Perl Short Course

Fifth Session

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

Matt Johnson
mwj@doc.ic.ac.uk

- In this session, we'll discuss using add-on modules to make Perl even more powerful.
- We'll cover:
  - **Perl's documentation system**,
  - **what a module is**,
  - **where we can find many useful modules**,
  - **how we use them in our own programs**,
  - **a brief detour into Perl objects**, and
  - **lots of examples of using some common modules**.
- But first, an aside: we've already said that you can omit the quotes on a *literal hash key string,* this is often used to pretend that a hash is a record/structure, as in: `$info{forename} = 'Duncan'` where `forename` is pretending to be a field name.

Perl's Documentation System

Perl has a tremendous amount of online documentation - reference info, tutorials, cookbooks and FAQs. All is accessed using **perldoc** - start by looking at the overview **perldoc perl**. You can also lookup documentation for:

- Any standard function: **perldoc -f functionname**.
- Any installed module: **perldoc modulename**.
- Standard library overview: **perldoc perlfunc**.
- Standard modules overview: **perldoc perlmodlib**.
- What a module really is: **perldoc perlmod**.
- The Perl FAQ: **perldoc perlfaq**.
- Search the FAQ for a term such as password: **perldoc -q password**.

All Perl documentation is written in Perl's own documentation format called **POD: Plain Old Documentation** (see **perldoc perlpod** for details).

One of the cute things about **POD** is that Perl understands **POD** documentation being included in your own Perl scripts and modules - so there's no possibility of losing the documentation!

Extending Perl - Add-on Modules

- One of the major motivations when Perl 5 was designed was to allow additional C libraries to be dynamically loaded into a running Perl interpreter. This feature is used to add major new functionality to Perl without having to recompile the Perl interpreter.
- Perl **modules** can either be *pure Perl* – literally just containing Perl code – or *XS* – containing code in C as well as Perl, either to provide an interface to an existing C library (like `Tk`), or, more rarely, to write code for which efficiency is vital.
- We're going to look at how to use modules which have been made available to us by others.
- Similarly, when we are writing large programs, we may want to structure our code as several Perl modules as well, with all the usual attributes such as *data hiding*, *abstraction* and separate *name spaces*. This will be covered in the final session.
- In addition to the Perl standard library of functions (see **perldoc perlfunc** and session 4 for details) which are immediately available for you to use, Perl comes with a *large number of modules* installed by default, which we can use simply by writing `use modulename` in our Perl scripts. At the end of the last session, we met one of these standard modules called `Data::Dumper`. We'll see more today.

CPAN: the Comprehensive Perl Archive Network

- Beyond Perl's standard modules, there are thousands of high-quality modules written by Perl developers, held together in a well organized collection called **CPAN**, found at:

  `http://www.cpan.org/`
- As well as the CPAN website – helpfully mirrored all over the world – there is a Perl module (called `CPAN`) which you can use to install most CPAN modules automatically. See **perldoc CPAN** for more information.
- It is *definitely* worth looking at CPAN before you start to write significant chunks of code – there may well be a module that already does a large part of what you want to do!

Module Naming Conventions

- Perl modules are always stored in files with the extension `.pm`, e.g. `POSIX.pm` – where `pm` stands for "Perl Module".
- The Perl module space is hierarchical. Module names - like `Data::Dumper` - may contain `::` and the first part of their name is usually a general indication of their area of interest.

- So `Data::Dumper` pretty-prints complex data structures for us, `XML::Simple` gives us a simple interface to manipulate XML structured data, `Math::BigInt` allows us to do mathematics with very large integers, etc etc.

Installing and Using Modules

- If you have to download and install a module yourself, you will be pleased to discover that the vast majority of modules have a common installation method:

```
perl Makefile.PL
make
make install
```

- You can specify extra switches on the `Makefile.PL` line to either install the module system-wide or for you alone. See **perldoc perlmodinstall** for details.

- Once a module has been installed, you need to tell Perl that you want to use the module in your own program - do this with the keyword `use`, as in:

```
use Data::Dumper;
```

near the top of your script.

- Note that we also use the keyword `use` for *pragmas* such as `use warnings` or `use strict`.

Perl Objects - Creating

- While Perl modules can provide simple subroutines and constants, quite a few also provide an *object-oriented* view of their functionality.

- Some of the modules we're about to use are OO-based, and so we need to briefly discuss how Perl does OO.

