# A Crash Course in

# Perl5

## Part 7: Web development in Perl

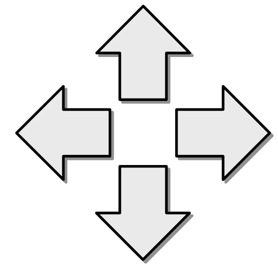
Zeegee Software Inc.
http://www.zeegee.com/

# 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.

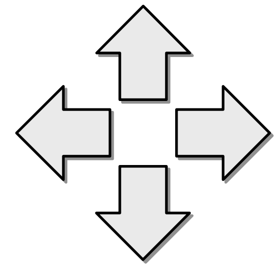


# OUTLINE

This class is devoted to one of the most popular Perl topics: how do you develop a (*fill in the blank*) program for the World Wide Web?

As standards converge, so does software. Today, you'll see some of the standard Perl tools - and approaches - for building WWW applications.

- Resources
- HTML/SGML
- CGI scripting
- WWW clienting
- Mail and MIME



# RESOURCES

There are numerous resources on the Web for Perl developers, especially ones who do Web-related work.

These are just a few.

Remember the ecological core of OO programming: **reduce**, **reuse**, **recycle**.

- Perl Language Home Page
- Perl5 Module List
- CPAN
- USENET newsgroups
- The Perl Institute

Copyright © 1996, 2000 Zeegee Software Inc.

# Perl Language Home Page

- The place to go to find almost anything perly:

**http://www.perl.com**

- From here, you can find hyperlinks to virtually every other resource...

# Perl5 Module List

- Before you spend precious time implementing and testing that all-important module, why not see if someone's already done the work for you?
- Scads of **free** semi-standard modules contributed by Perlers around the world are listed, by topic, at:

**http://search.cpan.org**

- Each entry hyperlinked to author's info in the Who's Who list: from there, you can email the author with questions, or go to their CPAN directory and download the software!

Resources

# CPAN

- The **Coordinated Perl Archive Network**: a set of anonymous FTP sites which contain modules from the Module List, latest Perl builds/patches/ports, and more!
- Mirror sites are all around the world... to find the site nearest you, go to:

**http://cpan.org**

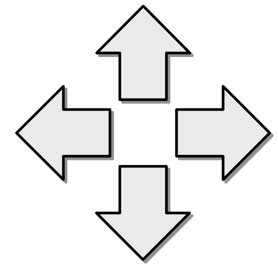
- Master site is the **Perl Authors' Upload SErver** (PAUSE), in Berlin.

Resources

# USENET newsgroups

- **comp.lang.perl.misc:** Newbie questions, postings related to general Perl issues, pleas for help, etc. UNMODERATED.
- **comp.lang.perl.modules:** Postings related to software in the Perl5 Module List. UNMODERATED.
- **comp.lang.perl.announce:** Announcements of new/improved modules, releases of Perl, and other stuff of general interest. MODERATED.

**The Perl community tends to self-moderate the unmoderated newsgroups: please keep postings brief and professional, with good subject lines... or risk being flamed.**



# HTML/SGML

HTML (a special case of SGML) is everywhere these days...

Your project may need to generate HTML, either for static WWW pages or for dynamic CGI output.

You may even need to parse HTML, to extract information.

- How to generate HTML
- How to build syntax trees
- How to output syntax trees

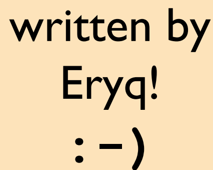
Copyright © 1996, 2000 Zeegee Software Inc.

# Why use modules?

*Why would I need modules to output HTML? Can't I just use print statements?*

- You could... but accidents happen. Leaving out a "<" here, a quote-mark there, forgetting to escape text with unsafe characters in it... all these can cause a browser to choke on your output. *And it might be someone else's browser - not yours - that shows the problem!*
- Generally, the modules you want are in the **HTML::** module tree.

his is 24 pt






# HTML::Stream

- Allows you to open an "HTML stream" on any FileHandle, or object which responds to print() message.

```
use HTML::Stream;

$HTML = new HTML::Stream \*STDOUT;
```

- Can intermix HTML output with ordinary printing.
- Provides HTML conversion functions, and 3 OO interfaces for outputting HTML: the **vanilla**, **chocolate**, and **strawberry** interfaces.

Copyright © 1996, 2000 Zeegee Software Inc.

# HTML::Stream, functions

```
use HTML::Stream qw(:funcs);

# Use html_tag to generate text for <TAGS>.
# Use html_escape to escape unsafe text.

print html_tag(A, HREF=>$url);
print '&copy; 1996 by', html_escape($myname), '!';

print html_tag('/A');
```

# HTML::Stream, vanilla

```
$HTML = new HTML::Stream \*STDOUT;

# Output an begin-hyperlink, and an image tag:
$HTML->tag('A', HREF=>$url);
$HTML->tag('IMG', SRC=>$gifurl, ALT=>"LOGO");

# Output some text, safely! Can also use t().
$HTML->text("Is 2 > 1, & if so, so what?");

# Output a copyright entity. Can also use e().
$HTML->ent('copy');

# End the hyperlink:
$HTML->tag('/A');
```

# HTML::Stream, chocolate

```
$HTML = new HTML::Stream \*STDOUT;

# Output same as previous slide.  Note method chaining!
$HTML -> A(HREF=>$url)
      -> IMG(SRC=>$gifurl, ALT=>"LOGO")
      -> t("Is 2 > 1, & if so, so what?")
      -> e('copy')
      -> _A;

# Only known HTML tags are turned into methods...
#    so typos like this fail at run-time:
$HTML -> IMGG(SRC=>$gifurl, ALT=>"LOGO")
```

HTML/SGML

# HTML::Parser

- Class for building a custom HTML parser. You define a subclass, and just override a few methods:

| | |
|---|---|
| **start**($tag, $attr) | when a <TAG> is seen |
| **end**($tag) | when a </TAG> is seen |
| **text**($text) | when plain text is seen |
| **comment**($comment) | when a comment is seen |

- Overrides get invoked automatically during parsing:

```
use HTML::MyParser;    # subclass of HTML::Parser;
$p = new HTML::MyParser;
$p->parse_file("foo.html");
```

Copyright © 1996, 2000 Zeegee Software Inc.

