


Perl Refresher

Mad Mongers

A vertical blue bar with a gradient, transitioning from a darker blue at the top to a lighter blue at the bottom, located on the left side of the slide.

Data Types

Scalars

```
#Obligatory Cheese  
my $message = "Hello World!";  
my $number = 7;
```

```
print $message, "\n";  
print $number, "\n";
```

```
Hello World!  
7
```

```
my $message = "World!";  
print "Hello $message\n";
```

```
Hello World!
```

```
my $sum = 2 + 3;  
print "The sum is: $sum\n";
```

```
The sum is: 5;
```

- Hold a single value
- Stings, Numbers, References, etc
- Denoted by **\$**
- Interpolation
- Math

Arrays

```
#Define Arrays of Numbers
my @primes = (1,2,3,5,7,11);

#Define Arrays of Strings
my @gfs = ("Sue","Amy","Jacqui");

#Define an empty Array
my @empty = ();

#Define a List Constructors
my @dogs = ( Charlie Lola );
```

```
#Access Elements
print $primes[3],"\n";
print $gfs[0] ,"\n";
print $dogs[1] ,"\n";
```

```
5
Sue
Lola
```

- Ordered Lists of Scalars
- Can be any length
- Denoted by **@**
- Preserves Order

Using Arrays

```
#Define Arrays of Strings
my @gfs = ("Sue","Amy","Jacqui");

#Girlfriends Increase
push (@gfs, "Jenny");

print "My new GF is ",$gf[3], "\n";
```

```
My new GF is Jenny
```

```
#Four GFs is too many!
$dumped = shift (@gfs);

print "I had to dump ",$dumped,"\n";
```

```
I had to dump Sue
```

```
#Jenny isn't happy
$jenny = pop (@gfs);
print "I need to move ",$jenny,"\n";

unshift (@gfs, $jenny);
print "My #1 GF is now ",$gfs[0], "\n";
```

```
I need to move Jenny
My #1 GF is now Jenny
```

- Adding and removing elements
- push
- shift
- pop
- unshift

More Array Stuff

```
#Define Arrays of Strings
my @gfs = ("Jenny","Amy","Jacqui");

#Count My Girlfriends
print "I have", ($#gfs+1),
      "girlfriends\n";
```

I have 3 girlfriends

```
#Now Jacqui is unhappy
@gfs = reverse (@gfs);

print "My #1 GF is now ", $gfs[0], "\n";
```

My #1 GF is now Jacqui

```
#It's too much trouble
$#gfs = -1;

print "I now have ", scalar(@gfs),
      "girlfriends";
```

I now have 0 girlfriends

- Counting Elements
 - \$# (last index value)
 - scalar()
- Reverse an Array
- Clear an Array

Hashes

```
#Define the hash
my %gfs = (
    Jacqui=> "Bossy",
    Amy=> "Whiney",
    Jenny=> "Demanding"
);

#Print the elements
print "I dumped Jacqui because she was
    ", $gfs{Jacqui}, "\n";
```

```
I dumped Jacqui because she was Bossy
```

- Associative Arrays
- Indexed by key
- Denoted by %
- Unordered

Using Hashes


```
my %gfs = (  
    Jacqui=> "Bossy",  
    Amy=> "Whiney",  
    Jenny=> "Demanding"  
);  
  
my @exGFNames = keys %gfs;  
  
print "My ex-girlfriends names were: ",  
    join (" ", "@exGFNames"), "\n";
```

```
My ex-girlfriends names were Amy,  
Jenny, Jacqui
```

```
foreach my $gf (@exGFNames) {  
    print "I dumped $gf because she was ", $exGFNames{$gf}, "\n";  
}
```

```
I dumped Amy because she was Whiney  
I dumped Jenny because she was Demanding  
I dumped Jacqui because she was Bossy
```

- Getting the Keys
- Use the keys to get the values

A vertical blue bar with a gradient, transitioning from a darker blue at the top to a lighter blue at the bottom, is positioned on the left side of the slide.

Variable Scope

Global Variables

- By default, all variables are global
- In general, don't use global variables

```
sub printem {  
    $inner = shift @_  
    print $inner;  
}  
  
printem "Hello!\n";  
print $inner
```

Output:

Hello!