- In Perl, a class is a special kind of module, so class names often also include double colons and everything!

- Unlike many other languages, in Perl an object constructor can be called *whatever the class chooses*! However, most class designers use `new` as the name of the class constructor.

- So, assuming we have a class `Student`, if we want to create a new instance of a `Student`, we say:

```
use Student;
my $bob = Student->new();
```

- `$bob` is now an instance of class `Student`.

- Perl also provides the syntactic sugar

```
new Class( parameter_list )
```

which is exactly equivalent to

```
Class->new( parameter_list )
```


Perl Objects - Using

- Now we have `$bob`, we can now use it as an object.

- Assuming the class `Student` has an object method called `attend`, which takes the name of a lecture and a room, we could then say:

```
$bob->attend('Perl Short Course', 'Huxley 311');
```

- Note that the syntax is very similar to reference syntax. Behind the scenes, objects are implemented as references - usually to hashes - associated (blessed) with a Perl package.

- If `Student` has a class method called `population`, which might tell us how many `Students` we've instantiated, we'd call that by:

```
my $pop = Student->population();
```

- Many member functions want optional arguments, and a conventional way of doing this has emerged: pass a single hash literal, with parameter names as keys and parameter values as values. The keys are conventionally chosen starting with '-' and written without string quote marks, as in:

```
$obj->method( -name => 'hello',
              -values => [ 'a', 'b', 'c' ] );
```

- This tells us enough about Perl objects to begin discussing modules with OO interfaces.

IO::File – object-like file handling

- `IO::File` converts plain Perl filehandles into objects, allowing us to manipulate files using object-oriented techniques.

- Remember that "normal" Perl filehandles are global variables written as UPPERCASE strings, as in `INFH` in the following:

```
open(INFH, "myfile");
```

- It would be nice if a filehandle could be stored in a scalar - to pass it to subroutines or to declare local filehandles, for example. `IO::File` (a standard Perl module) gives us this, and the ability to use object methods to read and write...

- Let's take one of our earlier examples of file I/O and convert it to use `IO::File` on a trivial level, replacing `open()` with `$fh = new IO::File` and all uses of `IN` with `$fh` – this is our new **eg1**:

```
use IO::File;

my $fh = new IO::File("fred") || die "can't open fred\n";
while (my $line = <$fh>)
{
        chomp $line;
        print "read '$line'\n";
}
close($fh);
```

IO::File continued

- `IO::File` also provides a fully object-oriented set of I/O methods that enable us to write programs such as **eg2**:

```
use IO::File;

my $infh = new IO::File("fred") || die "can't open fred\n";
my $outfh = new IO::File("> bob") || die "can't open bob\n";
while( my $line = $infh->getline )
{
        chomp $line;
        $line =~ s/^\s+//;      # zap leading whitespace
        $line =~ s/\s+$//;      # zap trailing whitespace
        $line =~ s/\s+/,/g;     # turn whitespace into commas
        $line = lc($line);      # and lowercase

        $outfh->print( "$line\n" );
}
$infh->close;
$outfh->close;
```

- Note that we're using `$infh->getline` to read a line of text rather than our old friend `<$infh>`. Similarly we're writing `$outfh->print("stuff")` and `$infh->close`.

- Like any module, you can find more information on the `IO::File` module via **perldoc IO::File**.

- On occasion, you will want to turn an existing filehandle like `STDOUT` into a `IO::File` filehandle:

```
my $stdout = IO::File->new_from_fd( fileno(STDOUT), "w" );
```


Writing CGI scripts

- **CGI** stands for **Common Gateway Interface**, and specifies how external programs can communicate with a webserver, and hence with a client viewing a web page.

- We know what HTML looks like, and so producing this in Perl is simple. . .

```
print "<HTML>\n";
print " <HEAD><TITLE>Hello World!</TITLE></HEAD>\n";
print " <BODY><H1>This is a simple web page.</H1>\n";
print "  <H2>Brought to you the hard way.</H2>\n";
print " </BODY>\n";
print "</HTML>\n";
```

- All we really need to know of CGI just to send trivial output is that we need to send a *header* before the content, followed by a blank line. So, to make our Perl script work from the web, we add to the beginning:

```
print "Content-type: text/html\n\n";
```

  . . . producing example **eg3**.