# HTML::HeadParser

- Lightweight HTML::Parser subclass, for just the <HEAD>...</HEAD> part of a document:

```
use HTML::HeadParser;

# Create new parser, and parse text:
$p = HTML::HeadParser->new;
$p->parse($text) and print "not finished";

# Access some common tags:
$p->header('Title')  # get <title>....</title>
$p->header('Base')   # get <base href="http://...">

# Access <meta http-equiv="Foo" content="...">:
$p->header('Foo')
```

Copyright © 1996, 2000 Zeegee Software Inc.

HTML/SGML

# HTML::TreeBuilder

- Heavy-duty HTML::Parser subclass
- Builds a ***syntax tree*** from the HTML as it parses. This is a tree-like structure of **HTML::Element** objects, which can be navigated, manipulated, and printed back out as HTML text.
- Other modules also deal with syntax trees...

Written by
Gisle Aas &
Tim Bunce

## HTML/SGML

# HTML::AsSubs

- Another nifty way to do things. Creates an HTML syntax tree (of HTML::Elements) by nested function calls...

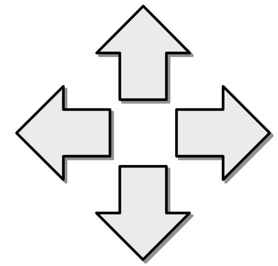
```
use HTML::AsSubs;
$h = body(
      h1("A heading"),
      p("The first paragraph, which contains a ",
         a({href=>'link.html'}, "link"),
         " and an ",
         img({src=>'img.gif', alt=>'image'}),
         "."
      ),
);
print $h->as_HTML;
```

# HTML::Formatter

- Abstract class. Subclasses take a syntax tree and output the tree in another format...
  - HTML::FormatText
  - HTML::FormatPS
- Don't see what you need? Define your own subclass!

Copyright © 1996, 2000 Zeegee Software Inc.

# CGI SCRIPTING



Many information systems want to provide WWW interfaces to their users.

Applets are great for UI at this point, but are still too restricted in power and audience.

For simple interfaces with sophisticated server-side processing, CGI is still a good, universal choice...

- Introduction to CGI scripting
- Writing a CGI script
- The many Perl CGI modules
- FAQs
- Tricks of the trade

# CGI?

**Common** CGI is universal in the WWW. Any browser that can generate an HTTP request can run a CGI script. You needn't be *too* concerned with browser compatibility.

**Gateway** A translation layer between one information system and another is often called a *gateway*. CGI's strength lies not in what it does by itself, but in the potential access it offers to other systems such as database and graphic generators.

**Interface** CGI isn't a library or a product: it's a *formally defined interface* between HTTP servers and CGI programs. HTTP servers on any system can theoretically run CGI programs written in any language: Perl, C, etc.

*Inspired by Andy Oram and Linda Mui.*

# Why use Perl for CGI?

Most CGI applications involve:

- Accessing external programs/databases
- Manipulating data
- Generating text or HTML

All of these are **incredibly easy** with Perl! As it even says in the first 3 lines of the Perl manpage...

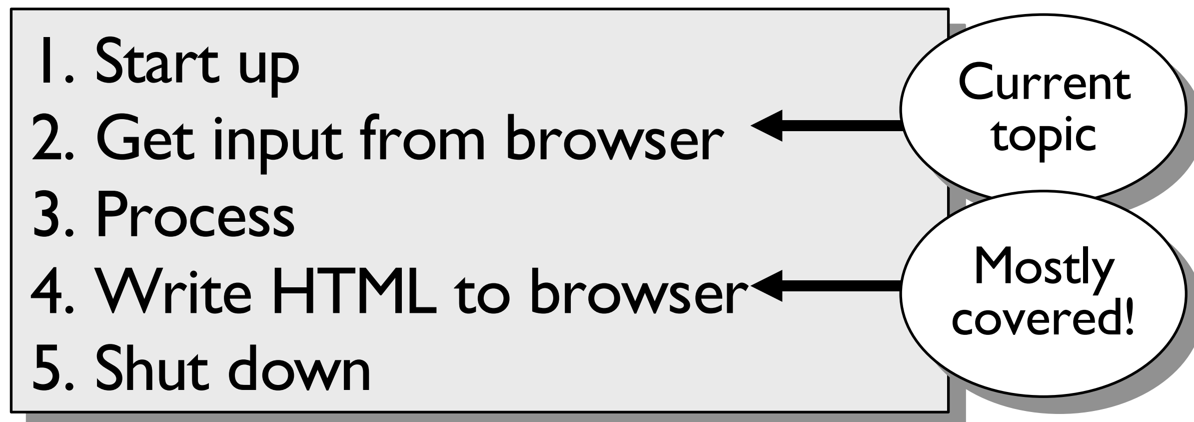
> Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information.

And that was just for Perl4... it's not just for text anymore!

Copyright © 1996, 2000 Zeegee Software Inc.