Hello!

Using “my”

- Limits scope to a block of code
- Lexically scoped
 - Not limited to a single code block

```
my $var = 1;
my $tmp = 2;

if($var == 1) {
    my $tmp = 4;
    $var += $tmp;
}

print "$var\n";
print "$tmp\n";
```

Output:
5
2

Using “local”

- Temporary copy of a global variable
- Necessary in certain situations, but as a general rule, not used.

```
$value = 1;

sub printem() {
    print "\$value = $value\n";
}

sub makelocal() {
    local $value = 2;
    printem;
}

makelocal;
printem;
```

Output:

```
$value = 2
$value = 1
```

use strict;

- Variables must be declared
- Distrust bare words
- In general, makes it harder to write bad code

```
use strict;  
use strict 'vars';  
use strict 'subs';  
no strict;
```

```
$a = test_value;  
print "First program: $a\n";  
sub test_value {  
    return "test passed";  
}
```

Output:

First Program: test_value

```
sub test_value {  
    return "test passed";  
}  
$a = test_value;  
print "Second program: $a\n";
```

Output:

Second Program: test passed

A vertical blue bar with a gradient, transitioning from a darker blue at the top to a lighter blue at the bottom, is positioned on the left side of the slide.

References

Hard References

- Scalars that refer to other data
- Any type of data can be referred to, including other references
- Used primarily for efficiency

Creating References

- Backslash “\” is the reference operator

➤ Scalar (\\$)

```
my $var = "WebGUI";  
my $ref = \$var;
```

➤ Array (\@)

```
my @arr = ("WebGUI", "7.0");  
my $ref = \@arr;
```

➤ Hash (\%)

```
my %hash = (  
    "WebGUI" => "7.0"  
);  
my $ref = \%arr;
```

➤ Subroutine (\&)

```
sub hello {  
    print "Hello";  
};  
my $ref = \&hello;
```

Dereferencing

➤ Scalar (\$\$)

```
my $var = $$ref;  
my $var = ${$ref};
```

➤ Array (@\$)

```
my @arr = @$ref;  
my @arr = @{$ref};  
my $val = $ref->[0];
```

➤ Hash (%\$)

```
my %hash = %$ref;  
my %hash = %{$ref};  
my $item = $ref->{"WebGUI"}
```

➤ Subroutine (&\$)

```
&$ref;  
&{$ref};  
$ref->();
```

- Place the data type symbol in front of the reference.
- Using the arrow operator

Anonymous References

➤ Array

```
my @arr = (1,2,3,4,5);  
my $arr_ref = [1,2,3,4,5];
```

➤ Hash

```
my %hash = (1=>1,2=>2);  
my $href = {1=>1,2=>2};
```

➤ Subroutine

```
sub hello {  
    print "Hello";  
}  
my $hello = sub {  
    print "Hello";  
};
```

- No need to create the data type
- Almost exactly the same as creating the data type
- In most cases, it saves you a step

Data Structures

```
my $ref = [  
  ["Frank","Dillon","555-2233"],  
  ["JT","Smith","555-2325"],  
  ["Colin","Kuskie","555-3344"]  
];
```

```
foreach my $arr_ref (@$ref) {  
  foreach my $data (@$arr_ref) {  
    print $data;  
  }  
}
```

```
foreach my $arr_ref (@$ref) {  
  print $arr_ref->[0];  
  print $arr_ref->[1];  
  print $arr_ref->[2];  
}
```

```
print "First Name:", $ref->[0]->[0];  
print " Last Name:", $ref->[0]->[1];  
print "      Phone:", $ref->[0]->[2];
```

- Store multiple dimensions of data
- Data Structures are combinations of anonymous array and hash references

Advanced Data Structures

```
$ref = {  
  a=>[  
    {  
      last=>"Allan",  
      first=>"Richard",  
      phone=>["555-5940", "555-4345"]  
    },  
    {  
      last=>"Anderson",  
      first=>"Kevin",  
      phone=>{  
        mobile=>"555-3422",  
        home=>"555-2459"  
      }  
    },  
  ],  
  b=>[  
    {  
      last=>"Bach",  
      first=>"Randy",  
      phone=>["555-4432", "555-5456"]  
    },  
    {  
      last=>"Bonds",  
      first=>"Kim",  
      phone=>{  
        mobile=>"555-8789",  
        home=>"555-9876"  
      }  
    }  
  ]  
};
```

