

# Scripting Languages

## Advanced Perl

Course: 67557

Hebrew University

Lecturer: Elliot Jaffe – אליוט יפה

# The principals of a programmer

- **Laziness** is the mother of invention
- **Impatience** makes us automate repetitive boring tasks
- **Hubris** makes us remember to write code that other people can read

# Perl

- -w turns on warnings
  - Warn about lots of questionable PERL constructs
- use strict
  - don't allow undefined global variables

# The list expansion problem

Remember this?

```
$a = 1;
```

```
$b = 2;
```

```
@c = (3,4);
```

```
sub MyFunction {
```

```
    ($arg1, @list, $arg2) = @_;
```

```
}
```

```
MyFunction($a, @c, $b);
```

# References

- Similar to pointers in C
- Are treated as scalar values
- Reference Counting for Garbage Collection

# Creating References

- Use the \ operator

```
@list = (1,2,3);
```

```
$listref = \@my_list;
```

```
$hashref = \%my_hash;
```

```
$coderef = \&my_function;
```

- Anonymous references

```
$anon_list = (1, 2, 3);
```

```
$anon_hash = {"a" => 1, "b" => 2};
```

# Using references

- Apply the appropriate data type operator

```
$bar = $$scalarref;
```

```
push(@$arrayref, $filename);
```

```
$$arrayref[0] = "January";
```

```
$$hashref{"KEY"} = "VALUE";
```

```
&$coderef(1,2,3);
```

```
print $globref "output\n";
```

# Using References (2)

- DeReferencing a block

```
$bar = ${$scalarref};
```

```
push(@{$arrayref}, $filename);
```

```
${$arrayref}[0] = "January";
```

```
${$hashref}{"KEY"} = "VALUE";
```

```
&{$coderef}(1,2,3);
```

# Using References (3)

- Syntactic Sugar

```
$arrayref->[0] = "January";  
                # Array element
```

```
$hashref->{"KEY"} = "VALUE";  
                # Hash element
```

```
$coderef->(1,2,3);  
                # Subroutine call
```

# The list expansion problem

Remember this?

```
$a = 1;
```

```
$b = 2;
```

```
@c = (3,4);
```

```
sub MyFunction {
```

```
    ($arg1, @list, $arg2) = @_;
```

```
}
```

```
MyFunction($a, @c, $b);
```

# Subroutines Revisited

- Pass by reference

```
$a = 1;
```

```
$b = 2;
```

```
@c = (3,4);
```

```
sub MyFunction() {  
    ($arg1, $list, $arg2) = @_  
    @list1 = @$list;  
}
```

```
MyFunction($a, \@c, $b);
```

# Dynamic Scoping

```
$a = "foo";  
sub dynamic {  
    local($arg1) = @_;  
    print "in dynamic $arg1 \n";  
    nested;  
}  
sub nested {  
    print "in nested arg1 = $arg1 \n";  
}  
dynamic($a);  
print "outside arg1 = $arg1 \n";
```

# File test operators

- A file test operator is a unary operator that takes one argument and tests to see if something is true about it

```
$readable = -r "myfile.txt";
```

- See the documents for more details

# Eval

- You can evaluate any string at run time

```
$myString = "print 'Hello World\n' ;";  
$val = eval( $myString );
```

- Create subroutines on the fly
- Evaluate statements from the command line

# \$<sub>\_</sub>

- Default input and pattern searching space

```
while (<>) {...} # equivalent only in while!  
while (defined($_ = <>)) {...}
```

```
/^Subject:/
```

```
$_ =~ /^Subject:/
```

```
tr/a-z/A-Z/
```

```
$_ =~ tr/a-z/A-Z/
```

```
chomp
```

```
chomp($_)
```

## \$\_ (2)

- Where Perl implies \$\_
  - Pattern matching operations `m//`, `s///`, and `tr///` when used without an `=~` operator.
  - Default iterator variable in a `foreach` loop if no other variable is supplied.
  - Implicit iterator variable in the `grep()` and `map()` functions.

# Map

`map BLOCK LIST` or `map EXPR, LIST`

- Apply the BLOCK to each element in LIST

```
%hash = map { getkey($_) => $_ } @array;
```

- is just a funny way to write

```
%hash = {};
```

```
foreach $_ (@array) {  
    $hash{getkey($_)} = $_;  
}
```

# Grep

`grep BLOCK LIST` or `grep EXPR, LIST`

- Apply the BLOCK to each element in LIST returning only elements which were TRUE

```
@foo = grep (!/^#/ , @bar);  
                # weed out comments
```

or equivalently,

```
@foo = grep { !/^#/ } @bar;  
                # weed out comments
```