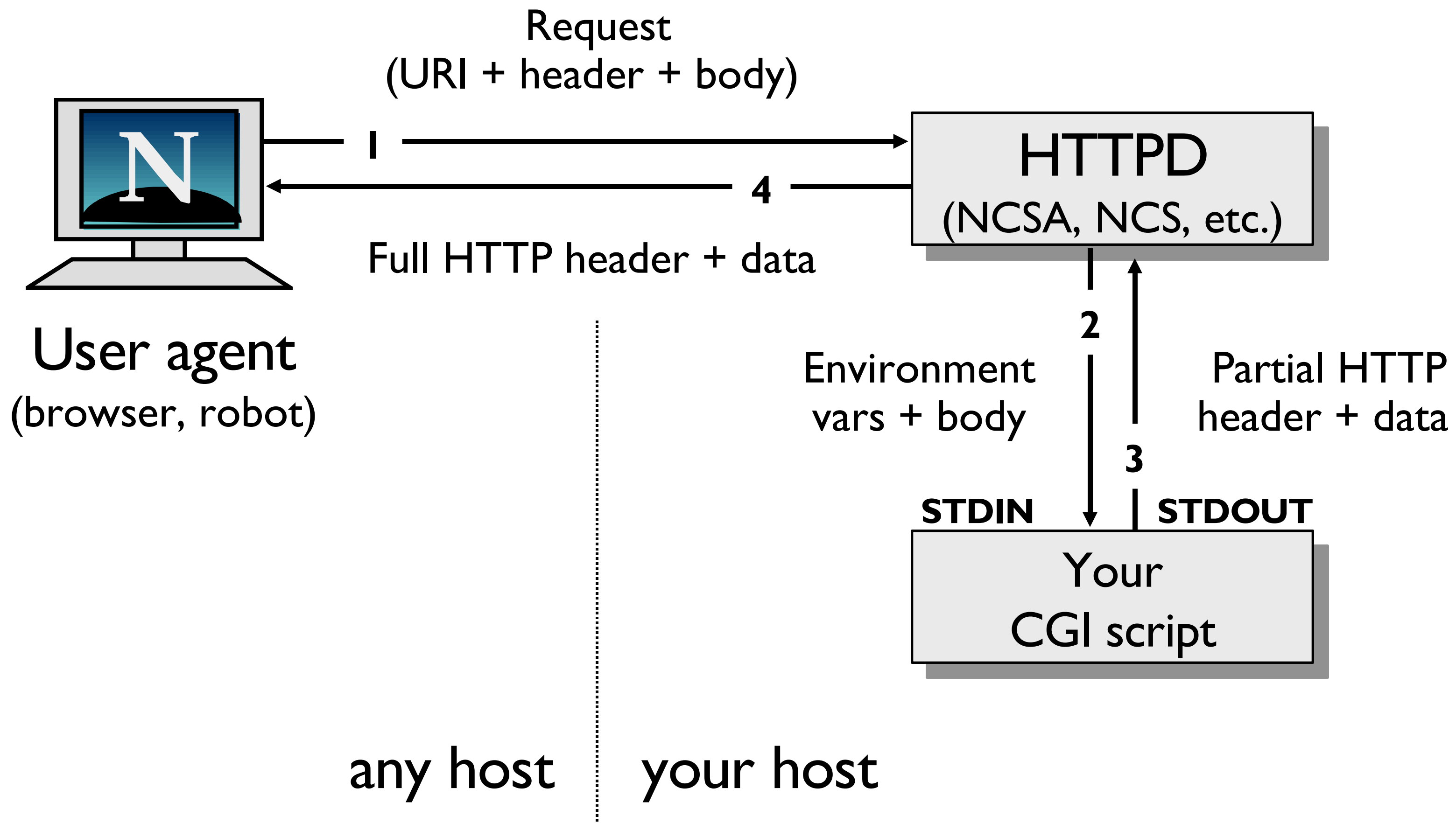
# Overview of CGI

- ***CGI scripts*** (or ***CGI programs***) are run by an HTTP server in response to a browser loading the script's URL.
- Generally have a short lifespan... most CGI scripts do some variation of the following...




Copyright © 1996, 2000 Zeegee Software Inc.

# HTTP and CGI




Copyright © 1996, 2000 Zeegee Software Inc.

# When are CGI scripts run?

- CGI scripts may run when...
  - A **user submits an HTML form** to the script.
  - A **user follows a hyperlink** to the script's URL.
  - An **automated process** (robot, spider, indexer, etc.) loads the script's URL or submits data to it.
- Any software that causes a CGI script to be invoked through HTTP is called a ***user agent***. In fact, any software which does any WWW clienting *at all* is called a *user agent*, even when there's no interactive user!

# CGI script URLs

- CGI scripts *used* to be kept in a single, special directory, and their URLs generally started with */cgi-bin* after the hostname:

  *http://your.host.com/cgi-bin/scriptname*

- *Nowadays*, common practice to have any file with a *.cgi* extension be recognized as a CGI script:

  *http://your.host.com/~user/hello.cgi*

- Some sites even allow *index.cgi* for directories... so you can't tell from the URL that you're running a script!

# How forms become CGI input

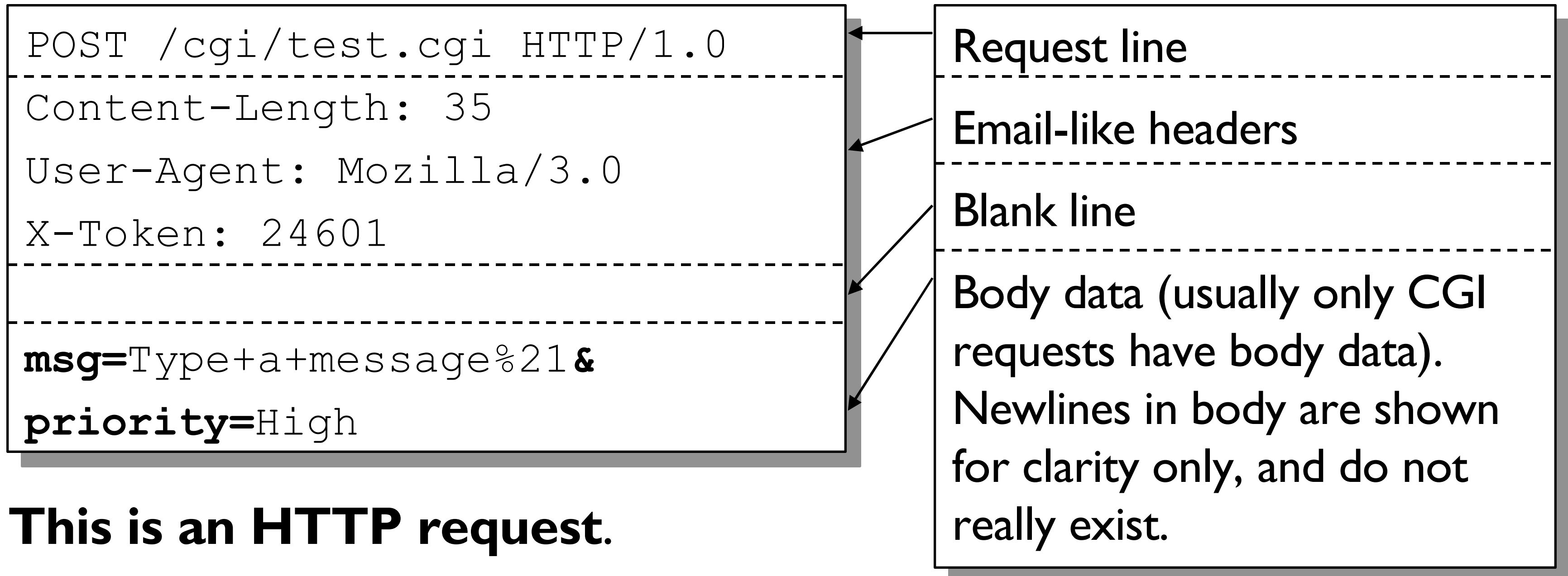
1. This is a simple HTML page your browser might load. It contains a form which is submitted to the CGI script http://some.host/test.cgi:

```
<FORM ACTION="http://some.host/cgi/test.cgi"
      METHOD=POST>
   <INPUT NAME="msg" VALUE="Type a message!">
   <SELECT NAME="priority">
     <OPTION>High
     <OPTION>Low
   </SELECT><BR>
   <INPUT TYPE=SUBMIT>
</FORM>
```

Copyright © 1996, 2000 Zeegee Software Inc.

