



# A Crash Course in Perl5

---

Part I: Basics

Zeegee Software Inc.

<http://www.zeegee.com/>

# Terms and Conditions

---

These slides are Copyright 2008 by Zeegee Software Inc. They have been placed online as a public service, with the following restrictions:

You may download and/or print these slides for your personal use only. Zeegee Software Inc. retains the sole right to distribute or publish these slides, or to present these slides in a public forum, whether in full or in part.

Under no circumstances are you authorized to cause this electronic file to be altered or copied in a manner which alters the text of, obscures the readability of, or omits entirely either (a) this release notice or (b) the authorship information.

# Road map

---



## Basics

- Introduction
- Perl syntax
- Basic data types
- Basic operators
- Patterns
  - Introduction
  - String matching and modifying
  - Pattern variables
- Data structures
  - LISTS and arrays
  - Context
  - Hashes
- Flow control
  - Program structures
  - Subroutines
  - References
  - Error handling
- Data
  - Input and output
  - Binary data
  - Special variables
- Object-oriented programming
  - Modules
  - Objects
  - Inheritance
  - Tying

Basics

# Introduction

---

# What is Perl?

---

- An interpreted language, based on *awk*, *sed*, *sh*, and *C*, with some *BASIC* thrown in.
  - Written originally by Larry Wall
  - Augmented by Tom Christiansen and Randal Schwartz
- The language of choice for writing CGI scripts for the World Wide Web.
- *Perl actually stands for “Pathetically Eclectic Rubbish Lister,” but don’t tell anyone I told you that -- Larry Wall*

# What is it good for?

---

- Report generation (e.g., log file analysis)
- System administration
- Email processing
- WWW applications (e.g., CGI scripts)
- Document generation/conversion (e.g., SGML translation)
- ...and much, much more...

# What makes it so darn great?

---

- Powerful regular expression matching/replacement built right into the language (*like sed and awk*)
- Variable-sized strings, arrays, and hashtables (*like Lisp*)
- Arbitrarily-nested data structures
- Object-oriented paradigm
- You can dynamically load/use your own C/C++ libraries
- You can “tie” data structures to underlying databases
- And much, much more!

# Yuck! *Is that Perl?!?*

---

- Perl balances its incredibly powerful and concise language with a syntax that is not of this earth...
- You will hate it.
- You will complain incessantly about it.
- But... comparing Perl syntax to C syntax is a lot like comparing a Boeing737 cockpit to a Ford Escort. Yeah, the Ford's easier to drive, but once you learn to fly the Boeing, you can go *anywhere, fast...*



# Oh yeah? Prove it.

---

- Okay... here's a Perl program which takes as arguments a list of filenames from the command line, and extracts “probable” email addresses. Printout is one address per line, with each address preceded by the filename and line number where it was found. If no command line args, gets input from stdin.
- Program is about 100 characters long. It took me one minute to write.

```
#!/usr/bin/perl -w
while (<>) {
    while (m/\w+\@\w[\w\.]*\w/ig) {
        print "$ARGV $.: $&\n";
    }
    close ARGV if eof;
}
```

Basics / Introduction

# The Perl philosophy

---

*There's more than  
one way to do it.*

(TMTOWTDI)

Basics

# Perl syntax

---

# Hello, world!

---

In file "hello.pl", put this...

```
print "Hello, world!\n";
```

Then, do this at the shell prompt (assuming Unix):

```
% perl hello.pl  
Hello, world!  
%
```

# Hello, world.. the program!

---

In file "hello", put this...

This should be the path to Perl on your system

The -w turns on warnings: use it always!

```
#!/usr/bin/perl -w
print "Hello, world!\n";
```

Then, do this at the shell prompt:

Make "hello" executable

```
% chmod +x hello
% ./hello
Hello, world!
%
```

# Anatomy of a Perl script

---

- A Perl script consists of a sequence of **declarations** and **statements**... hopefully, with some **comments** too...

```
#!/usr/bin/perl -w
# Comments go from the # to the end of line

# A subroutine declaration:
sub sum {
    my ($a, $b) = @_;          # gets two args
    return $a + $b;          # returns the sum
}

# A statement, ending in a semicolon:
print sum(1700, 1);
```

# Simple statements

---

- A **simple statement** is an EXPRession evaluated for its side-effects (e.g., variable assignment). It should be terminated by a semicolon:

```
$total = sum(1700, 1);
```

