Perl Short Course
Final (Sixth) Session

Duncan C. White
dcw@doc.ic.ac.uk

Matt Johnson
mwj@doc.ic.ac.uk

- In the fifth session, we saw how to use pre-existing Perl modules, so now we'll look at how we actually construct Perl modules, specifically:
  - **packages and namespaces**,
  - **creating Perl modules**,
  - **symbol export/import**,
  - **how to write Perl classes**
  - **how to inherit classes**
- Perl's approach to modularity, information hiding, abstraction and OO is refreshingly lightweight.
- Perl constructs its modules and classes using about half a dozen new concepts and keywords - which we'll introduce one-by-one as we come to them.
- But first, an aside. Sometimes, you want to declare that your code requires at least a certain version of Perl to run, eg 5.8:

```
use 5.008;
```

Packages and Namespaces

- The first concept needed to provide modularity is that of a separate *namespace* so that modules may be written separately without worrying when (say) two modules both define a subroutine called `printrecord`.

- These namespaces are called *packages* in Perl.

- Each package has a *symbol table* which contains all that package's symbols - subroutines, global variables, constants and filehandles defined inside that package.

- But not lexical (`my`) variables, which are different from global variables, they are dynamic variables that do not live in any package.

- The default package is called `main` - so all our previous examples were automatically in package `main`.

- At any point in a single Perl source file, we may switch from the current package to another via the declaration:

  ```
  package new_package_name;
  ```

- To see how packages work, let's start with **eg1**:

```
package main;   # default namespace anyway
$x = 3;         # this is now main's $x
print "first:  x = $x\n";

package fred;   # switch into new fred namespace
$x = 17;        # this is now fred's $x
print "second: x = $x\n";

package main;   # switch back to main namespace
                # $x is now main's $x again, still 3
print "third:  x = $x\n";
```

- When run, this produces:

```
first:  x = 3
second: x = 17
third:  x = 3
```

- We can access one package's variables from another package via the `packagename::` syntax - consider **eg2**, which is **eg1** plus the following:

```
package aardvark;

print "main's x = $main::x\n";
print "fred's x = $fred::x\n";
```

- Like so much in Perl, just because you *can* access another package's variables, doesn't mean you *should* do so in practice - at least, not very often, unless it's useful, cute or amusing:-)

- We can even do the whole example without switching package once, as in **eg3**:

```
$main::x = 3;
$fred::x = 17;

print "main's x = $main::x\n";
print "fred's x = $fred::x\n";
print "current x = $x\n";
```

- By the way, `$::x` is a shorthand for `$main::x`.

- We can do the same with arrays and hashes - all package variables in fact. Here's **eg4**:

```
package main;
$scalar = 3;
@arr = (5,3,8);
%hash = map { $_ => $_ * $scalar } @arr;
print "first:  hash{3} = $hash{3}\n";

package fred;
$scalar = 4;
@arr = (5,2,3);
%hash = map { $_ => $_ * $scalar } @arr;
print "second: hash{3} = $hash{3}\n";

package main;
print "third:  hash{3} = $hash{3}\n";
```

- The `map` calls are a standard idiom for converting an array to a hash, where the values are computed from the keys. In this case, `%main::hash` ends up mapping the array elements to their triples.

- We've now seen all types of package variables, but package subroutines are far more typical - **eg5**:

```
package one;
sub hello { print "one "; }
hello();

package two;
sub hello { print "two\n"; }
hello();

package main;
print "one more time..\n";
one::hello();
two::hello();
```

- When run, this prints:

```
one two
one more time..
one two
```

- We can do this without switching package - **eg6**:

```
sub one::hello { print "one "; }
one::hello();

sub two::hello { print "two\n"; }
two::hello();

print "one more time..\n";
one::hello();
two::hello();
```

- Doesn't this suddenly begin to sound a little like OO programming?

Modules = Single File Packages

- The core *modularity concept* is the ability to split a large program into *separate source files*. All serious languages have some form of this capability, often linked with *separate compilation* and *control of the interface*. These separate components are variously called *modules, units* or (in extreme cases) *classes*.