# How forms become CGI input

2. When you press the *SUBMIT* button, the state of the form is turned into an HTTP request that your browser sends to the HTTPD given in the *ACTION* URL.




**This is an HTTP request**.

# How forms become CGI input

3. The HTTP then takes that input, uses it to set up some environment variables, and then tries to run the named CGI script. Notice that REQUEST_METHOD...

```
REQUEST_METHOD: POST
HTTP_USER_AGENT: Mozilla/3.0
REMOTE_HOST: your.host.com
REMOTE_ADDR: 128.183.20.192
CONTENT_LENGTH: 35
```

A part of the environment passed into the CGI script.

```
msg=Type+a+message%21&
priority=High
```

The query is sent to the script's STDIN if a POST request, and is just put in the `QUERY_STRING` environment variable if a GET request.

Copyright © 1996, 2000 Zeegee Software Inc.

# The *GET* request method

- ***GET requests*** usually result from loading the script's URL directly, or by following a hyperlink:

```
http://hostname/scriptname?query
```

*?query* is optional

- Can also result from hitting SUBMIT on an HTML form... the **query** is a string that encodes the state of the form:

```
<FORM METHOD=GET
      ACTION="http://hostname/scriptname">
```

- In GET requests, the **query** (if any) is accessed by the CGI through the QUERY_STRING **environment variable**... and these are limited in size!!!

# The *POST* request method

- ***POST requests*** usually result from hitting SUBMIT on an HTML form... the **query** is a string that encodes the state of the form:

```
<FORM METHOD=POST
      ACTION="http://hostname/scriptname">
```

- In POST requests, the **query** (if any) is read by the CGI from its **standard input** (STDIN). That means there's **no real limit** on how long the posted query can be!

Copyright © 1996, 2000 Zeegee Software Inc.

# Gathering form input

- Important: this often tells us what the user wants to do!
- As we've seen, input from a user (e.g., form input) might be provided to the CGI script in several places:
  - **Environment variables** set up by the HTTPD: **%ENV**
  - The **standard input** of the script: **STDIN**
  - The **command line** (old style ISINDEX): **@ARGV**

Note that the same input can come from different places under different circumstances! CGI scripts should transparently look in *all* the necessary locations!

Copyright © 1996, 2000 Zeegee Software Inc.

# CGI-encoded input

- Input to CGI scripts is of the form

$$key_1\mathbf{=}val_1\mathbf{\&}key_2\mathbf{=}val_2\mathbf{\&}...\mathbf{\&}key_n\mathbf{=}val_n$$

- Each key/value is encoded so that special characters can be placed in URLs without problems:
  - Single spaces are turned into + signs.
  - Alphanumerics [A-Za-z0-9] are left alone.
  - Any other kind of character is usually turned into the sequence %XX, where XX is the hexadecimal code (0-255).

```
Hello, nurse!
```

```
Hello%2C+nurse%21
```

# STDIN, STDOUT, STDERR?

In a CGI environment, **STDERR** usually points to the HTTPD error log. You can advantage of this by outputting debug messages, and then checking the log file later on:

```
sub debug { print STDERR "$0 debug: ", @_, "\n" }
```

Both **STDIN** and **STDOUT** point to the client (the browser). Well, sort of...

**STDIN** really points to the HTTPD which got the client's original request and invoked the CGI script.

**STDOUT** usually point back to that HTTPD, which fixes up the outgoing HTTP header.  With Non-Parsed Header (NPH) scripts, STDOUT *might* lead right to the client.

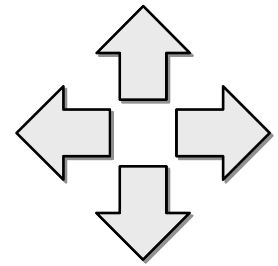
# HTTP output

- CGI scripts output their result document to their standard output (STDOUT).

  This document *must* be preceded by a simple **HTTP CGI header** (a MIME header, really) that tells what type the document is... so the browser can display it properly:

```
# The HTTP header:
print "Content-type: text/html\r\n";
print "\r\n";

# The document...
print "<HTML><H1>Hi!</H1></HTML>\n";
```

# WRITING A SCRIPT



So you think you're ready to try some of this stuff out for real, eh?

- Coding it
- Hello, world
- Installing it

Copyright © 1996, 2000 Zeegee Software Inc.

# Preflight checklist...

To help **find bugs/prevent security holes**, the shebang line should *always* turn on **warnings** and **taint checking**:

```
#!/usr/bin/perl -Tw
```

- To find nasty **hidden bugs**, you should *always*:

```
use strict;
```

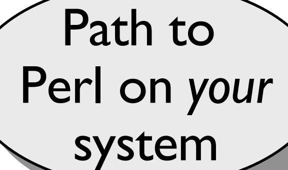
- Finally: find out where your **HTTPD's error log** is... if you get the dreaded "500 Server Error", look at the end of this file to see what went wrong.

*Okay!  Now let's do some coding...*

Copyright © 1996, 2000 Zeegee Software Inc.

# A "hello world" CGI script

```
#!/usr/bin/perl -Tw
use strict;

# Output the HTTP header for a plain text document:
print "Content-type: text/plain\r\n";
print "\r\n";

# Output our environment as the document:
print "Environment variables\n";
my $var;
foreach $var (sort keys %ENV) {
   print $var , " = ", $ENV{$var}, "\n";
}
print "BYE!\n";

# Not needed, but nice:
exit 0;
```



Copyright © 1996, 2000 Zeegee Software Inc. 

# Where to put it

- CGI scripts are often kept in **special directories**:
  - HTTPD knows to *run* such files instead of *serving* them.
  - Directories normally not writeable by average users.
  - Usually at least one main directory, called *cgi-bin*.
- *Nowadays*, some sites allow any file with a *.cgi* extension to be recognized as a CGI script. You can put your scripts under **your own public-html** directory:

```
mv hello.cgi ~/public_html/hello.cgi
```

- **Bottom line**: ask your sysadmin or webadmin.

Copyright © 1996, 2000 Zeegee Software Inc.