- Any simple statement may be followed by a single **modifier**. Possible modifiers are:

```
if EXPR          while EXPR  
unless EXPR      until EXPR
```

```
add_stuff() until ($total > 2000);
```

# Compound statements

---

- A sequence of statements that defines a scope is called a **block**. BLOCKS are usually delimited by curly braces `{}`.
- These compound statements are for control flow:

```
if (EXPR) BLOCK
```

```
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

```
LABEL while EXPR BLOCK
```

```
LABEL while EXPR BLOCK continue BLOCK
```

```
LABEL for (EXPR; EXPR; EXPR) BLOCK
```

```
LABEL foreach VAR (LIST) BLOCK
```

```
LABEL BLOCK continue BLOCK
```



Curly braces are *required* in **if**, **while**, etc. constructs!



Basics

# Basic data types

---

# Variables

---

- Variables in Perl have names of this form:

First character is any of: <b>A-Z, a-z, _</b>	Zero or more additional characters in: <b>A-Z, a-z, 0-9, _</b>
--	---

- Variable names are *case-sensitive*.
- Variable names are explicitly typed, and there are three basic types:


**\$x** a *scalar*: a simple thing: text string, number, etc.

**@x** an *array*: holds zero or more scalars

**%x** a *hash*: a "lookup table" that maps keys to values

# Types define namespaces

---

 Scalars, arrays, hashes (and subroutines) all exist in a separate namespace... so `$x`, `@x`, and `%x` are all different variables...

**But avoid doing this in practice... it's very confusing!**

# Scalars

---

- **Scalars** hold strings of zero or more bytes
- **Scalar variables** are signified by a \$ before the variable name; e.g., `$name`
- Use scalars to hold:

- Text strings

```
$msg = "Hello, world!\n";
```

- Integers

```
$deg = 360;
```

- Floating point numbers

```
$pi = 3.14159;
```

- Arbitrary binary data

```
$data = "\012\000\011";
```

# Scalar Variables (cont'd)

---



Scalar variables grow dynamically as you add to them.  
No need to allocate the memory yourself!

```
$alpha = "abc";  
$alpha .= "defghijk";  
$alpha .= "lmnopqrstuvwxyz";  
$alpha = $alpha x 1024;
```

`.=` adds text to the end of a string  
`x` duplicates a string *n* times

# Numeric Literals

---

- **Integer** numeric literals:

42	decimal integer
052	octal integer (has a leading <b>0</b> )
0x2A	hexadecimal integer (has a leading <b>0x</b> )
<u>2</u> <u>_</u> <u>2</u> <u>6</u> <u>7</u> <u>_</u> <u>7</u> <u>0</u> <u>9</u>	decimal integer, with underlines for clarity

- **Floating-point** numeric literals:

42.0	floating point
4.2E1	scientific notation ( <b>E</b> or <b>e</b> )
4.2E+1	explicit <b>+</b> is okay (and so is a <b>-</b> )
.42E+2	any floating point before the E is okay
4.2E+010	that's <i>not</i> an octal exponent: it's ten!

# String Literals, Double-Quoted

---

- **Double-quoted** string literals are much like those in C, with \-escapes recognized:

<code>\n</code>	newline	<code>\377</code>	byte value (3 octal digits)
<code>\r</code>	return	<code>\xFF</code>	byte value (x + 2 hex digits)
<code>\t</code>	tab	<code>\c [</code>	control character (c + 1 char)
<code>\f</code>	form feed	<code>\e</code>	escape
<code>\\</code>	a backslash	<code>\"</code>	quote char (if " is quote char)

...and many more...

```
print "Hello, \n\"world\"!";
```

```
Hello,  
"world"!
```

# String Literals, Double-Quoted

---

- In double-quoted strings, scalar (and array) variables are *interpolated*, meaning that their values are inserted:

```
$name = "Jim";  
print "He's dead, $name!\n";
```

```
He's dead, Jim!
```

- To suppress interpolation, put a `\` in front of the `$` (or `@`):

```
$fee = 300;  
print "That'll be \$ $fee.\n";
```

```
That'll be $300.
```



# Escapes for interpolation

---

- Certain escape sequences give you control over how Perl will interpolate scalars into your strings:

<code>\l</code>	lowercase next char	<code>\u</code>	uppercase next char
<code>\L</code>	lowercase until <code>\E</code>	<code>\U</code>	uppercase until <code>\E</code>
<code>\Q</code>	quote <b>regular expression metacharacters</b> until <code>\E</code>		
<code>\E</code>	end case modification		