- To install this as a CGI script:

```
cp eg3 ~/public_html/perl2007/eg3.cgi
```

  and then point a web browser at

```
http://www.doc.ic.ac.uk/~dcw/perl2007/eg3.cgi
```

- However, all this literal HTML is horribly unwieldy. Surely there must be a better way?

Use the CGI module, stupid!

- Of course there is! Perl has a brilliant `CGI` module that provides functions to deal with all of this nastiness. The following example **eg4** produces the same 'effect' (albeit with somewhat more verbose HTML and fewer linebreaks!):

```
use CGI;

my $q = new CGI;
print $q->header,
      $q->start_html("Hello World!"),
      $q->h1("This is a simple web page."),
      $q->h2("Brought to you the easy way."),
      $q->end_html;
```

- `CGI` contains many more methods, and can for example produce web forms:

```
use CGI;

my $q = new CGI;
print $q->header,
      $q->start_html('A trivial form'),
      $q->h1('A trivial form'),
      $q->start_form,
      'Enter your name:', $q->textfield('name'), $q->p,
      'Select your level of Perl expertise:',
      $q->popup_menu(
         -name => 'expertise',
         -values => ['Newbie', 'Adequate', 'Guru', 'Larry']
      ), $q->p,
      $q->submit,
      $q->defaults('Clear'),
      $q->end_form;

print $q->end_html;
```


Using CGI - Processing Forms

- The `CGI` module can also deal with processing form responses - the `param()` method either tells you whether any parameters are available, or extracts a particular parameter's value.

- So, let's extend our form to generate a suitably sarcastic response when you fill in the form and submit it (**eg5**):

```
my %response = (
        Newbie => "Get on with it then!",
        Adequate => "One day you too may wear sunglasses, my son
        Guru => "Pretty cool sunglasses.",
        Larry => "We bow before your godlike powers!" );
if( $q->param )
{
        # Process form parameters...
        my $name = ucfirst( lc( $q->param('name') ) );
        my $expertise = $q->param('expertise');
        my $msg = $response{ $expertise } || "umm?";
        print $q->hr, "Hello, $name the $expertise - $msg";
}
```

- Now on selecting a name (Joe) and an expertise level (Newbie), we get output like:

  Hello, **Joe the Newbie** - Get on with it then!

- Use **perldoc CGI** to find out more. You'll also come across CGI again later this year.

LWP::Simple - access the WWW

- LWP is one of those awkward module hierarchies whose name is not exactly obvious, but which is particularly useful! LWP stands for *libwww-perl*, and provides an interface to access resources on the WWW.
- One of the modules in this hierarchy is LWP::Simple, which, as its name suggests, provides a simple method of using LWP to access web content.
- Take a simple example: perhaps we want to be able to read some random web page from within a Perl script (**eg6**):

```
use LWP::Simple;
my $url = shift @ARGV ||
        "http://www.doc.ic.ac.uk/~dcw/perl2007/";
my $wp = get($url) || die "oops, no webpage $url!";
print $wp;
```

- Another powerful method in this module is getstore($url, $filename) which downloads the contents of $url directly to the named $filename.
- Note how the entire text of the web page is stored in a single Perl scalar. Did we mention that Perl strings can be big?

HTML::Parser - parse HTML

- One thing we can do with our newly-downloaded web page is to parse the HTML.
- HTML::Parser is a fairly complex beast, which requires reading its Perl documentation (via **perldoc HTML::Parser**) to understand fully!
- So, let's link LWP::Simple and HTML::Parser together to do something useful! (**eg7**):

```
use LWP::Simple;
use HTML::Parser;
use URI::URL;

my $url;
my @links = ();

#
# deal with a start tag with it's attributes
#
sub findlinks ($$)
{
        my( $tag, $attr ) = @_;

        return unless $tag eq "a";
        my $link = $attr->{href};
        return unless defined $link;
        $link = url( $link, $url )->abs;
        push @links, $link;
}
```


- And here's the main program of eg7:

```
# main program
die "Usage: eg7 [url]\n" unless @ARGV < 2;

$url = shift @ARGV || "http://www.doc.ic.ac.uk/~dcw/perl2007/";

my $webpage = get( $url ) || die "eg7: can't fetch URL $url\n";

my $parser = new HTML::Parser(
        start_h => [ \&findlinks, 'tagname,attr'] );

$parser->parse( $webpage );

# now @links contains the links - print them out.

foreach (@links)
{
        print "link: <$_>\n";
}
```