- Multiple data types
- Determining the reference type

```
if ( ref $ref->{$i}->[$j]->{$k} eq "ARRAY" ) {  
  print $ref->{$i}->[$j]->{$k}->[0];  
} else {  
  print $ref->{$i}->[$j]->{$k}->{0};  
}
```

- Using Data::Dumper

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

A vertical blue bar with a gradient, transitioning from a darker blue at the top to a lighter blue at the bottom, is positioned on the left side of the slide.

Reusable Code

Subroutines

- Can accept values
- Can return values
- Blocks of code:
Remember scope

```
my @gfs = ("Sue","Amy","Jacqui");

sub dump {
    my $dumped = shift @gfs;
    return $dumped;
}

foreach (@gfs) {
    print "I've just dumped", dump(), "\n";
}
```

```
I've just dumped Sue
I've just dumped Amy
I've just dumped Jacqui
```

Packages

- Packages define a namespace
- Define your package
- Return true

```
package Underlineit;

our $underline = "-";

sub underlineit {
    my $msg = shift;
    my $lines = $msg;
    $lines =~ s/\S/$underline/g;
    return "$msg\n$lines\n";
}

1;
```

Using a Package

- The “use” statement
- “use” vs “require”
- Setting package data
- Calling your method

```
use Underlineit;  
$Underlineit::underline = "*";  
print Underlineit::underlineit("Hello World!");
```

```
Hello World!  
*****
```

A vertical blue bar with a gradient, transitioning from a darker blue at the top to a lighter blue at the bottom, located on the left side of the slide.

Objects

A Little Theory

- What are Objects?
- Objects vs Classes
- PIE
 - Polymorphism
 - Inheritance
 - Encapsulation

Creating Classes

```
package Class1;

sub new {
    package Class1;
    my $class = shift;
    sub new {
        my $self = {};
        my $class = shift;
        bless $self, $class;
        return $class;
    }
    bless $self, $class;
    return $class;
}

sub var1 {
    my $self = shift;
    sub my $param = shift;
    if ($param) {
        my $self->{var1} = $param;
        if ($param) {
            return $self->{var1};
        }
    }
    return $self->{var1};
}

sub add {
    my $class = shift;
    sub add {
        my $num1 = shift;
        my $num2 = shift;
        return ($num1 + $num2);
    }
    my $num1 = shift;
    my $num2 = shift;
    return ($num1 + $num2);
}

sub DESTROY {
    my $self = shift;
    $self->{var1} = undef;
};

1;
```

- Create a Constructor
 - bless (ref, classname)
- Add Data Members
 - Data Access Methods
 - Private vs Public
- Add Methods
 - Instance Methods
 - Class Methods
- Create a Destructor

Creating Objects

```
use Class1;  
my $obj = Class1->new("4");
```

```
$obj->var1("6");  
print $obj->var1;
```

6

```
print Class1->add(2,4);
```

6

```
$obj->{var2} = 4;  
print $obj->{var2};
```

4

- Instantiating a Class
- Invoking a Method
- Instance Variables
 - Storing Data

Inheritance

Output:

Ref: Class2
3+4=: 7

Output:

Ref: Class2
3+4=: 17

```
package Class2;

use base 'Class1';

sub new {
    my $class = shift;
    my $self = Class1->new(4);
    bless $self, $class;
    return $self;
}

sub add {
    $self = shift;
    my $num1 = shift;
    my $num2 = shift;

    my $sum =
        $self->SUPER::add($num1,$num2);
    return ($sum + 10);
}

1;
```

```
use Class2;

my $obj = Class2->new();

print "Ref: ", (ref $obj), "\n";

print "3+4= ", $obj->add(3,4), "\n";
```

- Derive one class from another
- use base
- Overriding Methods
- SUPER

Ref: Class2
3+4= 17



Q&A