- The package examples we've seen so far have been single source files which switch package several times.

- But most Perl packages occupy single complete files, with the package declaration right at the top.

- These are simply called *modules* - a Perl module is defined as follows:

  module `fred` = *package* `fred` *in file* `fred.pm`, *with optional interface control (later).*

- To gain access to the facilities of a module `fred`, we simply do

  ```
  use fred;
  ```

  just as we did with standard Perl modules.

- One weird detail is that each module, when loaded via `use`, must *succeed*. As such, every Perl module must end with a spurious true value, such as `1;`

- To see modules in action, let's split **eg5** into separate modules - copy **eg5** to **eg7**, move the `one` chunk into **one.pm** and add a trailing `1;` so that the module load will succeed:

```
package one;
sub hello { print "one "; }
hello();

1;
```

- Back in **eg7** we replace the missing code with `use one`, and then split out **two.pm** similarly, adding a `use two` to **eg7**:

```
package two;
sub hello { print "two\n"; }
hello();

1;
```

- Our main program **eg7** has now become:

```
use one;
use two;

print "one more time..\n";
one::hello();
two::hello();
```

- Syntax check all three files, then run `./eg7`. Now, let's add the obvious module `three`.

- Package names may be hierarchical, as we've already seen in modules like `Data::Dumper` - which would correspond to a file called **Dumper.pm** in a directory called **Data**.

- Suppose our modules were not `one` and `two` but `one::two` and `three::four` - **eg8**:

```
use one::two;
use three::four;

print "one more time..\n";

one::two::hello();
three::four::hello();
```

- Module `one::two` corresponds to a file **two.pm** in a directory called **one**, and similarly for `three::four`. So, we'll need to create the directories:

```
mkdir one three
```

- Now we can create **one/two.pm**:

```
package one::two;
sub hello { print "one two "; }
hello();

1;
```

- Finally, create **three/four.pm**:

```
package three::four;
sub hello { print "three four\n"; }
hello();

1;
```

- Syntax check all three files, then run `./eg8`.

Perl Modules and Strict

- So far, you may have noticed that we haven't been using `use strict` in our package/module examples.

- Recall from the fourth session that one of the main consequences of strict mode was that every global variable must either be turned into a lexical variable - by `my` - or defined - by `our/use vars`.

- We now know that global variables are contained within a package, and hence the proper Perlish term for a global variable is a *package variable*.

- We can use strict mode in modules (indeed, it's recommended) as long as we either:
  - have no package variables in a module (because we often don't need them)
  - turn all package variables into lexical `my` variables (this has the added advantage that they become truly private to the module)
  - turn most package variables into `my` and define the rest using `our`.

- So, let's turn **eg8** and it's modules into strict versions - just add `use strict;` to all three files.

Interface Control

- All this `one::two::hello` stuff is rather longwinded, in most languages the module designer can choose which (public) symbols to export, and the module user can choose which symbols to import. *Interface control* is our *third modularity concept*.

- How do we do it in Perl? By using a Perl special module called `Exporter` which exists for precisely this purpose:

- `Exporter` defines three conceptual sets:
  - The set of symbols exported from a module and imported into a client by default (`@EXPORT`).
  - The set of additional symbols exported from a module which a client can choose to import (`@EXPORT_OK`).
  - The set of named *tags*, each of which represents a set of symbols which may be imported via the tag name (`%EXPORT_TAGS`).

- We will cover the first two - see **perldoc Exporter** for all the gory details (tagged symbol sets, importing symbols matching a regex, etc).

- On the client side, we control what is imported via variations on the `use` syntax:

`use module;`	*import the default set of symbols* - everything on the module's `@EXPORT` list.
`use module ();`	*import no symbols.*
`use module qw(A B C);`	*import only symbols* `A`, `B` *and* `C` - these symbols must either by on the default list `@EXPORT` or the optional list `@EXPORT_OK`. As a special case, the symbol `:DEFAULT` means *all the default symbols.*

- Here's an example of an `Exporter` module **bob**:

```
package bob;
use strict;
use Exporter qw(import);

our @EXPORT = qw(A1 A2);
our @EXPORT_OK = qw(B1 B2);

sub A1 { print "A1 "; }
sub A2 { print "A2 "; }
sub B1 { print "B1 "; }
sub B2 { print "B2 "; }
sub C1 { print "C1 "; }

1;
```

- Here's the main program **eg9** that uses **bob**:

```
use strict;
use warnings;
use bob;

A1();
A2();
print "\n";
```

- Let's play with the variations of `use` to see what we can do.

