Object Orientation Basics

Lesson 2

Classes and Objects in Perl

Inheritance

Basic Concepts

(cited from Damian Conway: Object Oriented Perl, p. 2)

- **Object**: anything that provides a way to locate, access, modify, and secure data
- **Class**: description of what data is accessible through a particular kind of object, and how to access that data
- **Method**: means by which an object's data is accessed, modified or processed
- **Inheritance**:the way in which existing classes of objects can be upgraded to provide additional data or methods
- **Polymorphism**: the way that distinct objects can respond differently to the same message, depending on the class

Objects

- Contain the real data (or pointers to it)
- Data in objects are called attribute values
- Access to data should go via the object(address)
- Access should be done by subroutines only
  - these subroutines are called object methods
  - in some languages this is enforced
  - Perl allows direct access to the data (discouraged!!)

Classes

- Describe a particular kind of object
  - what attributes belong to such a kind of object
  - how to create an object (constructor)
  - how to get access to the attributes (methods)
- The methods related to the kind of object define the class **interface**
- The class itself can define data (class data)
  - access to these class data with class methods
- The object is modeled after the class definition, it is a class **instance**, the class is the blueprint for it

Inheritance

- A Class can be defined in terms of other classes
  - the derived class inherits from the base class
  - inheritance can extend to several levels (hierarchies)
  - inheritance from several base classes is allowed: multiple inheritance
- Inheritance hierarchy describes degree of abstraction
- Abstract classes useful, blueprint for derived classes
- Abstract methods serve as placeholder for methods in derived classes (define interface, but no working code)

Polymorphism

- Methods with the same name in several classes
- Proper method gets called according to kind of object
- Inheritance polymorphism
  - derived classes have common ancestor
  - all methods defined in base classes
  - methods get redefined in derived classes if required
- Interface polymorphism
  - Classes only share method with the same name
  - method not guaranteed to exist, fallback mechanism required, if method not provided in the class

Further Concepts

- Aggregation
  - collection of related objects form new objects
  - the new objects can have additional methods
  - simple objects easier to debug, complex structures can be built from simple building blocks
- Persistence
  - Objects survive the end of the program
  - done by serializing objects and storing it in databases, files etc.

OO and perl related documentation

- Tutorials that come with perl
  - perlreftut, perlboot, perltoot, perltooc, perlbot
- Book: Object Oriented Perl, Damian Conway, Manning Publications, 2000
- Perl specific: Collection of links on OOP in Perl (not up to date, 1998)
  - http://genome-www.stanford.edu/perlOOP/

Classes in Perl

- A class in perl is built upon the package concept
- A package is a separate name space
  - Namespaces get switched by the **`package`** command
  - All data types of a package are globally visible
  - Adressing possible using ***`package_name`*`::`** prefix
  - The default namespace is **`main::`** or simply **`::`**

```
$a=0;                    #Variable $main::a or $::a
package myclass;
  $a=1;                  #Variable   $myclass::a
  sub inc {$a++;}        #Subroutine &myclass::inc
```

Package (De)Initialization

- In each package (also in main) code that gets executed as early/late as possible can be defined

```
BEGIN { statements }
END {statements }
```

- Analogy to **awk**

```
print "Step 2\n";
BEGIN { print "Step 1\n"; }
```

Modules

- Modules are files (suffix `.pm`) that contain packages
- Modules usually contain package of same name
  - but a module can contain more than one package
  - or a package can consist of several modules
- Modules get loaded with the `use` command
  - a module has to return true (last line is `1;`)
  - old perl4 style programs were using `.pl` files that got loaded using `require`
  - for old `.pl` files there are more recent `.pm` modules

Looking up modules

- Modules are searched in directories whose names are stored in the `@INC` variable
- There is a correspondence between `use Module;` statements and file names

```
use Test;       # look for file Test.pm
use Test::Log;  # look for file Test/Log.pm
```

- The search path for modules can be extended by
  - using the command line flag `-i`
  - changing the contents of `@INC` using `BEGIN` blocks