- Now, suppose we want to fetch all linked .ps or .tgz files, storing them together in a new directory (a new command line argument)? The guts of the change is replacing the link printout with:

```
mkdir( $destdir, 0755 ) unless -d $destdir;
chdir( $destdir ) || die "can't cd into $destdir\n";

foreach (@links)
{
        next unless m#([^/]+)\.(ps|tgz)$#;
        my $filename = "$1.$2";
        print "fetching $_ -> $destdir/$filename\n";
        getstore( $_, $filename ) || next;
}
```


DBI - access databases

- DBI is a module which allows Perl to connect to databases and manipulate data within them.
- Databases supported by DBI include MySQL, Oracle, Sybase, Microsoft SQL server, and PostgreSQL – we use both of the last two here in DoC.

Handling database connections

- DBI provides a *class method* connect to connect to a database. A typical example would be:

```
use DBI;

my $db = 'films';
my $host = 'db.doc.ic.ac.uk';
my $port = 5432;
my $user = my $password = 'lab';
my $dbh = DBI->connect(
            "dbi:Pg:dbname=$db;host=$host;port=$port",
            $user, $password
          ) || die "can't connect to $db as $user";
```

- $dbh is now a *database handle*, connected to the database we choose. (In this case, the DoC lab "films" database.)
- When we have finished, we need to disconnect the handle.

```
$dbh->disconnect;
```

DBI cont - Issuing queries

- Once we have a connection to the database, we then need to be able to issue queries over that connection to retrieve data.
- The first stage in querying via DBI is to `prepare` it. This entails specifying precisely what query we are going to run. We do this by calling an object method on the database handle:

```
my $sth = $dbh->prepare("select * from films");
```

- This returns a *statement handle*, which will contain the state of the query. We now need to `execute` this query:

```
$sth->execute || die "Database error: " . $dbh->errstr;
```

- `DBI` provides the last error to us in human-readable format via the database handle method `errstr`. Using the *do or die* idiom with this is a good idea!
- If we are running a `select` query, then we need to fetch the rows returned by the query (see next slide). Otherwise, we just need to `finish` the statement handle:

```
$sth->finish;
```


DBI cont - Fetching results

- There are several methods in DBI to do this – probably the most useful are the statement handle methods `fetchrow_hashref` and `fetchrow_array`.
- `fetchrow_hashref` returns the next row resulting from the query as a reference to a hash, with keys being field names and values being the table values.
- `fetchrow_array` returns the next row resulting from the query as an array containing the table values in the order in which the query requested the fields.
- Both of these return `undef` when all the rows are exhausted, so we commonly use them in a `while()` loop:

```
while (my $row = $sth->fetchrow_hashref)
{
   # do something
}
```

- Having fetched rows, we should `finish` the query, as stated earlier.
- Let's put all this together with an example (**eg8**).

DBI cont - example (eg8)

```
use DBI;

my $db   = "films";
my $host = "db.doc.ic.ac.uk";
my $port = 5432;
my $user = 'lab';
my $password = 'lab';

my $dbh = DBI->connect(
            "dbi:Pg:dbname=$db;host=$host;port=$port`",
            $user, $password
          ) || die "can't connect to $db as $user";

my $sth = $dbh->prepare("select * from films");

$sth->execute || die "Database error: " . $dbh->errstr;

while( my $record = $sth->fetchrow_hashref )
{
        print "Title:    $record->{title}\n";
        print "Director: $record->{director}\n";
        print "Origin:   $record->{origin}\n";
        print "Made:     $record->{made}\n";
        print "Length:   $record->{length}\n";
        print "-" x 30 . "\n";
}

$sth->finish;

$dbh->disconnect;
```

- Let's run it!
- And then fix the warning:-)

Personally, I tend to wrap all the prepare/execute/foreach record/finish clutter up into a reusable sql query function with a per-record callback function:

```
sub sqlcb ($$$)
{
        my( $dbh, $sql, $recordcb ) = @_;
        my $sth = $dbh->prepare( $sql );
        $sth->execute || die "Database error: " . $dbh->errstr;
        while( my $record = $sth->fetchrow_hashref )
        {
                $recordcb->( $record );
        }
        $sth->finish;
}

sub printrecord ($)
{
        my( $record ) = @_;
        print "Title:    $record->{title}\n";
        print "Director: $record->{director}\n";
        print "Origin:   $record->{origin}\n";
        print "Made:     $record->{made}\n";
        print "Length:   $record->{length}\n";
        print "-" x 30 . "\n";
}

my $dbh = DBI->connect( ... ) || die "can't connect...";
sqlcb( $dbh, "select * from films", \&printrecord );
$dbh->disconnect;
```