- How might we gain access to `C1` which is not exported in either list?

What can/should we Export?

- Technically, we can export any package-based symbol: any package variable, any subroutine, any constant, any old-style filehandle and any typeglob (not covered in this course!). In practice, the typical mix of exported symbols is $90-100\%$ subroutines, $0-10\%$ constants, and $0-1\%$ variables.

- The *information hiding principle* suggests that you should export as little as possible, to hide as much as possible. Certainly, all private subroutines and variables should be hidden - recall that turning private variables from package `our` variables into lexical `my` achieves this neatly and robustly.

- When you've decided that something should be exported, there's still the choice of whether to export it by default (in `@EXPORT`) or optionally (in `@EXPORT_OK`).

- The *namespace pollution principle* suggests that as little as possible should be in `@EXPORT`. Most public symbols should be in `@EXPORT_OK`.

- The basic rule of thumb is that it should be "safe" to import the default set without causing problems.

- What about name clashes? What happens when two modules both export a symbol (eg. our earlier `hello` subroutine), and a single client script tries to import the same symbol twice?

- You get a perl warning at the second `use` statement saying `Subroutine main::hello redefined`; the second imported `hello` is used!

- Of course, the client can always choose whether or not to import that symbol via specifying an import list. But it's particularly unpleasant if the client can no longer import the default set!

- Alternatively, even without importing the symbol at all, for that matter even if the module does not export the symbol(!), the client can still access it when needed via `pkg::hello`!

Perl Objects and Classes

- As in any language, the basic purpose of *classes* in Perl is to provide *objects* - tidy little collections of data and behaviour. We've already seen how to use predefined classes to create and use objects, now we'll see how to write classes.

- The basic concepts involved here are *objects*, *classes*, *methods (object and class)* and *inheritance*. Let's try to give a first cut set of Perlish definitions:
  - A *class* is a Perl module, usually not exporting any symbols, containing class and object methods obeying the following conventions.
  - An *object* is some piece of reference data - usually a hash reference - which remembers the name of it's own class. This is called a *blessed reference*.
  - A *class method* (such as the *class constructor*) is a subroutine that takes the class name as it's first argument. The *class constructor* is conventionally called `new`, *destructors* are possible (but not often needed).
  - An *object method* receives it's own object (`$self`) as it's first argument.
  - *Single and multiple inheritance* is provided by a simple package search algorithm used to locate method subroutines.

Example Perl Class - Person.pm

```
package Person;
use strict;

my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26);

# the object constructor
sub new {
        my( $class, %arg ) = @_;
        my $obj      = bless( {}, $class );
        $obj->{NAME} = $arg{NAME} // $default{NAME};
        $obj->{SEX}  = $arg{SEX}  // $default{SEX};
        $obj->{AGE}  = $arg{AGE}  // $default{AGE};
        return $obj;
}

# get/set methods - set the value if given extra arg
sub name {
        my( $self, $value ) = @_;
        $self->{NAME} = $value if defined $value;
        return $self->{NAME};
}

sub sex {
        my( $self, $value ) = @_;
        $self->{SEX} = $value if defined $value;
        return $self->{SEX};
}

sub age {
        my( $self, $value ) = @_;
        $self->{AGE} = $value if defined $value;
        return $self->{AGE};
}

1;
```

- Here's **eg10**, the main program that uses **Person**:

```
use strict;
use warnings;
use Person;

sub printperson ($)
{
	my( $person ) = @_;

	my $class = ref($person);
	my $name = $person->name;
	my $age = $person->age;
	my $sex = $person->sex;
	print "$class: name=$name, age=$age, sex=$sex\n";
}

my $dunc = Person->new( NAME => "Duncan",
                        AGE  => 42,
                        SEX  => "m" );
printperson( $dunc );

$dunc->age( 20 );
$dunc->name( "Young dunc" );
printperson( $dunc );
```

- Note that the Perl function `ref()`, when applied to a blessed reference, returns the name of the package the reference was blessed into.

