A Crash Course in Perl5

Part 6: Object-oriented programming

Zeegee Software Inc.
http://www.zeegee.com/

Terms and Conditions

Road map

- Basics
  - Introduction
  - Perl syntax
  - Basic data types
  - Basic operators
- Patterns
  - Introduction
  - String matching and modifying
  - Pattern variables
- Data structures
  - LISTs and arrays
  - Context
  - Hashes
- Flow control
  - Program structures
  - Subroutines
  - References
  - Error handling
- Data
  - Input and output
  - Binary data
  - Special variables
- Object-oriented programming
  - Modules
  - Objects
  - Inheritance
  - Tying

OOP

Modules

Packages

- **Packages** in Perl are like packages in Ada. They allow code from many different developers to be combined with *very* low risk of naming conflicts.
- Each package defines its own **namespace**. To reference a global `$VarName` in *another* package, code in *your* package would have to refer to it like this...

```
$OtherPackage::VarName
```

- The **default package** is `main`, and `$main::sail` may be abbreviated as `$::sail`

Using packages

- Here's a simple program...

```
#!/usr/bin/perl -w;
                          # package is main initially
$side = 'Us';

package Other;            # package is Other at this point
$side = 'Them';
print "Other> \$side        = $side\n";
print "Other> \$main::side  = $main::side\n";
print "Other> \$Other::side = $Other::side\n";
```

```
Other> $side        = Them
Other> $main::side  = Us
Other> $Other::side = Them
```

Nesting packages

- **Packages may be nested**... so inside `Outer::`, you can have package `Outer::Inner`, with vars like `$Outer::Inner::var`.
- You must always use the **full package name** to refer to something outside your own package... so inside `Outer::`, you can't just say `$Inner::var`: you have to use the full name.
- Each package has its own **symbol table** where it keeps the values of all identifiers (variables, subroutines, etc.) defined to belong to that package.

What gets packaged?

- Only identifiers starting with **letters or underscores** are stored in a package's symbol table
- All other symbols (e.g., special variables like $/ and even $_) are forced to belong to package `main::`
- As a special case, the following symbols are *also* forced to belong to `main::`

STDIN	ENV	ARGV
STDOUT	INC	ARGVOUT
STDERR	SIG	

Modules

- A **module** is just a package which has been placed into a source file with the name "*package*.pm".
- Generally pulled in via **use()**.
- Modules are designed to hold **reusable code**...
  - They may contain useful functions, which **use()** will import directly into the "user's" namespace for convenience.
  - They may contain OO code, which provides class and method definitions *without* namespace corruption.