# Sort

`sort BLOCK LIST` or `sort LIST`

- Sort the list using BLOCK or lexically

```
@foo = sort @list;
```

- or equivalently,

```
@foo = sort {$a cmp $b} @list;
```

- Numerically

```
@foo = sort {$a <=> $b} @list;
```

# Packages and Modules

- Package
  - BEGIN/END
- Modules
  - Require
  - Use

# Packages and Namespaces

- Store identifiers in separate namespaces

```
package MyPackage;  
our ($bar) ;  
sub foo {};
```

```
package main;  
MyPackage::foo ($MyPackage::bar) ;
```

# Modules

`use Module;` or `require module;`

- Find and load the package
- File name is Module.pm
- Module::SubModule means  
Module/Submodule.pm
- Searches @INC

# Closures

- Lamda ( $\lambda$ ) Calculus
- Two approaches
  - Maintain state across calls to a function
  - Operate on local values of variables
- Same implementation

# Consider this

- Dynamic scoping
  - Search for the variable at run-time

```
sub fred {  
    return $a++;  
}
```

- Where does \$a come from?

# What happens to \$a

```
{  
    my ($a) = 0;  
    sub fred {  
        return $a++;  
    }  
}  
print defined($a) ;
```

# Iterators

```
{  
  my $a = 0;  
  sub initCounter { $a = @_; }  
  sub nextCounter { return $a++; }  
}  
while (100 < ($b = nextCounter())) {  
  # do something  
}
```

# Fibonacci

```
{
  my $prev = 0;
  my $current = undef;
  sub fibonacci {
    if (!defined($current)) {
      $current = 1;
      return 0;
    }
    my $tmp = $current + $prev;
    $prev = $current;
    $current = $tmp;
    return $tmp;
  }
}
```

# Iterator Pattern

- `create_iterator()`
- `first()` - reset the iterator to its initial value
- `next()` - return the next item, increment
- `is_done()` - boolean – are there any more?
- `current_item()` - return the item but do not increment

# Perl Standard Modules

- More than 200 modules come standard with any Perl installation
- `bignum` – support arbitrary precision numbers
- `strict` – disallow some common mistakes
- `warning` – turn on specific warnings

# File::Compare

- `cmp(file1, file2, [bufsize])`
  - Compare two files. Return true if equal
- `compare(file1, file2, [bufsize])`
  - Compare two files. Return true if equal
- `compare()` is exported by default
- use `File::Compare qw(cmp);`

# Getopt::Long

- Accept command line arguments with long names

```
use Getopt::Long;
```

```
&GetOptions("size=i" => \ $offset,  
            "file=s" => \ $filename,  
            "names=s@" => \ @namelist);
```

```
% foo.pl --size 10 --file localdir/file.txt \  
        --name john --name tim --name elliot
```

# CPAN

- A. A Perl Module for downloading, building and installing Perl modules
- B. A very large collection of Perl modules

Comprehensive Perl Archive Network

Online since 1995-10-26

2808 MB 271 mirrors

4197 authors 7688 modules

# Data::Dumper

- Pretty print Perl objects
- Given a list of scalars or reference variables, writes out their contents in perl syntax (suitable for eval).

```
use Data::Dumper;  
print Dumper($foo);
```

# LWP

- The World Wide Web library for Perl
- Supports HTTP, HTTPS, FTP, News, Gopher
- The backbone of most web enabled Perl programs