- When syntax checked and run, **eg10** produces:

```
Person: name=Duncan, age=42, sex=m
Person: name=Young dunc, age=20, sex=m
```

- But why did we put `printperson` in the main program - it obviously should have been a method in class `Person`! How hard is converting a normal subroutine to a method?

Aside - Overloading Stringification

- Perl has an advanced feature called *operator overloading*. We can use this to specify how to automatically convert a `Person` object to a string.

- First, split out the formatting code from `printperson` into a separate method called `as_string`:

  ```
  # new method
  sub as_string {
          my( $self ) = @_;

          my $class = ref($self);
          my $name = $self->name;
          my $age = $self->age;
          my $sex = $self->sex;
          return "$class: name=$name, age=$age, sex=$sex\n";
  }
  ```

- Now, `printperson` is simply:

  ```
  print $self->as_string;
  ```

- Now, add the magic pragma:

  ```
  use overload '""' => \&as_string;
  ```

  and the `printperson` method becomes:

  ```
  print $self;
  ```

- Now, *delete the* `printperson` *method entirely* - **eg10** can now simply print each object itself.

Inheritance = Method Search

- Our final modularity concept is *inheritance*, sometimes known as *subclassing*. Perl implements full single or multiple inheritance in a very simple way:

- A Perl class (module) can name one or more parent classes as an ordered list, in a package variable called `@ISA`.

- `@ISA` is used in only one way: to determine which package's subroutine should be invoked when a method call is made. The method search algorithm for a method (say `hello`) is simple:

  - Start the search in the object's *blessed* package. If that package has a `hello` subroutine, use that.

  - Otherwise, perform a depth first search of the first parent class named in `@ISA`.

  - If still not found, perform a depth first search of the second parent class named in `@ISA`.

  - And so on through the remaining `@ISA` elements.

  - If still not found, report an error.

- Note that this search algorithm is even used for constructors - unlike many other OO languages, only one constructor method is called automatically. If you want *constructor chaining* you must do it yourself. We'll see an example shortly.

- Let's create a `Programmer` subclass of `Person`, with an additional property - a set of language skills (each one a language name and an associated competence level) represented by a hash ref.

- It's good practice when subclassing to check that an empty subclass doesn't break things, before adding any new functionality. So, here's our *empty subclass version* of `Programmer` - reusing everything from `Person`:

```
# stub class Programmer - reuse all methods!
package Programmer;
use strict;
use Person;
our @ISA = qw(Person);

1;
```

- Let's make **eg11** a copy of our final version of **eg10**, and then change both occurrences of `Person` to `Programmer`, i.e.:

```
use Programmer;
my $dunc = Programmer->new( NAME => "Duncan",
                            AGE  => 42,
                            SEX  => 'm' );
```

- What do we expect to happen? It should work just like before, but the object should know that it's a `Programmer`!

- Now syntax check everything, and then run **eg11** to see what happens:

```
Programmer: name=Duncan, age=42, sex=m
Programmer: name=Young dunc, age=20, sex=m
```

- But how did it work? Let's start by understanding how the constructor call works:

Constructor call:	`Programmer->new(args)`
Does `Programmer::new` exist?	no! continue search...
Find the first parent class of `Programmer`	`@Programmer::ISA = (Person)`, so `Person` is first parent
Does `Person::new` exist?	yes! use that!
Call `Person::new` as a class method:	`Person::new("Programmer",args)`

- `Person::new` is called with the arguments:

```
$class = "Programmer";
%arg = ( "NAME" => "Duncan", "AGE" => 42, "SEX" => "m" );
```

  and then creates a new object, `blesses` it into package `$class` (i.e. `"Programmer"`), initializes it, and finally returns it.

- All the ordinary method calls are handled in the same way, each time they are tried in package `Programmer`, found not to exist, and then tried (and found) in package `Person`.