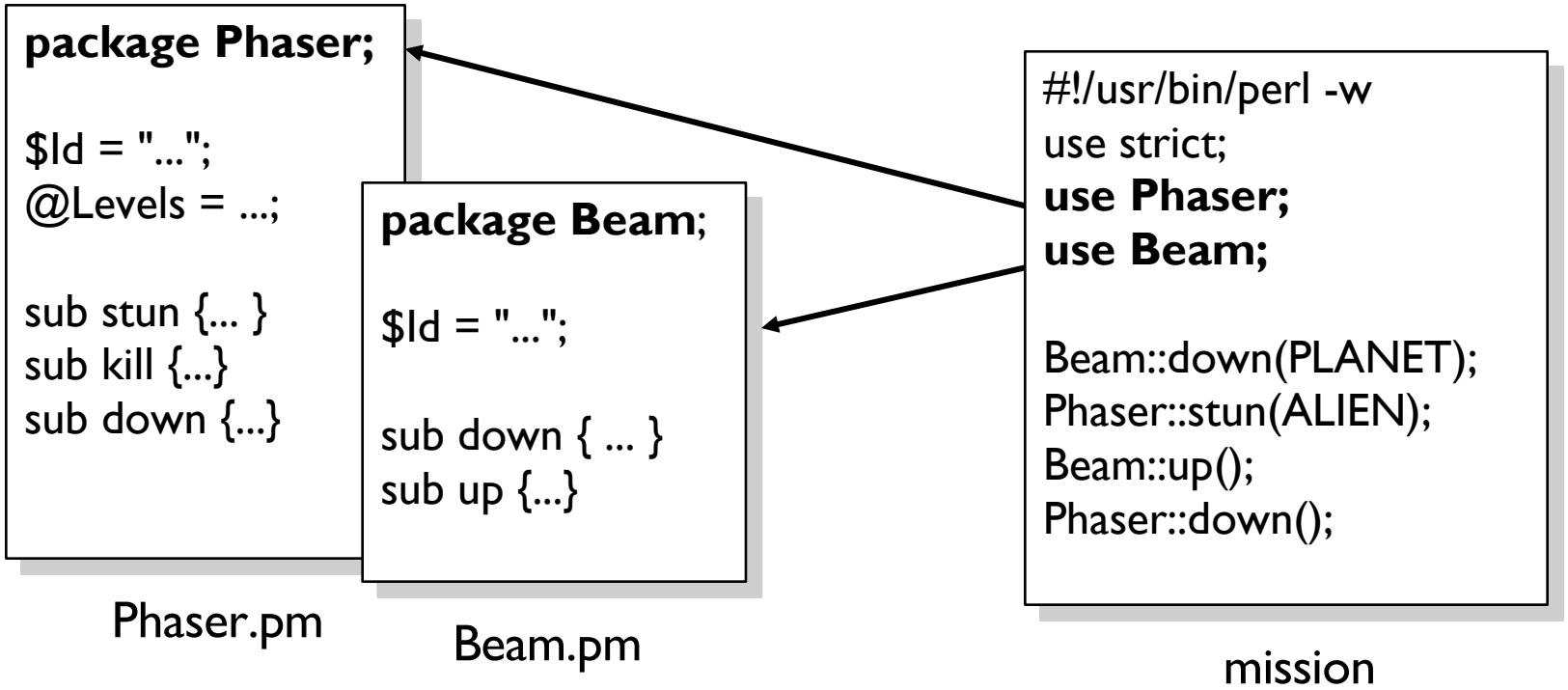
A sample module

 # In file Phaser.pm...

```
package Phaser;                # begin Phaser:: namespace
use strict;                    # helps catch many, many errors

$Id     = '$Header: Phaser.pm v.3.4 mscott $';
@Levels = qw(STUN HEAVYSTUN KILL BARBEQUE);

sub stun  { print "Stun $_[0]: Zap!\n"; }
sub kill  { print "Kill $_[0]: ZZZZZZAP!\n"; }
sub down  { print "Charging down\n"; }

1;                                 # make sure Phaser loads okay
```

Code organization

- Here's how the packages comprising a possible program, ***mission***, might be organized:



BEGIN / END

- There are two special subroutines that you can define in a package to **initialize/deinitialize** it:
  - BEGIN() will get called as soon as it is completely defined, even before the rest of the package is parsed.
  - END() is executed as late as possible, usually as the result of calling die().
- The **sub** keyword is optional when defining them
- You can have **multiple BEGIN blocks** (which are called in order of definition) and **multiple END blocks** (which are called in reverse order of definition).

require()

- **require** will ensure that the contents of a module are loaded, *without* importing any symbols into *your* package's namespace:

```
require ProofOfPurchase;
```

- To access anything in that module, you would have to qualify it with the package name:

```
$date = $ProofOfPurchase::Date;
```

- But **use()** is preferred...

use()

- Use **use** to ensure that the contents of a module are loaded, *and,* to import any symbols into *your* package's namespace that...
  - the module wishes to **export**, *and*...
  - that you wish to **import**
- There are 3 basic forms you will see...

```
use Phaser;                    # imports @EXPORT
use Phaser qw(stun $ID);       # imports just these
use Phaser ();                 # imports nothing
```

Preferred over `require Phaser;`

Importing with use()

- If the package being **use**d inherits from Exporter, then you cannot import any symbols that aren't in that package's @EXPORT or @EXPORT_OK
- Any symbols you import are now part of *your* package:

```
stun($foe);   # stun() imported from Phaser
```

- Symbols that were *not* imported must be qualified with the package name:

```
Phaser::kill();
```

require vs. use

The statement:

```
use Module LIST;
```

Is exactly equivalent to:

```
BEGIN {
    require Module;
    import Module LIST;
}
```

And since BEGIN blocks are evaluated at compile time, so are "use" statements.

@INC

- List of directories to look for files/modules referenced by **require** and **use**.
- Initially consists, in order, of...
  - Any **-I** arguments to Perl (just like **cc**)
  - The default Perl library directory
  - "." (the current working directory)
- May be modified by your program at any point

Altering @INC, and "use"

Careful! Since "use" statements are executed at compile time (as soon as they are parsed), the following *will not work*:

```
push @INC, "/my/perl/dir";
use MyModule;
```

NO!

- If you modify @INC, you must do so inside a BEGIN{...} block that precedes the first "use" statement:

```
BEGIN { push @INC, "/my/perl/dir" }
use MyModule;
```

- Best bet:

```
use lib "/my/perl/dir";
use MyModule;
```

Extension modules

- If you have a C library you'd like to call directly from Perl, you can do so if your Perl5 has been installed to support *dynamic loading*.
- Modules that provide an interface to underlying C/C++ functions are called **extension modules**. They look just like ordinary modules to the outside world.

