sub new {
    ...
    my $self = bless {}, $class;
    $text{$self} = undef;
    $level{$self} = undef;
    $self;
}

sub text {
    my $self = shift;
    @_ ? $text{$self} = shift : $text{$self};
}
sub level {
    my $self = shift;
    @_ ? $level{$self} = shift : $level{$self};
```


Using Separate Hashes

```
}
```


Singleton Classes

Singleton, monadic or highlander classes are special in that there is always no more than one instance.

Often there is no instance at all – the class just provides methods for the application to use.

If there's a constructor, it re-uses a single referent, or it returns just the class name.

Oops!

Aren't we forgetting something?

Documentation

Any software system is essentially useless without proper documentation.

This is particularly true for object based systems.

Every object based module should document:

- High level: how application programs should use it
- Medium level: how the module can be reused to form other components.
- Low level: how the module can be maintained

If you want your module to be used, pay special attention to documenting the high level API.

Object Oriented Programming with Perl

Part V

Overload

In the HTML examples, method `publish` produces an HTML string representing the contents of an instance.

With `overload`, we can do this automatically when an object is evaluated in string context.

```
use overload '""' => "publish";
```

Now, referencing an instance of HTML::Head in string context will automatically call the `publish` method to produce the HTML string.

```
my $h1 = HTML::Head::->new(level=>1, text=>"A Title");
print STDOUT ("<html><body>\n",
              "$h1\n",
              "</body></html>\n");
```


Overload (cont'd)

Overloading of operators is also very convenient when the meaning of operators can be extended to the objects.

For example, in the `HTML::List` example, concatenation could be used to add another thing to the list:

```
{
    package HTML::List::UnorderedList;
    use overload "."  => "add";
    use overload '""' => "publish";
}

my $ul = HTML::List::UnorderedList::->new;
$ul->add("uitem1");
print STDOUT ($ul . "uitem2" . "uitem3", "\n");
```


Overload (cont'd)

Beware of unexpected results.

```
{
    package HTML::List::UnorderedList;
    use overload "."  => "add";
    use overload '""' => "publish";
}

my $ul = HTML::List::UnorderedList::->new;
$ul . "uitem1" . "uitem2" . "uitem3";
print STDOUT ("$ul\n");
```


Overload (cont'd)

So this is probably the best to do:

```
{
    package HTML::List::UnorderedList;
    use overload ".="  => "add";
    use overload '""' => "publish";
}

my $ul = HTML::List::UnorderedList::->new;
$ul .= "uitem1";
$ul .= "uitem2";
$ul .= "uitem3";
print STDOUT ("$ul\n");
```


Class::Struct

The standard `Class::Struct` module can be used to quickly setup classes.

```
package HTML::Head;

use Class::Struct;
struct ( level => '$', text  => '$' );

sub publish {
    my $self = shift;
    "<h" . $self->level . ">" .
      $self->text .
        "</h" .  $self->level . ">";
}

1;
```


Class::MethodMaker

Powerful module to construct classes.

Supports several class creation styles: hashes, hash from args, hash with init.

Supports several attribute access styles.

Lots of fancy things like abstract methods, class attributes, attribute groupings and delegation.

Available on CPAN.

Object Oriented Programming with Perl

Epilogue

The Books

Object Oriented Perl
Damian Conway
Manning, 1999
ISBN 1-884777-79-1

Programming Perl, 3rd ed.
Larry Wall, Tom Christiansen & Jon Orwant
O'Reilly & Associates, 2000
ISBN 0-596-00027-8

The Perl Docs

`perlobj`: *Perl Objects*
`perlboot`: *Beginner's Object Oriented Tutorial*
Randal L. Schwartz
`perltoot`: *Tom's Object-Oriented Tutorial*
Tom Christiansen
`perltootc`: *Tom's Object-Oriented Tutorial for Class Data*
Tom Christiansen
`perlbot`: *Bag'o Object Tricks*