- Note that stringifying our object for printing still works - so even the stringification overloading must be inherited properly.

- Ok, now let's start really implementing **Programmer**.

- Our second attempt will reuse the constructor and all the existing get/set methods from **Person**, add a new `skills` method, include a helper method `skills_as_string` which formats the skill set nicely, and override `as_string`:

```
package Programmer;
use strict;
use Person;
our @ISA = qw(Person);

sub skills {                    # additional get/set method
        my( $self, $value ) = @_;
        $self->{SKILLS} = $value if defined $value;
        return $self->{SKILLS};
}

sub skills_as_string {          # additional method
        my( $self ) = @_;
        my $sk = $self->skills;
        my @str = map {
                sprintf( "%s:%s", $_, $sk->{$_} )
        } sort(keys(%$sk));
        return "{" . join(", ", @str) . "}";
}

use overload '""' => \&as_string;
sub as_string {                 # override method
        my( $self ) = @_;
        my $pers = $self->Person::as_string;
        my $skills = $self->skills_as_string;
        return "$pers\t    skills=$skills\n";
}

1;
```

- `$self->Person::as_string` is an example of *method chaining*, which does a normal method call to the `Person::as_string` subroutine.

- Here's our test harness **eg11a** which uses some of the new features:

```
use strict;
use warnings;
use Programmer;

my $dunc = Programmer->new( NAME   => "Duncan",
                            AGE    => 42,
                            SEX    => "m",
                            SKILLS => {
                                "C" => "godlike",
                                "perl" => "godlike",
                                "C++" => "ok",
                                "pascal" => "good",
                                "java" => "minimal"
                            } );
print $dunc;

$dunc->age( 20 );
$dunc->name( "Young dunc" );
$dunc->skills( { "C" => "good", "pascal" => "ok" } );
print $dunc;
```

- When syntax checked and run, **eg11a** produces:

```
Programmer: name=Duncan, age=42, sex=m
            skills={}
Programmer: name=Young dunc, age=20, sex=m
            skills={C:good, pascal:ok}
```

- But... this is awful! Where have all Duncan's skills gone? Answers on a postcard please:-)

- The problem is that `Person::new` has no code to initialize a `SKILLS` field. And nor should it!

- So we must define our own `Programmer` constructor. The following would definitely work, by repeating all of `Person::new`'s initializations:

```
my %default = (NAME=>"Shirley", SEX=>"f", AGE=>26,
               SKILLS=>{java=>"ok"});

sub new {                       # the object constructor
        my( $class, %arg ) = @_;
        my $self        = bless( {}, $class );
        $self->{NAME}   = $arg{NAME}   // $default{NAME};
        $self->{SEX}    = $arg{SEX}    // $default{SEX};
        $self->{AGE}    = $arg{AGE}    // $default{AGE};
        $self->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
        return $self;
}
```

- This is a highly error-prone technique - the constructor logic might diverge over time.

- What we really want is *constructor chaining* - we'd like to call the `Person::new` constructor and then do a bit more work. Well, we can easily enough:

```
my %default = ( SKILLS => { java => "ok" } );

sub new {
        my( $class, %arg ) = @_;
        my $obj        = Person->new( %arg );
        $obj->{SKILLS} = $arg{SKILLS} // $default{SKILLS};
        bless( $obj, $class );
        return $obj;
}
```

- Note that `Person->new(%arg)` creates a full-blown `Person` object, blessed into package `Person`, which we then modify - add the `SKILLS` field, and *re-bless the object into* `$class`.

- Let's try that change!

- Isn't there a better way? Well, the only thing varying per-class appears to be the set of data fields which we want to initialize, and their default values. Even better, the data fields are just the keys of the default values. We can generalise `Person's` constructor as follows:

```
sub new {
        my( $class, %arg ) = @_;
        my $obj    = bless( {}, $class );
        my %default = $obj->_defaultvalues;
        while( my($datum,$value) = each( %default ) )
        {
                $obj->{$datum} = $arg{$datum} // $value;
        }
        return $obj;
}
```

- Now, each class defines a private `_defaultvalues` method, listing the default values of all the initializable data fields:

```
sub Person::_defaultvalues {
        return (NAME=>"Shirley", SEX=>"f", AGE=>26);
}
```

- Continuing:

```
sub Programmer::_defaultvalues {
        return ( NAME=>"Shirley", SEX=>"f",
                AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we still seem to be relisting all the defaults in each subclass.

- Can we fix this? Yes, with method chaining!

