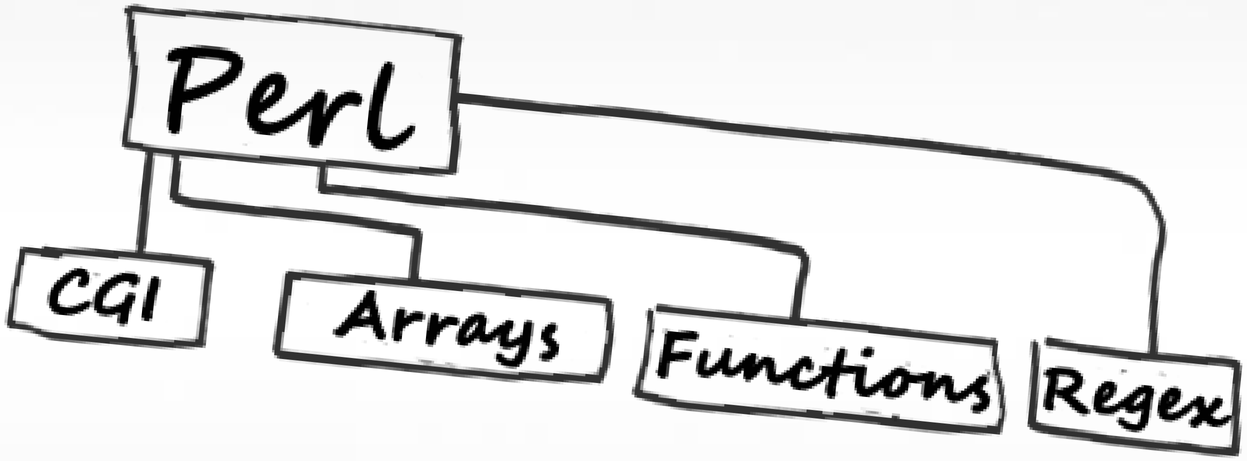
# Perl

NOW



OPEN SOURCE
DEPARTMENT

Information Technology Institute

## Course Materials

You can access the course materials via this link

http://goo.gl/YEXrGG

## Day 2 Contents

- Basic I/O
- Subroutines
- Regular expressions
- Filehandles and file tests

## Input from STDIN

```
$a = <STDIN>;  # read the next line
@a = <STDIN>;
while (defined($line = <STDIN>)) {
# process $line here
}
```

## Input from STDIN

```
while (<STDIN>) {
#like "while(defined($_ = <STDIN>)) {”
chomp;             # like "chomp($_)"
# other operations with $_ here
}
```

## Diamond Operator

```
#!/usr/bin/perl
while (<>) {
print ;
}                                   # cat command

@ARGV = ("aaa","bbb","ccc");
while (<>) {
print "this line is: $_";
}       # process files aaa, bbb, and ccc
```

## Output to STDOUT

- Use print for normal output

```
print (2+3),"hello"; # prints 5,
  ignores "hello"
print ((2+3),"hello"); # prints 5hello
print 2+3,"hello";       # also, prints
  5hello
```

- Use printf for formatted output

```
printf "%15s %5d %10.2f\n", $s, $n, $r;
```

## Types of Variable

- Perl variables are of two types
- Important to know the difference
- Lexical variables are created with `my`
- Package variables are created by `our`
- Lexical variables are associated with a code block
- Package variables are associated with a package

# Lexical Variables

- Created with `my`

-
  ```
  my ($doctor, @timelords,
          %home_planets);
  ```
- Live in a pad (associated with a block of code)
  - Piece of code delimited by braces
  - Source file
- Only visible within enclosing block
- "Lexical" because the scope is defined purely by the text

## Packages

- All Perl code is associated with a package
- A new package is created with package

```
package MyPackage;
```

- Think of it as a namespace
- Used to avoid name clashes with libraries
- Default package is called main

## Package Variables

- Live in a package's symbol table
- Can be referred to using a fully qualified name
  - `$main::doctor`
  - `@Gallifrey::timelords`
- Package name not required within own package
- Can be seen from anywhere in the package (or anywhere at all when fully qualified)

## Declaring Package Vars

- Can be predeclared with `our`

```
our ($doctor, @timelords,
     %home_planet);
```

- Or (in older Perls) with `use vars`

```
use vars qw($doctor
            @timelords
            %home_planet);
```

## local

- You might see code that uses local
- `local $variable;`
- This doesn't do what you think it does
- Badly named function
- Doesn't create local variables
- Creates a local copy of a package variable
- Can be useful in a small number of cases

## local Example

- `$/` is a package variable
- It defines the record separator
- You might want to change it
- Always localise changes

-
  ```
  {
     local $/ = "\n\n";
     while (<FILE> ) {
        ...
     }
  }
  ```