# Set the file permissions

Now make the script readable and executable by world:

```
chmod 0755 ~/public_html/hello.cgi
```

*This is vital*. If you forget this step, you'll get a

**Permission denied**

error when you try to load your script from a browser.

# Try it out!

Open up a browser, and load the script's URL. In the current example, this might be:

```
http://hostname/~user/hello.cgi
```

Where *hostname* is the name of the host the script is on, and *user* is your user id on that host. Again, you may need a webadmin or other seasoned individual to help you with the URL.

You should get back a plain text document, with a lot of environment variables in it. If you didn't, well... we'll discuss troubleshooting later...

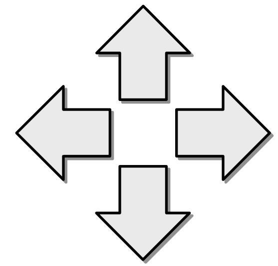
Copyright © 1996, 2000 Zeegee Software Inc.

# *PROJECT*

- Create the simple HTML form shown previously, and modify its action so it points at your *hello.cgi* script. Load it into your browser, and submit the form. What do you see? How does this differ from the output when you just load the script URL?
- Modify the *hello.cgi* script so it outputs an HTML form which, when submitted, will go back to the script. Do you need to hardcode your script's URL... or do the environment variables hint at a better way?

Copyright © 1996, 2000 Zeegee Software Inc.

# PERL CGI MODULES

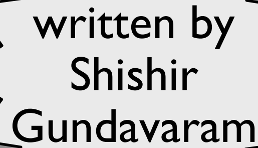


That "hello world" script is nice... but what if we want the user's form input?

Don't despair! Much of the really nasty work has already been done for you...

- CGI_Lite
- CGI.pm
- The CGI:: family
- CGI::Carp

Copyright © 1996, 2000 Zeegee Software Inc.






# CGI_Lite

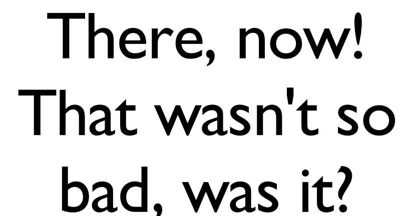
- Simplest of all the Perl5 CGI interface modules... a real nice way to get your feet wet.
- Provides support for reading tags from GET/POST, and handling file upload.
- Like all the other CGI modules, CGI_Lite is an *interface class*... you create an instance, and send it messages to "talk" to the Common Gateway Interface.

Copyright © 1996, 2000 Zeegee Software Inc.

CGI scripting / Perl CGI modules

# Using CGI_Lite

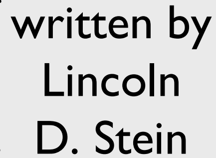
```
#!/usr/local/bin/perl -Tw
use CGI_Lite;
use strict;

# Create the CGI interface object:
my $cgi = new CGI_Lite;

# Parse the form data:
$cgi->parse_form_data;

# Output the HTTP header:
print "Content-type: text/plain\r\n";
print "\r\n";

# Output the document:
print "Here's the data we got back...\n";
$cgi->print_form_data;
print "BYE!\n";
```




Copyright © 1996, 2000 Zeegee Software Inc.

# More CGI_Lite

```
#!/usr/local/bin/perl -Tw
use CGI_Lite;
use strict;

# Create the CGI interface object, and get the query:
my $cgi = new CGI_Lite;
my %query = $cgi->parse_form_data;

# Output the HTTP header:
print "Content-type: text/plain\r\n\r\n";

# Output the document:
print "Here's the data we got back...\n";
my $key;
foreach $key (keys %data) {
   print $key, " = ", $query{$key}, "\n";
}
print "BYE!\n";
```




CGI scripting / Perl CGI modules

# CGI.pm

- Slightly more complex, but more thorough, CGI interface module.
- Kind of a "backpacker's interface": this one compact (well, not *so* compact) module has support for GET/POST, file upload, HTML form generation, HTTP cookies, saving queries to file, etc., etc.!
- Various components have been split off into the newer CGI:: modules, but this little baby's great for those one-shot, not-so-easy CGI scripting jobs...

CGI scripting / Perl CGI modules

# Using CGI.pm

```
#!/usr/local/bin/perl -Tw
use CGI;
use strict;

# Create the CGI interface object, and get the query:
my $cgi = new CGI;

# Output the HTTP header:
print "Content-type: text/plain\r\n\r\n";

# Output the document:
print "Here's the data we got back...\n";
my ($param, $value);
foreach $param ($cgi->param) {
   foreach $value ($cgi->param($param)) {
      print "$param = $value\n";
   }
}
```

# Debugging CGI.pm scripts

- One really nice thing about CGI.pm scripts is that you can run them from the Unix command line:

```
perl -d mycgi
key1="val1... this can be quoted"
key2=val2
^D
```

- or -

```
perl -d mycgi key1="val1" key2=val2
```

- This can make it a lot easier to see "what's going on" as you start working with CGI scripts.

Copyright © 1996, 2000 Zeegee Software Inc.

# The CGI:: family

- The heavy-duty CGI interface for the serious developer. A whole family of related classes.
- Very large... can take a while to load on slower machines.
- Important modules:

| | |
|---|---|
| **CGI::Base** | interface to the **CGI environment** |
| **CGI::Request** | for accessing the **query parameters** |
| **CGI::Carp** | redefine error handling |

# CGI::Base

```
#!/usr/local/bin/perl -Tw
use CGI::Base;
use strict;

# Create the CGI interface object:
my $cgi = new CGI::Base;

# Get some CGI variables... no need to use %ENV!
my $agent = $cgi->var('HTTP_USER_AGENT');

# Set up logging to a file (STDERR goes there as well):
$cgi->open_log("/home/mylog");
$cgi->log("Hello, nurse!");

# Handy debugging:
print "Content-type: text/html\r\n\r\n";
print $cgi->as_string;
```

Copyright © 1996, 2000 Zeegee Software Inc.

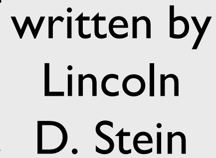
# CGI::Request

```
#!/usr/local/bin/perl -Tw
use CGI::Request;

# Create the interface object (contains a CGI::Base):
my $req = new CGI::Request;

# Get some CGI variables... (note use of CGI object!)
my $agent = $req->cgi->var('HTTP_USER_AGENT');

# Output the header and document:
print "Content-type: text/plain\r\n\r\n";
print "Here's the data we got back...\n";
my ($param, $value);
foreach $param ($req->params) {
   foreach $value ($req->param($param)) {
      print "$param = $value\n";
   }
}
```

Copyright © 1996, 2000 Zeegee Software Inc.