```
sub Programmer::_defaultvalues {
        my $self = shift;
        my %default = $self->Person::_defaultvalues;
        $default{SKILLS} = { java => "ok" };
        return %default;
}
```

- More generically still, we can even write the chained method call as:

```
        my %default = $self->SUPER::_defaultvalues;
```

  to call the first available parental `_defaultvalues` method.

Where are Modules found?

- When you say `use one::two` where does Perl search for the directory `one` containing the file `two.pm`?

- Evidently, the current directory is searched, otherwise our examples wouldn't work! Similarly, wherever the Perl standard (and additional CPAN) modules are stored is searched.

- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).

- Typically, then, we build a Perl program and some associated modules, and then want to:

  – Install the program into (say) `/homes/dcw/bin`.

  – Ensure that `/homes/dcw/bin` is on our path.

  – Install the modules into (say) `/homes/dcw/perllib`.

  – And have the program *know where to find the modules*.

- Perl has a list of locations that it searches, called the *include path*. The include path is available within a Perl script as the special variable `@INC`.

- You can add an extra directory (`/homes/dcw/perllib` for example) to the include path in two ways:
  - Run your Perl script with the `-I` flag on the command line (or `#!` line) via:

    ```
    perl -I/homes/dcw/perllib ...
    ```

  - Alternatively, near the top of your script, add:

    ```
    use lib qw(/homes/dcw/perllib);
    ```

- This works, but it's a real pain to give the code away, or move it to another location yourself - because you have to change every reference to `/homes/dcw/perllib` to (say) `/vol/project/XYZ/lib`.
- This becomes a serious program as your applications grow larger - imagine an application comprising 10 main programs and 50 support modules.
- We'd like a *position independent* way of specifying where to find the modules - by specifying their location *relative to* where the programs are - for example, wherever the programs are installed, the modules are found in `../perllib`.
- The standard Perl module `FindBin` helps:

```
use FindBin qw($Bin);
use lib qw($Bin/../perllib);
use MyModule;
```

What haven't we mentioned?

- Perl features such as:
  - *typeglobs* - manipulating symbol tables.
  - The old `local` keyword - a first attempt at `my` variables with lifetime-based semantics, still useful sometimes.
  - Autoloading - defining a subroutine `AUTOLOAD` which handles missing subroutines!
  - *Compile time* vs *run time* distinctions, `BEGIN` and `END` blocks.
  - Writing Perl code on the fly via `eval`.
- Using the Perl debugger (**perldoc perldebug** and **perldoc perldebtut**).
- *Perl and graphics* - building GUIs using `Tk` or `Gtk`, visualizing directed graphs via `GraphViz` and it's friends, constructing image files via `GD` (very useful for CGI programs that generate dynamic images).
- *Parser generators* using Perl - especially the awesome yacc-like module `Parse::RecDescent`.
- Perl threads - semaphores, thread queues etc.
- Interfacing external C libraries into Perl via `XS`, and embedding a Perl interpreter in other programs, eg. Apache and mod_perl.
- And lots lots more.... Perl 5.10 new features, Perl 6, Parrot..

Finding More Information about Perl

- O'Reilly's site `http://www.perl.com/` (*The Perl Resource*) is a wonderful source of Perl information, containing links to a multitude of Perl information.

- Our old friend **CPAN**, found at `http://www.cpan.org/`.

- The wonderful *Perl Journal* at `http://tpj.com/` which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.

- *The Perl Directory* at `http://www.perl.org/` is a directory of links to other Perl information and news.

- *The Perl Monks* at `http://www.perlmonks.org/` is a forum-based discussion site for all matters Perlish.

- That's all folks! Enjoy your Perl programming - and remember the Perl motto:

  There's More Than One Way To Do It!