## Subroutines

- Self-contained "mini-programs" within your program
- Make it easy to repeat code
- Subroutines have a name and a block of code

-
  ```
  sub NAME {
     BLOCK
  }
  ```

## Subroutine Example

```
• sub exterminate {
     print "Ex-Ter-Min-Ate!!\n";
     $timelords--;
  }
```

## Calling a Subroutine

```
&exterminate;
exterminate();
```

## Subroutine Arguments

- Functions become far more useful if you can pass arguments to them
- `exterminate('The Doctor');`
- Arguments end up in the `@_` array within the function

-
  ```
  sub exterminate {
     my ($name) = @_;
     print "Ex-Ter-Min-Ate $name\n";
     $timelords--;
  }
  ```

## Multiple Arguments

- As `@_` is an array it can contain multiple arguments

-
  ```
  sub exterminate {
     foreach (@_) {
        print "Ex-Ter-Min-Ate $_\n";
        $timelords--;
     }
  }
  ```

## Calling Subroutines

- A subtle difference between `&my_sub` and `my_sub()`
- `&my_sub` passes on the contents of `@_` to the called subroutine
- 
```
sub first { &second };
sub second { print @_ };
first('some', 'random', 'data');
```
- You usually don't want to do that

## By Value or Reference

- Passing by value passes the ***value*** of the variable into the subroutine. Changing the argument doesn't alter the external variable
- Passing by reference passes the ***actual*** variable. Changing the argument alters the external value
- Perl allows you to choose

# By Value or Reference

- Simulating pass by value

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

- Updating `$arg1` and `$arg2` doesn’t effect anything outside the subroutine
- Simulating pass by reference

```
$_[0] = 'whatever';
```

- Updating the contents of `@_` updates the external values

## Returning Values

- Use `return` to return a value from a subroutine

-
  ```
  sub exterminate {
     if (rand > .25) {
        print "Ex-Ter-Min-Ate $_[0]\n";
        $timelords--;
        return 1;
     } else {
        return;
     }
  }
  ```

## Returning a List

- Subroutines can return lists

-
  ```
  sub exterminate {
     my @exterminated;
     foreach (@_) {
        if (rand > .25) {
           print "Ex-Ter-Min-Ate $_\n";
           $timelords--;
           push @exterminated, $_;
        }
     }
     return @exterminated;
  }
  ```

## Regular Expressions

- Patterns that match strings
- A bit like wild-cards
- A “mini-language” within Perl
- The key to Perl's text processing power
- Documented in perldoc perlre

# Match Operator

- `m/PATTERN/` - the match operator
- Works on `$_` by default
- In scalar context returns true if the match succeeds
- In list context returns list of "captured" text
- m is optional if you use / characters
- With m you can use any delimiters

## Match Examples

-
  ```
  while (<FILE>) {
      print if /foo/;
      print if /bar/i;
      print if m|http://|;
  }
  ```

## Substitutions

- `s/PATTERN/REPLACEMENT/` - the substitution operator
- Works on `$_` by default
- In scalar context returns true if substitution succeeds
- In list context returns number of replacements
- Can choose any delimiter

## Substitution Examples

-
  ```
  while (<FILE>) {
     s/teh/the/gi;
     s/freind/friend/gi;
     s/sholud/should/gi;
     print;
  }
  ```

## Binding Operator

- If we want `m//` or `s///` to work on something other than `$_` then we need to use the binding operator
- `$name =~ s/Dave/David/;`

## Metacharacters

- Matching something other than literal text
- `^` - matches start of string
- `$` - matches end of string
- `.` - matches any character (except \n)
- `\s` - matches a whitespace character
- `\S` - matches a non-whitespace character

## More Metacharacters

- `\d` - matches any digit
- `\D` - matches any non-digit
- `\w` - matches any "word" character
- `\W` - matches any "non-word" character
- `\b` - matches a word boundary
- `\B` - matches anywhere except a word boundary

## Metacharacter Examples

-
  ```
  while (<FILE>) {
     print if m|^http|;
     print if /\bperl\b/;
     print if /\S/;
     print if /\$\d\.\d\d/;
  }
  ```

## Quantifiers

- Specify the number of occurrences
- `?` - match zero or one
- `*` - match zero or more
- `+` - match one or more
- `{n}` - match exactly n
- `{n,}` - match n or more
- `{n,m}` - match between n and m

## Quantifier Examples

-
  ```
  while (<FILE>) {
     print if /whiske?y/i;
     print if /so+n/;
     print if /\d*\.\d+/;
     print if /\bA\w{3}\b/;
  }
  ```