# More CGI::Request

```
#!/usr/local/bin/perl -Tw
use CGI::Request;
use strict;

# Create the interface object (contains a CGI::Base):
my $req = new CGI::Request;

# Add some values to it:
$req->extract_values("msg=Hello%2C+nurse%21");

# Handy debugging to see what we've got:
print "Content-type: text/html\r\n\r\n";
print $req->as_string;
```






# CGI::Carp

- By default, Perl's warnings/errors go to stderr... which goes to the HTTPD error log... which gets polluted with error message that say nothing about who logged them, or when:

```
I'm confused at test.pl line 3.
Error: Permission denied.
I'm dying.
```

- When your CGI script uses this module, the normal error-output routines are redefined to put timestamps and program names in the message:

```
[Fri Nov 17 21:40:43 1995] test.pl: I'm confused at test.pl line 3.

[Fri Nov 17 21:40:43 1995] test.pl: Error: Permission denied.
[Fri Nov 17 21:40:43 1995] test.pl: I'm dying.
```

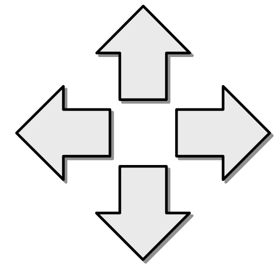
Copyright © 1996, 2000 Zeegee Software Inc.

# More CGI::Carp

- Unfortunately, CGI::Carp's redefinition of carp(), croak(), confess(), etc. can cause warning messages to be output to STDERR. This is annoying, but perfectly normal.
- The normal output to STDERR can be redirected to a file, if you don't care for the HTTPD error log. Do it in a BEGIN block to catch compilation errors as well:

```
use CGI::Carp qw(carpout);

BEGIN {
   use CGI::Carp qw(carpout);
   open(LOG, ">>/my/cgi-log") or die("open: $!");

   carpout(LOG);
}
```

# FAQs



If you've gotten this far, chances are, something has gone wrong at some point... or maybe you just want to do something a little more advanced, but don't see how.

- Troubleshooting
- HOW-TOs
- Security-related questions

Copyright © 1996, 2000 Zeegee Software Inc.

# Why am I getting "500" errors?

- The script does not contain the "#!/usr/bin/perl" shabang line that points to the Perl interpreter, or the path to the interpreter is invalid.
- Your script has compile-time errors (syntax error, violating "strict" rules, "use"ing a library file that is not in the @INC path). Always run scripts from the command-line first!
- The first line output from the script is not a valid HTTP header (like "Content-type: text/html"). Watch out for buffering effects, as in:

```
print "Content-type: text/plain\r\n\r\n";
`ls -l /usr/local/bin`;
```

- You forgot the blank line after the HTTP header data.

# I can't seem to write a file

- Generally, the HTTP server will be running as user "nobody", or "www", or some other user id with minimal privileges. So the directory where you intend to create the file *must* be writeable by *this* user id.
- In general, you should *always* check the return status from the open command to see if it was a success:

```
open(FILE, "/some/file.dat") or die "open: $!";
```

(I'll teach you how to die gracefully soon...)

Copyright © 1996, 2000 Zeegee Software Inc.

# My output is missing/out of order!

Stuff like this:

```
print "Content-type: text/plain\r\n\r\n";
`ls -l /usr/local/bin`;
```

Breaks easily, due to the fact that STDOUT is buffered. You can turn buffering off by using the $| variable at the start:

```
$| = 1;
```

But this makes output inefficient. You can also try this:

```
use FileHandle;
print "Content-type: text/plain\r\n\r\n";
STDOUT->flush;
`ls -l /usr/local/bin`;
```

# It's not running my subprogram!

Stuff like this:

```
@output = `myprog -a foo.dat`;
```

Is vulnerable to the shell's environment. Often, HTTPDs set up a minimal PATH before invoking the CGIs...

```
PATH = /bin:/usr/bin
```

The solution is to either muck with the PATH:

```
$ENV{PATH} .= ":/usr/local/bin:/usr/ucb";
```

Or use absolute paths to executables:

```
@output = `/usr/local/bin/myprog -a foo.dat`;
```

Copyright © 1996, 2000 Zeegee Software Inc.

# It's not finding my file!

Same problem as above:

```
@out = `/c/spot -rUN spot.run`;
```

Depends on the current working directory. Don't do that. Use absolute paths:

```
@out = `/c/spot -rUN /run/spot.run`;
```

Copyright © 1996, 2000 Zeegee Software Inc.

# How do I redirect to another page?

Just output a *Location* HTTP header, instead of the *Content-type* header:

```
print "Location: $otherURL\r\n\r\n";
exit 0;
```

# How do I handle different browsers?

Use the HTTP_USER_AGENT environment variable:

```
$browser = $ENV{'HTTP_USER_AGENT'};
if ($browser =~ /Mozilla/) {
    # Netscape code...
}
else {
    # Non-Netscape code...
}
```

You're better off doing all this at the start to figure out what *features* the user agent has... then set global flags like $Agent{HAS_TABLES} = 1.

CGI scripting / FAQs

# How do I mail the form data?

```
$cgif = new CGI::Form;          # a Perl CGI module

$from    = $cgif->param('from');      # sender e-address
$name    = $cgif->param('name');      # sender name
$subject = $cgif->param('subject');   # subject
$message = $cgif->param('message');   # message body

open SENDMAIL, "|/usr/bin/sendmail -t -i" or die "$!";
print SENDMAIL <<EOF;
From: $from <$name>
To: me\@myhost.com
Reply-To: $from
Subject: $subject

$message
EOF
close SENDMAIL;
die "sendmail failed" if ($? >> 8);
```

Here's a real simple script for doing that. Make sure to edit the path to sendmail for your system, as well as the destination!

Copyright © 1996, 2000 Zeegee Software Inc.