```
$first = "james";  
$last  = "kirk";  
print "Hello, \u$first \U$last\E!\n";
```

```
Hello, James KIRK!
```

# More on interpolation

---

- If you want to interpolate a scalar variable into a string, but the character after the variable name can be mistaken for part of the variable name, you can use { } to distinguish the variable name:

```
$thing = "tribble";  
print "Too many things!";
```

# String Literals, Here-is

---

- For very long strings, especially ones with newlines, you may find the *here-is* syntax more to your liking:

```
print <<EOF;
```

The string begins on this line, and can have \$variables, blank lines, etc...

...and it doesn't end until a line containing nothing more than the token after the << ("EOF", in this case) is encountered.

**EOF**

```
$msg = <<STREND;
```

Basically, it works like a funny double-quote.

**STREND**

# String Literals, Single Quoted

---

- **Single-quoted** string literals are similar, except that...
  - A backslash (\) may only be used to escape the quote character (*usually* ' ) or another backslash
  - Variables are *not* interpolated

```
$name = "Scotty";  
print '\\Hello, $name!\\n';
```

```
\\Hello, $name!\\n
```

# String Literals, Barewords

---

- Strings consisting solely of upper- and lowercase characters and underscores may be used without quotes wherever a string literal is expected:

```
$level = DEBUG;      # same as if we used 'DEBUG'
```

- **Always avoid** all-lowercase barewords: they may conflict with future reserved words (Perl will warn you):

```
$level = debug;      # may someday be a reserved word!
```

# Beware the Bare

---

- **Beware:** if you use a bareword with the same name as a function, *Perl will call the function:*

```
print "!", Debug, "\n";           # prints !Debug
sub Debug { return 123 }          # declares Debug()
print "!", Debug, "\n";           # prints !123
```

- **Avoid barewords** for now, and in general... later on, we'll show you a few places that you can use them safely and conveniently.

# The empty string

---

- If you follow a single- or double-quote character immediately by that same quote character, you get an **empty string**. It has a length of 0.

```
$name = "";           # double-quote, double-quote  
$name = ";           # single-quote, single-quote
```

- This is *not* the same as the "undefined" value which a scalar has before you assign to it.

<u>Perl scalar</u>		<u>C string (char *)</u>
undefined	⇔	NULL pointer
empty string	⇔	non-NULL pointer to a \000 char

# The undefined value

---

- If you use a scalar variable before giving it a value, it has the special value of `undef`
- You can assign the value explicitly:

```
$x = undef; # one way
```

```
undef $x;          # another way
```

- You can also test for it:

```
if (defined($x)) { ... }
```



# Using undefined

---

- When used in a **string** context (e.g., in concatenation or printing), `undef` behaves like the **empty string**.
- When used in a **numeric** context (e.g., in addition), `undef` behaves like **zero**.
- When used in a **boolean** context (e.g., in an "if" test), `undef` is **false**.
- **Perl will warn you** if you use `undef` in a string or numeric context.

# Boolean equivalents

---

- In a boolean context (e.g., inside an "if" test), here's what the following scalar values behave as:
  1. Any string is **true**, except for the empty string and the one-character string '0' (that's ASCII 0x30).
  2. Any number is **true**, except for 0 (integer or float).
  3. Any **reference** (pointer to an object) is **true**.
  4. Any undefined value is **false**.

```
$value = "Scotty";  
if ($value) {  
    print "value is defined, nonempty, and nonzero\n";  
}
```

Basics

# Basic operators

---

# Arithmetic operators

---

<code>\$a + 1</code>	Addition
<code>\$a - 2</code>	Subtraction
<code>\$a * 3</code>	Multiplication
<code>\$a / 4</code>	Division... note that if both sides are integers, then integer division will be done; else, floating-point division will be done.
<code>\$a % 5</code>	Modulus: in this case, the remainder when <code>\$a</code> is divided by 5 using integer division.
<code>\$a ** 6</code>	Exponentiation (negative / real exponents are ok!)
<code>-\$a</code>	Negation

# Autoincrement/autodecrement

---

- The `++` and `--` operators work on integers just as in C:
  - If placed *before* a variable, they increment/decrement the variable *before* returning the value.
  - If placed *after* a variable, they increment/decrement the variable *after* returning the value.