## Character Classes

- Define a class of characters to match

```
/[aeiou]/ # match any vowel
```

- Use - to define a contiguous range

```
/[A-Z]/ # match upper case letters
```

- Use ^ to match inverse set

```
/[^A-Za-z] # match non-letters
```

## Alternation

- Use | to match one of a set of options
- `/rose|martha|donna/i;`
- Use parentheses for grouping
- `/^(rose|martha|donna)$/i;`

## Capturing Matches

- Parentheses are also used to capture parts of the matched string
- The captured parts are in `$1`, `$2`, etc…
- 
```
while (<FILE>) {
   if (/^(\w+)\s+(\w+)/) {
      print "The first word was $1\n";
      print "The second word was $2";
   }
}
```

## Returning Captures

- Captured values are also returned if the match operator is used in list context

-
  ```
  my @nums = $text =~ /(\d+)/g;
  print "I found these integers:\n";
  print "@nums\n";
  ```

## Examples

| Metacharacter | Meaning |
|---|---|
| \ | Escapes the character(s) immediately following it |
| . | Matches any single character except a newline |
| ^ | Matches at the beginning of the string |
| $ | Matches at the end of the string |
| * | Matches the preceding element 0 or more times |
| + | Matches the preceding element 1 or more times |
| ? | Matches the preceding element 0 or 1 times |

## Simple Uses of REGEX

```
if (/abc/) {
        print $_;
}
While (<>) {
        if (/abc/) {
                print $_;
                }
        }
if (/ab*c/) {
        print $_;
}
```

# Single-Character Patterns

| | |
|---|---|
| [0123456789] | # match any single digit |
| [0-9] | # same thing |
| [0-9\-] | # match 0-9, or minus |
| [a-z0-9] | # match any single lowercase letter or digit |
| [a-zA-Z0-9_] | # match any single letter, digit, or underscore |
| [^0-9] | # match any single non-digit |
| [^aeiouAEIOU] | # match any single non-vowel |
| [^\^] | # match any single character except an up arrow |

## Examples

- abc*    # matches ab, abc, abcc, abccc, abcccc, and so on
- (abc)*    # matches "", abc, abcabc, abcabcabc, and so on
- ^x|y    # matches x at the beginning of line, or y only
- ^(x|y)    # matches either x or y at the beginning of a line
- a|bc|d    # a, or bc, or d
- (a|b)(c|d)    # ac, ad, bc, or bd
- (song|blue)bird    # songbird or bluebird

## Filehandles

- Perl provides three filehandles,
  - STDIN
  - STDOUT
  - and STDERR
- Which are automatically open to files or devices established by the program's parent process (probably the shell).

## Opening & Closing Filehandle

- Examples
  - `open(DATA, "<file.txt");` *# Open file in read-only mode*
  - `open(OUT, ">outfile");` *# Open file in write mode and truncate the file before writing*
  - `open(DATA, "+<file.txt"); ");` *# Open file for reading and writing without truncating it*
  - `open(LOGFILE, ">>mylogfile");` *# Open file for appending*
  - `open(DATA,"+>>file.txt");` *# Open file for appending and reading.*
  - `close(LOGFILE);`

## Opening & Closing Filehandle

- A filehandle that hasn't been successfully opened can be used without a warning.
  - If you read from the filehandle, you'll get end-of-file right away.
  - If you write to the filehandle, the data is discarded.

```
open(DATAPLACE,">/tmp/dataplace")
|| die "Sorry, I couldn't create
/tmp/dataplace: $!";
```

## Open file for reading

```
#!/usr/bin/perl

open(DATA, "<file.txt") or die
"Couldn't open file file.txt, $!";

while(<DATA>){
   print "$_";
}
```

## Open file for reading

```
#!/usr/bin/perl

open(DATA,"<import.txt") or die "Can't
open data";
@lines = <DATA>;
close(DATA);
```

## File Tests

| File Test | Meaning |
|---|---|
| -r | File or directory is readable |
| -w | File or directory is writable |
| -x | File or directory is executable |
| -e | File or directory exists |
| -z | File exists and has zero size (directories are never empty) |
| -s | File or directory exists and has nonzero size |
| -f | Entry is a plain file |
| -d | Entry is a directory |
| -l | Entry is a symlink |

## Example

```
$name = "index.html";
if (-e $name) {
    print "I see you already have a file named $name\n";
} else {
    print "Perhaps you'd like to make a file called
$name\n";
}
if (-e "index.html" && -e "index.cgi") {
    print "You have both styles of index files here.\n";
}
foreach (@some_list_of_filenames) {
    print "$_ is readable\n" if -r; # same as -r $_
}
```