# How can I tell who accessed me?

| | |
|---|---|
| `HTTP_FROM` | Theoretically set to the email address of the user. Many browsers do not set it at all, and ones that do allow the user to set it to anything. Unreliable. |
| `REMOTE_USER` | Only set if secure authentication was used to access the script. |
| `REMOTE_IDENT` | Only set if server has contacted an IDENTD server on the client machine... a slow, rare, and unreliable operation. |
| `REMOTE_HOST` | Will not identify the user specifically, but does provide information about the site the user has connected from, if the hostname was retrieved by the server. |
| `REMOTE_ADDR` | The dotted-decimal IP address of the client machine. |

Copyright © 1996, 2000 Zeegee Software Inc.

# How secure are "password" fields?

**They're not!** The forms interface allows you to have a "password" field, but it should not be used for anything highly confidential.

All form data (including "password" fields) gets sent from the browser to the server as *plain text*... *not* as encrypted data.

If you want to solicit secure information, you need to use a secure HTTP server.

# Is this really a security hole?

Consider the following bit of code, which grabs a user's form field and uses it as a search pattern:

```
$pattern = $req->param('pattern');
@ans = `grep $pattern some.file`;
```

Harmless, right? Wrong. Consider what that shell line would be if the user entered into the HTML form the pattern:

```
; cat /etc/passwd ;
```

Are you scared yet? No? How about:

```
; rm -fr / ;
```

Hmm?

# Why use -T?

Fortunately, Perl prevents you from hurting yourself if you just turn on taintchecking (-T).

Perl would then detect that `$pattern` is ***tainted*** (it originates from outside the code) and would halt if...

- You attempt to pass `$pattern` into a shell.
- You attempt to use `$pattern` in the naming of a file.

Perl CGI scripts with *taintchecking* and *strict* are much safer than their sh/C counterparts.

# So how do I run programs securely?

*Avoid situations where a shell is involved.* Shells interpret metacharacters like ; and |, and unless you know exactly what you're doing, untainting strange input just because you've "quoted" it can be dangerous.

Code that "grep" example like this:

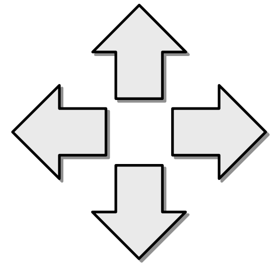
```
if (open(GREP, "-|")) {    # open forks; we're the parent
    @ans = <GREP>;
}
else {                     # open forked; we're the child
    exec("/usr/local/bin/grep", $user_field, "some.file")
        || die "Error exec'ing command", "\n";
}
close(GREP);
```

CGI scripting

# *PROJECT*

- Write a Perl CGI script which serves as a gateway to the *finger* program (or, if you like, the *man* program). If invoked with no parameters, it should serve a simple HTML form; with parameters, it should invoke *finger* (*man*) appropriately.
- Start off with just plain text output. When you get it to work, use Perl's text-processing power to create very pretty HTML output instead.

# TRICKS OF THE TRADE



There are a few of issues that all CGI developers must face to develop solid code.

Here are some good tips and strategies.

- Page buffering
- Trace logging

Copyright © 1996, 2000 Zeegee Software Inc.

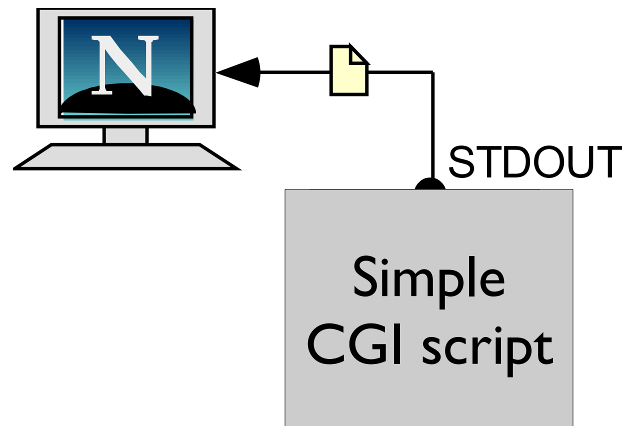
# Tired of seeing this, yet?

## 500 Server Error

The server encountered an internal error or misconfiguration.  Please alert the server administrator at this host.

# Page-buffering

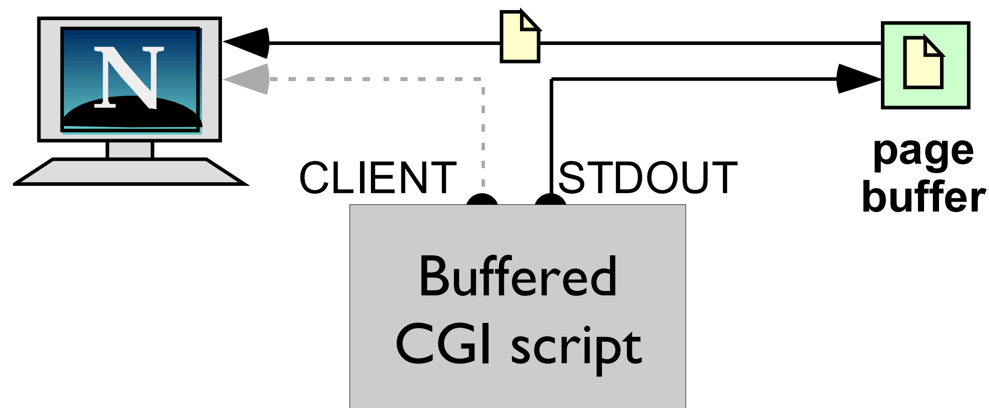
- Output to STDOUT often goes directly to the client's browser... but what if you've output half a page of HTML, and an error occurs? Can't retract output!
- One solution I came up with is *page-buffering*...




Copyright © 1996, 2000 Zeegee Software Inc. 

# Page-buffering

1. Duplicate STDOUT into a new filehandle, CLIENT.
2. Redirect STDOUT to a *tmpfile()* temp file... the ***page buffer***.
3. **If script finishes normally**, rewind page buffer, write contents to CLIENT, and exit.
4. **If an error occurs**, discard page buffer, write error message to CLIENT (I call it the script's *dying gasp*), and exit.




Copyright © 1996, 2000 Zeegee Software Inc.

# Using STDERR

- As noted, STDERR normally points to the HTTPD error log, and can be used for general logging:

```
sub debug { print STDERR "$0 debug: ", @_, "\n" }
debug "Look for me in the error log!";
```

- One (sloppy) way to catch errors is to "dup" STDERR to STDOUT early on in your script (after outputting the valid HTTP headers, of course!):

```
open(STDERR, ">&STDOUT");
```

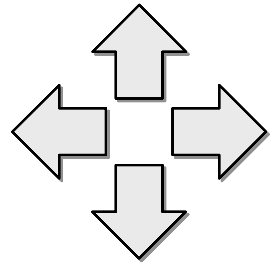
This redirects all of the error messages to STDOUT (the client). But it's **messy**, and requires that you know early on what HTTP header you want to output!