`Socket::`	interface to BSD socket library
`Fcntl::`	interface to file descriptor library
`POSIX::`	interface to POSIX routines

- Read the Perl *XS* and *XS Tutorial* manual pages for details...

OOP

Objects

Don't panic!

- **An object is simply a reference** to a data structure (scalar, hash, array) that happens to know which class it belongs to.
- **A class is simply a package** that happens to provide methods to deal with object references.
- **A method is simply a subroutine** that expects an object reference (or, for "static" methods, a package name) as the first argument.

The scenario...

- Let's say we want to have a class, **Person**, where...
  - Instances of Person have instance variables **name** (a string), **age** (an integer), and **hobbies** (an array of strings).
  - Instances of Person have instance methods for getting/modifying these variables

A class is just a package...

- ...So a good way to start is by creating a `Person` module dedicated to our new `Person` package:

```
#  A class defining a person:
package Person;

#  ...end of file: make sure it returns true!
1;
```

Person.pm

An object is just a reference

- An object is just a reference to a data structure.
- Most common data structure to use for objects is a **hash**, since we can easily store/retrieve instance variables by name.
- The **bless()** function tells the data structure (e.g., the hash) what class it belongs to. It's not an instance until it's blessed!
- Here's a simple constructor, **new()**, which returns a new Person as a reference to an initially-empty hash:

```
package Person;
sub new {              # constructor: create and return a new Person...
   my $self = { };        # ref to an empty hash
   bless $self;           # bless and return $self
}
```

A method is just a subroutine

- A **static method** (or **class method**) is a subroutine which:
  - Takes the **class** as its first argument (often ignored)
  - Provides functionality for the **class as a whole** (e.g., construct an object, look up an object by name, etc.)
- A **virtual method** (or **instance method**) is a subroutine which:
  - Takes an **object reference** as its first argument (usually shifted into a variable called `$self` or `$this`)
  - Provides functionality for a **single object** (e.g., access/modify an instance variable of the object, print the object, etc.)

Invoking methods

- You can use **indirect object** syntax...

```
$will = new Person "Riker";
output $will;
```

method class args, ...
method obj args, ...

- You can use C++-style **message** syntax...

```
$will = Person->new("Riker");
$will->output();
```

class->**method**(args, ...)
obj->**method**(args, ...)

- I like the latter when using message-chains:

```
Person->find("Riker")->output();
```

Unlike ordinary functions, the empty argument list `()` is *optional*, and is assumed if not provided.

Invoking "unknown" methods

- Sometimes you want to call one of several similar methods, but you don't know ahead of time which one you want. To avoid excessive if-thens, you can put the method name in a scalar:

```
#  The following assumes that the methods...
#          Person::name()
#          Person::formalname()
#  ...are legal,and take their arguments identically...

$will = Person->find("Riker");
$getname = ($casual ? 'name' : 'formalname');
$name = $will->$getname();
```

bless(*REF, PACKAGE*)

- **bless()** takes the object referenced by REF, tells it that it now belongs to the given PACKAGE, and then returns REF.
- If PACKAGE arg not given, defaults to caller's package.

Think you might use inheritance? Then it's more robust to *explicitly* provide the PACKAGE (which is usually given by the first argument to the constructor):

```
package Person;
use strict;              # for safety!
sub new {                # constructor...
        my $class = shift;   # get actual class of object being built
        bless {}, $class;    # bless and return ref to empty hash
}
```

More complex constructors

- Let's do some initialization before returning the new object...

```
sub new {                # constructor...
     my $class = shift;            # get actual class
     my $self = {};                # create ref to empty hash

     $self->{Name} = 'Anonymous';  # default name
     $self->{Hobbies} = [];        # empty array of hobbies

     bless $self, $class;          # bless and return object
}
```

in Person.pm

Using constructors

- How do we create a new Person object? Like this:

```
#!/usr/bin/perl -w
use strict;
use Person;                          # load Person class

$person = Person->new();             # create a new Person
print $person->{Name}, "\n";         # prints "Anonymous"
```

- Remember, you can also use **indirect object** syntax:

```
$person = new Person();
```