# LWP Example

```
use LWP::UserAgent;
$ua = LWP::UserAgent->new;
$ua->agent("MyApp/0.1 ");

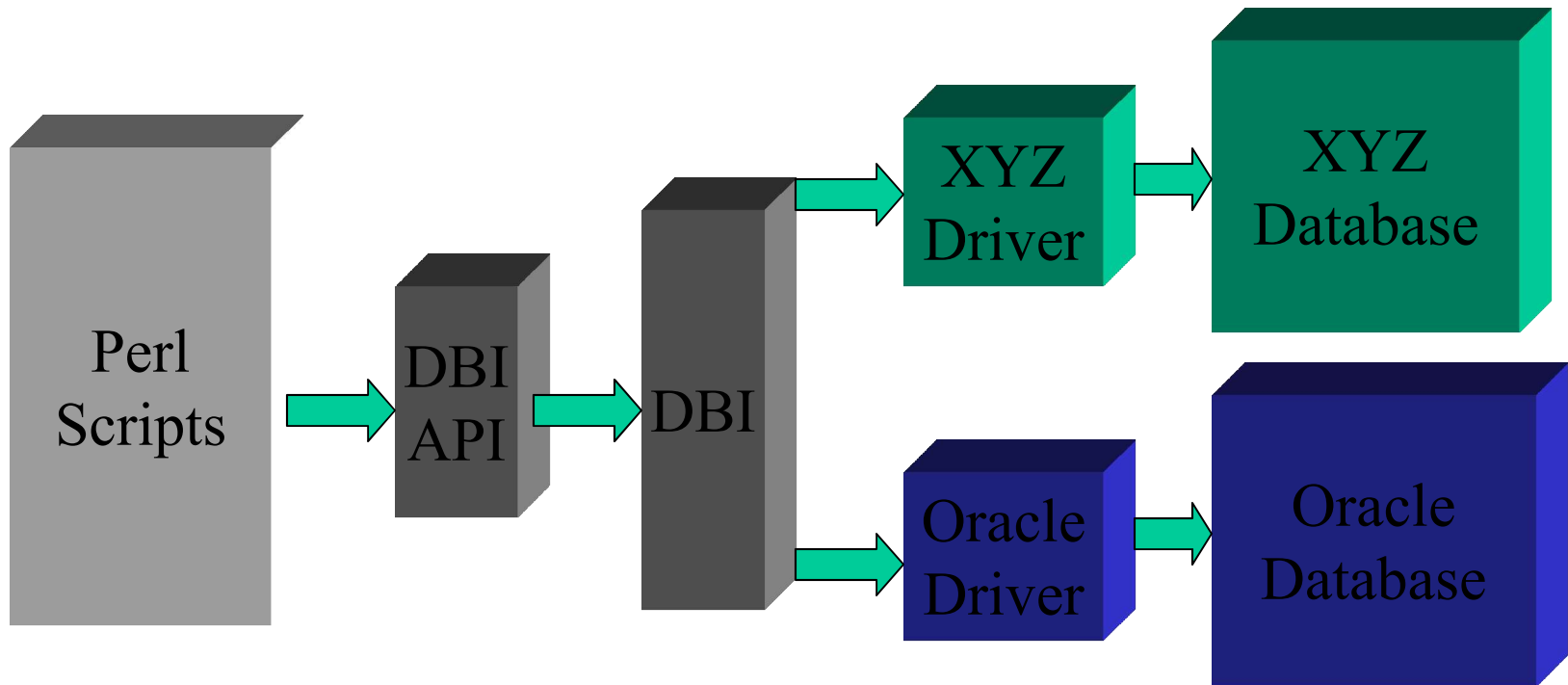
# Create a request
my $req = HTTP::Request->new(POST => 'http://search.cpan.org/search');
$req->content_type('application/x-www-form-urlencoded');
$req->content('query=libwww-perl&mode=dist');

# Pass request to the user agent and get a response back
my $res = $ua->request($req);

# Check the outcome of the response
if ($res->is_success) {
    print $res->content;
} else {
    print $res->status_line, "\n";
}
```

# DBI

- Database Independent module



# DBI

```
use DBI;

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                      'user',
                      'password' )
    || die "Database connection not made: $DBI::errstr";

my $sql = qq{ SELECT * FROM employees };
my $sth = $dbh->prepare( $sql );
$sth->execute();

my( $id, $name, $title, $phone );
$sth->bind_columns( undef, \$id, \$name, \$title, \$phone );

while( $sth->fetch() ) {
    print "$name, $title, $phone\n";
}

$sth->finish();
$dbh->disconnect();
```

# Strange modules

- SCUBA::Table::NoDeco
  - Compute no-decompression limits on repetitive dives
- Games::Backgammon
  - Play Backgammon (Sheysh-Beysh)
- LEGO::RCX
  - Control you Lego Mindstorm RCX computer

# Object Oriented Perl

- Classes are Packages, Methods are functions

```
package Person;
```

```
sub print {  
    my ($self) = shift;  
    printf( "Name:%s %s\n\n",  
           $self->firstName,  
           $self->lastName );  
}
```

# Bless me Father

```
#constructor
sub new {
  my $self = {
    _firstName => undef,
    _lastName => undef,
    _ssn => undef,
    _address => undef };
  bless $self, 'Person';
  return $self;
}
```

# Accessors

```
#accessor method for Person first name
sub firstName {
    my ( $self, $firstName ) = @_;
    $self->{_firstName} = $firstName
        if defined($firstName);
    return $self->{_firstName};
}
```

# Calling

```
use Person;  
$p1 = new Person;  
$p1->firstName = "Elliot";  
$p1->print;
```

# Making babies

```
# class Employee
package Employee;
use Person;
use strict;
our @ISA = qw(Person); # inherits from Person

sub new { #constructor
    my ($class) = @_ ;

    #call the constructor of the parent class, Person.
    my $self = $class->SUPER::new();
    $self->{_id} = undef;
    $self->{_title} = undef;
    bless $self, $class;
    return $self;
}
```

# Making babies

```
# class Employee
package Employee;
use Person;
use strict;
our @ISA = qw(Person); # inherits from Person

#create Employee class instance
my $emp1 = new Employee();
#set object attributes
$emp1->firstName('Ruti');
$emp1->title('Director of Operations');
$emp1->print;
```