Module Creation

- can be done using the **`h2xs`** program

  `h2xs -AXn Test::Log` creates the skeleton files

  Test/Log/Log.pm Test/Log/Makefile.PL
  Test/Log/test.pl Test/Log/Changes Test/Log/MANIFEST

  - documentation should be written in perlpod format
    - plain old documentation, see **`perldoc perlpod`**
  - installation with **`perl Makefile.PL; make; make test; make install`**

- For much more information see e.g.

  http://world.std.com/~swmcd/steven/perl/

Package and lexical Variables

- Package variables always global, can be accessed in main program and other packages
- lexical variables do not belong to a package
  - created using `my $var;` or `my ($var1, $var2);`
  - access only within block (or file or eval string)
  - get erased when leaving scope (refcount = 0)
- Lexical variables help in encapsulating data
  - (see closures)
- See also `perldoc perltooc`

Perl objects

- Each call to the constructor has to give a new object, i.e. a separate container for data
- Cannot be achieved with ordinary arrays or hashes
  - Will always be tied to a specific storage location
  - Anonymous hashes and arrays provide distinct and adjustable portions of memory to hold object data

```
$p1={}; $p2={}; print "$p1, $p2\n";
$p1->{attribute} = "value";
```

  - The storage has to be labeled according to the class
  - This is "magically" done by the function **`bless`**

Perl Objects

- An object is a "blessed" reference to data
  - blessing is done with Class name (=package name)

```
package myclass;              # Class myclass
# no named hash, could be accessed/modified by name!
$record = {num=>1, str=>'a'};
print ref($record);           # HASH
bless $record, 'myclass';     # Object $record
print ref($record);           # myclass
```

- Object creation usually done in subroutine **new** (called constructor), but e.g **connect** also legal

Methods (1)

- Object methods and Class methods are normal subroutines
- Call has to be done using additional syntax
- Object method call

```
$object->method(@args);
```

- Class method call

```
Class->method(@args);
```
or
```
Class::method(@args);
```

Methods (2)

- Called subroutine gets an additional first argument
  - Class name for class methods
  - Object (blessed reference) for object methods
- The class an object belongs to is obtained with **`ref`**

```
sub Hello {
    my $self = shift;
    my $class = ref $self;
    print "A Hello from class $class\n";
}
$record->Hello; # call of the object method
```

Introductory Example

```
package Simple::Test;
use strict;
sub new {
  my ($self, $hashref) = @_;
  $hashref = {} unless $hashref;
  bless $hashref, $self;
}
```

Introductory Example(2)

```
sub get_num { my $self=shift; $self->{num} }
sub set_num { my $self=shift;
              $self->{num}=shift;
}
sub str { my ($self,$arg)=@_;
          $self->{str}=$arg if @_=1;
          return $self->{num}
}
```

Introductory Example(3)

```
package main;
my $obj1=new Simple::Test  {str=>'Obj1',num=>7};
my $obj2=Simple::Test->new({str=>'Obj2', num=>3});
my $num = $obj1->get_num;
$obj1->set_num($num*$num);
my $str = $obj2->str();
$obj2->str("New String");
use Dumpvalue;
my $dumper = new Dumpvalue;
print $dumper->dumpValue($obj1),
      $dumper->dumpValue($obj2), "\n";
```

Inheritance

- Inheritance in Perl is Inheritance of methods
- Inheritance is controlled by the `@ISA` array
- `@ISA` contains class names which are inherited from

```
package Printer;      # current package is Printer
use vars qw( @ISA ); # to use @ISA under use strict;
@ISA=('Net::Node');  # Printer is a Net::Node
```

- multiple inheritance: more than one element in `@ISA`

```
@ISA=('Net::Node','Device');
```

- Inheritance is recursive, i.e. may span several levels

Inheritance Hierarchy

- Inheritance is used to look for methods
  - If method not in current class
  - then search first for methods in `$ISA[0]`
  - then search in parents of `$ISA[0]`
  - then search in further elements of `@ISA`
  - then search in class `UNIVERSAL`
  - then search for method `AUTOLOAD` in current class
  - then search for method `AUTOLOAD` in parents
  - otherwise report an error

Inheritance Hierarchy (2)