Constructors with arguments

- Let constructor take name and age as optional arguments:

```
sub new {
     my $class = shift;              # get actual class
     my $name = shift || 'Anonymous';
     my $age  = shift;
     my $self = {};                  # create ref to empty hash
     $self->{Name} = $name;
     $self->{Age} = $age;
     $self->{Hobbies} = [];
     bless $self, $class;            # return object
}
```

in Person.pm

```
$will  = Person->new('W. Riker', 40);
$tasha = Person->new('T. Yar');
```

Sample virtual methods

- Let's add a method to output an individual Person:

```
sub output {
     my $self = shift;                # get the object

     my $age = $self->{Age} || 'unknown';
     my $hobbies = (int(@{$self->{Hobbies}}) ?
          join(", ", @{$self->{Hobbies}}) : 'none');

     print "Person:\n";
     print "   Name:    $self->{Name}\n";
     print "   Age:     $age\n";
     print "   Hobbies: $hobbies\n";
     1;                               # always nice
}
```

in Person.pm

More virtual methods

- Let's add some storage/access methods:

```
sub name {                          # Get the name...
     my $self = shift;
     $self->{Name};
}
sub set_name {                      # Set the name...
     my ($self, $newname) = @_;
     $self->{Name} = $newname;
}
```

in Person.pm

```
$jeanluc = new Person('Picard');
$jeanluc->set_name('Capt. Picard');
print $jeanluc->name(), "\n";      # prints "Capt. Picard"
```

Destructors

- Objects are automatically **destroyed** when the last reference to them goes away
- If you want to do something just before an object goes away, provide a DESTROY method:

```
sub DESTROY {
     my $self = shift;
     print "$self->{Name}: ",
           "(cough) I'm... (gasp) dying...\n";
}
```

OOP

Inheritance

The scenario

- We want to create Doctor, a subclass of Person, where Doctor has all the instance variables and methods of Person, plus...
  - Doctor has an additional instance variable, *specialty*
  - Doctor has an additional virtual method, *diagnose()*
  - Doctor's *name()* method is slightly different from Person's *name()* method, in that it automatically puts the title "Dr." in front

Defining a subclass

- We'll need a new module, called Doctor, of course:

```
package Doctor;
use strict;
use Person;              # load parent class(es)
@ISA = qw(Person);       # declare all parent classes

1;
```

in Doctor.pm

- If we didn't need to add new instance variables or alter functionality, we could stop right here: even inherited *new()* will bless as Doctor, since we used 2-argument *bless()*!

@ISA

- The `@Class::ISA` array holds the list of the names of all parent classes of `Class`. Often listed in **qw().**
- If `Class` is asked to call a method `fubar` that it doesn't recognize, classes in the @ISA array are traversed recursively (depth-first, left-to-right) until `fubar` is found.

```
package Doctor;
use strict;
@ISA = qw(Person Printable);
```

```
$doctor = new Doctor;
$doctor->printme();      # okay, if Printable::printme exists!
```

Overriding constructors

- We want to use the same constructor as for Person, but...
  - We want to initialize an additional inst. var, *specialty*
  - We want to make sure new object is blessed as a Doctor!

```
sub new {
     my $class = shift;
     my $self = Person->new(@_);  # call inherited constructor
     bless $self, $class;         # now, we're a Doctor!
     $self->{Specialty} = 'GP';   # set Doctor's instance var
     $self;
}
```

in Doctor.pm

Adding methods

- It's easy to add a method to a subclass that isn't in the superclass: just define it!

```
sub diagnose {
     print "He's dead, Jim!\n";
}
```

in Doctor.pm

```
Doctor->new('McCoy')->diagnose();     # okay
Person->new('Uhura')->diagnose();     # ERROR!!
```

Calling overridden methods

- When you override a **virtual method** but want to call the overridden method, you can specify the class explicitly by qualifying the method name with the scoping (::) operator:

```
sub name {
     my $self = shift;
     my $name = $self->Person::name(@_);
     "Dr. $name";
}
```

in Doctor.pm

- This syntax also works with **static methods**, but the usefulness of it escapes me...

The special SUPER class

- If, within a class, you override a method but want to use the inherited method *no matter which of your parent classes it's defined in*, you can use the SUPER class:

```
sub name {
     my $self = shift;
     my $name = $self->SUPER::name(@_);
     "Dr. $name";
}
```

in Doctor.pm

- Now it's easier to see what's really going on

ref(*REF*)

- Sometimes you want to know exactly what class something is. The **ref()** operator takes a reference and returns the class:

```
sub be_a {
     my ($self, $thatclass) = @_;
     my $myclass = ref $self;
     ($myclass eq $thatclass) or
        print "Dammit, Jim, I'm a $myclass, ",
              "not a $thatclass!\n";
}
```

in Doctor.pm

```
$bones = new Doctor('McCoy');
$bones->be_a('bricklayer');
```

OOP

Tying

What is tying?