```
$i = 3;  
$j = ++$i; # sets $i to 4, $j to 4  
  
$i = 3;  
$j = $i++; # sets $i to 4, $j to 3
```

# String autoincrement

---

 If you *autoincrement* a scalar which has never been used in a numeric context and whose value matches the pattern...

Zero or more of: **A-Z, a-z**

Zero or more of: **0-9**

...the increment is done as a *string*, preserving each character within its range, with carry:

```
print ++($i = '99');      # prints '100'  
print ++($i = 'a0');      # prints 'a1'  
print ++($i = 'Az');      # prints 'Ba'  
print ++($i = 'zz');      # prints 'aaa'
```

- Autodecrement, however, is *not* magical!

# Bitwise boolean operators

---

`$a & $b` **Bitwise AND:** a bit in result is on only if it is on in *both \$a and \$b*.

`$a | $b` **Bitwise OR:** a bit in result is on only if it is on in *either \$a or \$b or both*.

`$a ^ $b` **Bitwise XOR:** a bit in result is on only if it is on in *either \$a or \$b, but not both*.

`~$a` **Bitwise NOT:** a bit in the result is on if it is *off* in *\$a*

```
$a = 0xA5;           # bits = 1010 0101
$b = 0x2B;           # bits = 0010 1011
printf "%X ", $a & $b; # bits = 0010 0001 (x21)
printf "%X ", $a | $b; # bits = 1010 1111 (xAF)
printf "%X ", $a ^ $b; # bits = 1000 1110 (x8E)
```

# Shift operators

---

`$a << $b`

**Left shift:** returns value of *\$a* with its bits *shifted left* by the number of places specified by right argument. *Both arguments should be integers.*

`$a >> $b`

**Right shift:** returns value of *\$a* with its bits *shifted right* by the number of places specified by right argument. *Both arguments should be integers.*

```
$a = 39;                # bits = 00100111

printf "%d ", $a >> 1;  # bits = 00010011  (19)
printf "%d ", $a >> 0;  # bits = 00100111  (39)
printf "%d ", $a << 1;  # bits = 01001110  (78)
printf "%d ", $a << 2;  # bits = 10011100 (156)
```



# Logical boolean operators

---

- Much like in C...

`$a && $b`      **Logical AND:** true if both `$a` and `$b` are true

`$a || $b`      **Logical OR:** true if either `$a` or `$b` is true

`!$a`      **Logical NOT:** true if `$a` is false

- Evaluation of `&&` and `||` is done left-to-right, and it "short circuits" as soon as result can be determined.
- Unlike C, `&&` and `||` return the **last value evaluated**:

```
$home = getenv('HOME') ||  
       getenv('LOGDIR') ||  
       die "Couldn't get home directory!";
```

# Short if-thens with &&/||

---

- Thanks to short-circuiting, && and || can be used as shorthand for *if-then* constructs. This is very common:

```
open(LOG, ">test.log") || die("no log file!");  
$verbose && print LOG "Logging begun\n";
```



**Remember precedence!** You may need to use parentheses to prevent problems:

```
defined($x) || $x = 'DEFAULT';
```



**NO!**

```
defined($x) || ($x = 'DEFAULT');
```

**YES!**

# Better if-thens with and/or

---

- To avoid problems and enhance readability, there are several boolean operators that make life easier:

`$a and $b`    Identical to `&&`, but with lower precedence

`$a or $b`    Identical to `||`, but with lower precedence

`$a xor $b`    Exclusive-or, with low precedence

`not $a`    Identical to `!`, but with lower precedence

- Now we can write:

```
defined($x) or $x = 'DEFAULT';
```

# Numeric relational operators

---

- \$a < \$b      Numeric "less-than"
- \$a > \$b      Numeric "greater-than"
- \$a <= \$b     Numeric "less-than-or-equal"
- \$a >= \$b     Numeric "greater-than-or-equal"
- \$a == \$b     Numeric equality
- \$a != \$b     Numeric inequality
- \$a <=> \$b    Numeric **comparison**: returns -1, 0, or 1...

<u>if:</u>	<u>then:</u>	<u>result:</u>
\$a < \$b	result < 0	-1
\$a = \$b	result = 0	0
\$a > \$b	result > 0	1

# String relational operators

---

- \$a **lt** \$b String "less-than"
- \$a **gt** \$b String "greater-than"
- \$a **le** \$b String "less-than-or-equal"
- \$a **ge** \$b String "greater-than-or-equal"
- \$a **eq** \$b String equality
- \$a **ne** \$b String inequality
- \$a **cmp** \$b String **comparison**: returns -1, 0, or 1...

<b><u>if:</u></b>	<b><u>then:</u></b>	<b><u>result:</u></b>
\$a lt \$b	result < 0	-1
\$a eq \$b	result = 0	0
\$a gt \$b	result > 0	1

# Using relational operators

---



**Be careful!** Only use the numeric relational operators when comparing scalars with **integer** or **floating point** values:

```
if ($name == 'Bones') {
```



**NO!**



Note that the numeric operators can return **very different results** from the string operators:

```
$a = 42;  
$b = 101;  
  
print $a <=> $b, "\n"; # prints -1  
print $a cmp $b, "\n"; # prints 1
```

# Conditional operator

---

- Ternary "?:" works like an if-then-else:

*boolean-test ? value-if-true : value-if-false*

- For example:

```
$warpspeed = $emergency ? 9 : 1;
```

- Can also be assigned to if the second and third operands are legal lvalues:

```
($useA ? $a : $b) = $value;
```

## Basics / Basic operators / String

# String operators

---

`$a . $b`      **Concatenation:** returns single string consisting of *\$a* followed by *\$b*.

`$a x $b`      **Repetition:** returns single string consisting of *\$b* repetitions of *\$a*. The left operand (*\$a*) is the string, and the right operand (*\$b*) *must* be an integer.

```
print '-' x 80;           # print a row of 80 dashes

$a = "hello";
$b = "world";
$c = $a . '*' . $b;     # sets $c to "hello*world"
```



# Quote operators

---

- Sometimes, using "standard" quotes is awkward:

```
print "<img src=\"logo.gif\" alt=\"logo\">";
```

- But in Perl, you can use the **generic quote operators**, and choose your own quote character:

```
print qq!!;
```

```
print qq();
```

```
print qq<>;
```

- **Note:** the left-paren-like quote characters (, [, <, { are terminated by their counterparts, and they *must* balance!

# Quote operators

---

- Perl supplies the following generic quote operators:

Customary	Generic	Meaning	Interpolates?
' '	q{ }	Literal	no
" "	qq{ }	Literal	yes
` `	qx{ }	Command	yes
	qw{ }	Word list	no
//	m{ }	Pattern match	yes
	s{ }{ }	Substitution	yes
	tr{ }{ }	Translation	no

- You can use any character in place of the { and }, but remember: the following must be in *balanced pairs*:

{ }    ( )    < >    [ ]

# Assignment operators

---

- **Binary =** is the ordinary assignment operator.
- Can be coupled with other operators, as in C:

`$a += 2` is equivalent to `$a = $a + 2`

- Recognized are:

<code>**=</code>	<code>+=</code>	<code>*=</code>	<code>&amp;=</code>	<code>&lt;&lt;=</code>	<code>&amp;&amp;=</code>
	<code>-=</code>	<code>/=</code>	<code> =</code>	<code>&gt;&gt;=</code>	<code>  =</code>
	<code>.=</code>	<code>%=</code>	<code>^=</code>		
		<code>X=</code>			

- All have the same precedence

# Range operator, list context

---

- In a **list context** (e.g., assigning to an array), **binary ".."** returns an array of values from the left to right operand:

```
@numbers = (1 .. 100);      # array of 100 elements
```

- Uses magical autoincrement if operands are strings:

```
@alphas = ('A'..'Z', 'a'..'z');
```

- Can be used in `foreach` loops:

```
foreach (1 .. 100) { print $_, "\n"; }
```



Be careful, though... it creates a temporary array, so it can burn a lot of memory (consider `1..1000000`)!

# Range operator, scalar context

---

- In a **scalar context** (e.g., inside an *if* condition), **binary** `".."` returns a boolean which can be used as a range test:
  - Initially...      FALSE      ...until left operand becomes TRUE
  - Then stays...    TRUE      ...until right operand becomes TRUE
  - Then...          FALSE      ...again
- If either operand is a numeric literal, operand is implicitly compared to `"$."` (the current line number):

```
if (101 .. 200) { print; }      # print lines 101..200
```

```
if ((101 <= $.) && ($. <= 200)) { print; }
```