- Inheritance rules can be described as follows
  - Search from current position to top of inheritance tree for a given method
  - Continue search from left to right in `@ISA`
  - All methods inherit from class `UNIVERSAL`
  - If method not found this way then look for method `AUTOLOAD` using the same rules

The class SUPER

- SUPER is a pseudo package
- Usage: method in parent class performs a partial task (delegation), remaining part in current method
- `SUPER::test` looks in parents of the current class
  - whole inheritance hierarchy is searched
  - **first** found method `test` gets used

Usage of the SUPER class

```
package Simple::Test;
use vars qw( @ISA );
@ISA=qw(Simple);
sub print2 {
  my $self = shift;
  print "First the specific print2 is called ...\n";
  $self->SUPER::print2;
}
package Simple;
sub print2 {
  print "Then the generic one: object belongs to ", ref shift,
   "\n";
}
$obj->print2;
```

The AUTOLOAD method

- `AUTOLOAD` is called if a method does not exist
- `$AUTOLOAD` contains name of the missing method
  - `$AUTOLOAD` is a variable of the current package
  - `AUTOLOAD` sees object reference as first parameter
- Can be used to replace similar get/set methods
- Disadvantage: method lookup always triggered
- `AUTOLOAD` is called for **every** undefined method

A simple AUTOLOAD example

```
use vars qw ( $AUTOLOAD );
sub AUTOLOAD {
  print "AUTOLOAD called: $AUTOLOAD\n";
  my ($self, $val) = @_;
  $self->{$1} = $val if $val && $AUTOLOAD =~ /.
  *::set_(\w+)/;
  return $self->{$1} if $AUTOLOAD =~ /.*::get_(\w+)/;
}
package main;
$obj->set_value("a string");
print $obj->get_value(), "\n";
```

Optimized AUTOLOAD

- **`AUTOLOAD`** generates method on request
  - see example in ”Object Oriented Perl” p.94/95
- A method has to be generated on the fly whose name is the value of **`$AUTOLOAD`**
- Solution using the typeglob operator and a closure

```
*{$AUTOLOAD} = sub {return $_[0]->{$attr}};
```

- Only first call triggers AUTOLOAD

Optimized AUTOLOAD example

```
sub AUTOLOAD {
  print "AUTOLOAD called: $AUTOLOAD\n";
  my ($self, $val) = @_;
  if ($AUTOLOAD =~ /.*::fetch_(\w+)/) {
    no strict "refs";
    my $attr = $1;
### dynamic code generation ###
    *{$AUTOLOAD} = sub {return $_[0]->{$attr}};
    print "missing function $AUTOLOAD has been defined\n";
    return $_[0]->{$1};
  }
}
package main;
print $obj->fetch_value(), "\n";
print $obj->fetch_value(), "\n";
```

Automatic method generation

- In simple cases classes consist of a constructor and set/get methods for attributes
- Methods look similar
- Method generation according to templates possible
- Several modules in standard Perl and on CPAN
- Class::Struct in core Perl
- Class::MethodMaker on CPAN (not installed here)

Class::Struct

- Generates Perl Code for new and accessors
- provides subroutine `struct`
- simplest usage with

```
use Class::Struct;
struct Test => {
                    name   => '$',
                    scores => '@'
                };
```

- Disadvantage: not well suited for complex tasks

Questions and answers

- What is contained in the hash **`%main::`** (also **`%::`**)
  - It contains the symbol table (can be inspected and printed)
- Can I have subroutines that act both as object method and as a ordinary subroutine

```
sub print3 {
  if ( ref $_[0] ) {
    print "called as method from object ", ref shift,
  "\n"
  }
  print "Subroutine arguments: @_\n";
}
```

Questions and answers (2)

- Can I call a constructor using an object method (construct an object of the same type instead of a given class)

```
sub new2 {
  my ($caller, @args) = @_;
  my $class = ref $caller || $caller;
  bless {}, $class;
}
my $obj3 = Simple::Test->new2;
my $obj4 = $obj3->new2;
print "\obj3 and \$obj4 are objects: " ,ref $obj3,
      " and ",ref $obj4, "\n";
```