- Sometime you will define a class which is very much like one of the built-in datatypes (scalar, array, or hash)... and you'd like to use normal Perl syntax instead of method calls:

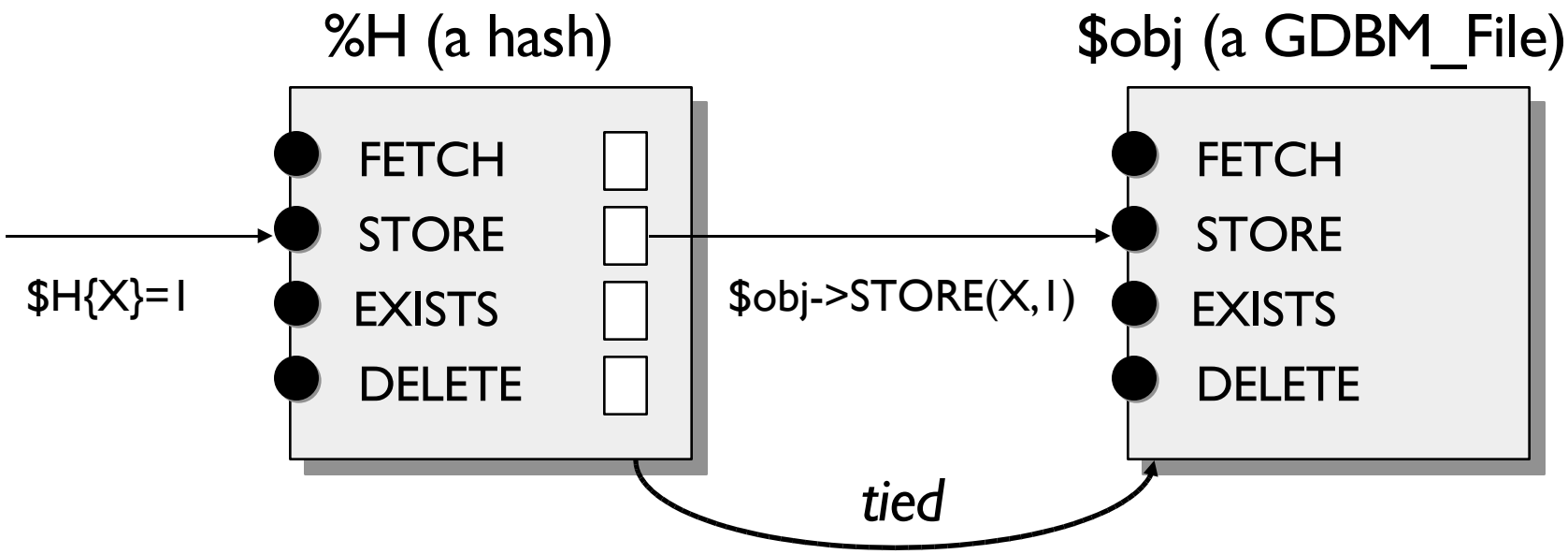
```
# I'd REALLY like to say $mything{'name'} = $value:
$mything->store('name', $value);
```

SURPRISE! Perl lets you make any abstract data type "pretend" to be a scalar, array, or hash... all you have to do is have it support a few simple methods!

- Associating a variable of a built-in datatype to a user-defined datatype is called ***tying***.

What's going on

```
# Tie a hash %H to a GDBM_File:
tie %H, 'GDBM_File', "test.gdb", &READER;

# Get the underlying object:
$obj = tied(%H);
```



Where is this used?

- Older versions of Perl had a special call, *dbmopen()*, which tied a hash to a DBM database (like a hash on disk). The call *dbmclose()* broke the tie.
- Newer versions of Perl have generalized these calls to *tie()* and *untie()*.
- Now, many ways of accessing DB-like files (GDBM, SDBM, ODBM, NDBM) through tied hashes.

OOP / Tying

Let's tie one on!

To allow your class to be tied to a hash, just define the following methods:

TIEHASH class, arglist	Constructor. Return blessed instance.
FETCH this, key	Fetch the value at *key*.
STORE this, key, value	Store *value* under *key*.
DELETE this, key	Delete entry *key*.
CLEAR this	Clear entire hash.
EXISTS this, key	Does this *key* have an entry?
FIRSTKEY this	Rewind, and return the first key.
NEXTKEY this, lastkey	Return the next key, given the *lastkey*.
DESTROY this	Destructor.

Same basic idea for arrays and scalars.