Note that if the per-record work is trivial you can call `sqlcb` with an anonymous subroutine, as in:

```
my $numrecords = 0;
sqlcb( $dbh, "select count(*) from films",
       sub { $numrecords = $_[0]->{count} } );
```

DBM - On-Disk Hashes Revisited

- In the first session's exercises, we briefly mentioned DBM - a very efficient storage system which can associate an arbitrary string with an arbitrary key with efficient indexed access.
- Back then, we used `dbmopen` and `dbmclose` to access the file using a platform-specific default DBM format. However, a much better way is to use `tie`, since it will let us specify exactly *which* DBM format to use (for there are many)!
- Here's our simple **mksecret** program from the first session, but using `tie` instead to create an SDBM, which is good for small amounts of data (**eg9**):

```
use Fcntl;
use SDBM_File;

tie(my %secret, 'SDBM_File', 'secrets-sdbm',
       O_RDWR|O_CREAT, 0666
   ) || die "oops, couldn't tie SDBM";

$secret{Romulan} = 1;
$secret{Klingon} = 1;
$secret{Vulcan} = 1;

untie(%secret);
```

- Note that SDBM actually creates two files: `secrets-sdbm.pag` and `secrets-sdbm.dir`. You still access the SDBM from Perl by calling it `secrets-sdbm`, though.

Using tie to perform conversions

- Using `tie` more than once allows us to convert between DBM formats easily! Let's convert our secrets file from SDBM to Berkeley DB format, provided by the `DB_File` module (**eg10**):

```
use Fcntl;
use SDBM_File;
use DB_File;

tie(my %secret, 'SDBM_File', 'secrets-sdbm',
       O_RDWR, 0666
   ) || die "oops, couldn't tie SDBM";
tie(my %newsecret, 'DB_File', 'secrets-bdb',
       O_RDWR|O_CREAT, 0666
   ) || die "oops, couldn't tie BDB";

%newsecret = %secret;                    # shazam!

untie(%newsecret);
untie(%secret);
```

- Berkeley DB is a single-file DBM format, and so it really writes a file called `secrets-bdb` (with a `.db` file extension on some platforms).
- If in doubt which DBM format to use, **perldoc AnyDBM_File** provides useful information on which to choose in a given situation.

Handling Command Line Options

- Many programs take extra options or switches on their command line. For example, many Unix commands understand `--help` to mean "tell the user how to use me".
- We've already discussed `@ARGV`, and we could obviously just use that to detect and process switches. However, as is often the case with Perl, someone else has already done the hard work for us and turned it into a module: `Getopt::Long`.
- `Getopt::Long`'s primary method is `GetOptions`, which looks at `@ARGV` and deals with anything which looks like an option its been told about.

```
use Getopt::Long;

my $list;
my $format = "DB_File";
my $result = GetOptions('list'     => \$list,
                        'format=s' => \$format);
```

- `--list` is merely a flag, whereas `--format` will require a string (`=s`). Both `--list` and `--format` are optional.
- On the next slide we'll use `Getopt::Long` in anger, to provide a command-line interface for our DBM files (**eg11**). As usual, there's lots more information available at **perldoc Getopt::Long**.

Example - Getopt::Long and tie

```
use Fcntl;
use SDBM_File;
use DB_File;
use Getopt::Long;

my $format = "DB_File";
my $result = GetOptions('format=s' => \$format);

die "Usage: eg11 [--format=S] filename [secrets]\n"
    unless $result && @ARGV >= 1;

my $filename = shift @ARGV;

tie(my %secret, $format, $filename, O_RDONLY, 0666) ||
    die "can't tie $filename using $format\n";
if( @ARGV == 0 )
{
    foreach (keys %secret)
    {
        print "$_ is a secret\n";
    }
} else
{
    foreach (@ARGV)
    {
        if( exists $secret{$_} )
        {
            print "Yes, $_ is a secret\n";
        } else
        {
            print "No, $_ is not a secret\n";
        }
    }
}
untie(%secret);
```