# Trace logs

- Nice to be able to generate temporary debugging info during execution of a script... a *trace log*:
  - If script exits okay, discard the trace log.
  - If script dies, output the trace log in the "dying gasp".
- Use **same basic scheme as page-buffering**:
  - Open log at start, using *tmpfile()*. Maybe even redirect STDERR to it (though you can lose warnings this way).
  - Write entries to it.
  - If script dies, rewind trace log, and output.

Copyright © 1996, 2000 Zeegee Software Inc.

# WWW CLIENTING



Just as many of us need systems to provide information, some of us need systems to download it.

- URI modules
- LWP modules

Copyright © 1996, 2000 Zeegee Software Inc.

# URI::Escape

Encode/decode query strings. Useful for generating URLs to CGI scripts.

```
use URI::Escape;

$query = "10% is enough\n";
$safe = uri_escape($query);
$verysafe = uri_escape($query, "\0-\377");
print "http://somehost/script?$safe";

$str  = uri_unescape($safe);
```

## WWW Clienting

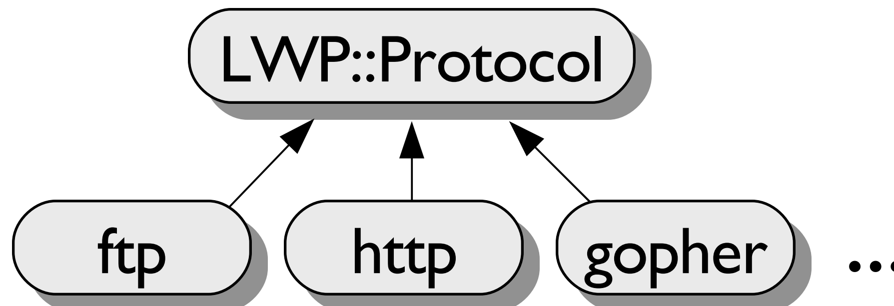
# URI::URL

Objects representing URLs of various flavors. Nice if you hate parsing URLs with regexps...

```
use URI::URL;

$raw = 'http://enterprise.sf.ufp:1701/~picard';
$url = new URI::URL $raw;  # or, url($raw)

$scheme = $url->scheme;    # http
$host   = $url->host;      # enterprise.sf.ufp
$port   = $url->port;      # 1701
$path   = $url->path;      # /~picard

$url->host('ncc1701d.sf.ufp');
print $url->as_string;     # http://ncc1701d...
```

# LWP

- The Library for Web Programming in Perl.
- Collection of many different modules, which can be used to build Perl "User Agents" (spiders, robots, etc.)
- Design allows support for new protocols to be added as simple "plug-ins"... just write an LWP::Protocol::*xyz* module to spec.




Copyright © 1996, 2000 Zeegee Software Inc.

# LWP::Simple

Very simple interface to the LWP client modules. Good for when you "just want to download a file by its URL":

```
use LWP::Simple;

# Download data to a scalar (careful of big files!!):
$data = get "http://voyager.sf.ufp/emh/doctor.tar";

# Download data into a file (much better):
$code = getstore("http://voyager.sf.ufp/emh/doc.tar",
                 "the-doctor");
```

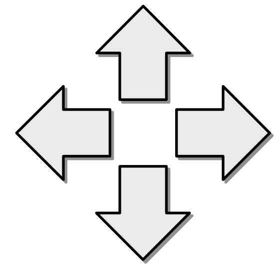
Copyright © 1996, 2000 Zeegee Software Inc.

# LWP::UserAgent

Object-oriented interface to the LWP client modules.

Good for complex client activities.

```
use LWP::UserAgent;

# Create a user agent:
$ua = new LWP::UserAgent;

# Create a request object:
my $request = new HTTP::Request 'GET', $url;

# Submit the request, grabbing a response object:
my $response = $ua->request($request);

# Extract and print the content data:
print $response->content;
```



# MAIL AND MIME

Often, one needs to write a program that is able to send mail... maybe even read it.

Newer email developments (like MIME) have made email processing more rewarding... and challenging.

- The Mail:: modules
- The MIME:: modules

# *PROJECT*

- Design a class, Mail::Msg, whose instances are outgoing mail messages. You should be able to:
  - Create a new instance.
  - Set up the destination(s), subject line, return address, etc.
  - Dump the contents of a file to it.
  - Sign it with your signature file.
  - Send it.
- You may assume Unix.

# Mail-Tools

- Tool kit providing almost all you need to be able to...
  - Send email
  - Parse email headers (good for mail servers)
  - Manage .mailcap files
- Basically, the Mail:: modules...

# Mail::Send

Object-oriented way of constructing outgoing email:

```
use Mail::Send;

# Create message object:
$msg = new Mail::Send;

# Set header fields:
$msg->to('sisko@ds9.sf.ufp');
$msg->cc('worf@ds9.sf.ufp');
$msg->subject('Still on for Poker Thursday?');
$msg->set('From', 'god@universe.infi.net');  # fake!

# Print the message:
$fh = $msg->open;
print $fh "Remember: Worf's turn to buy beer!";
$fh->close;         # complete message and send it
```

# Setting Mail::Send's mailer

Uses Mail::Mailer to do the actual sending. This example uses *sendmail* instead of *mail*:

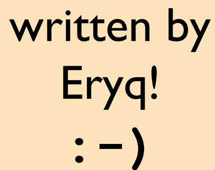
```
# Set up the path to sendmail:
$Mail::Mailer::Mailers{'sendmail'} =
    '/usr/sbin/sendmail';

# Print the message:
$fh = $msg->open("sendmail");
print $fh "Remember: Worf's turn to buy beer!";
$fh->close;         # complete message and send it
```

Interesting note: that filehandle *$fh* is actually a Mail::Mailer::sendmail object.

# Mail::Header

For building or parsing email headers. Useful if you're developing an email gateway, or sending mail the hard way...

```
# Create a new header from input:
$head = new Mail::Header \*STDIN;

# Extract information:
my $sender = $head->get('sender',0);
my $subject = $head->get('subject', 0);
my @history = $head->get('received');

# Modify and output:
$head->replace('to', $newuser);
$head->print(\*STDOUT);
```



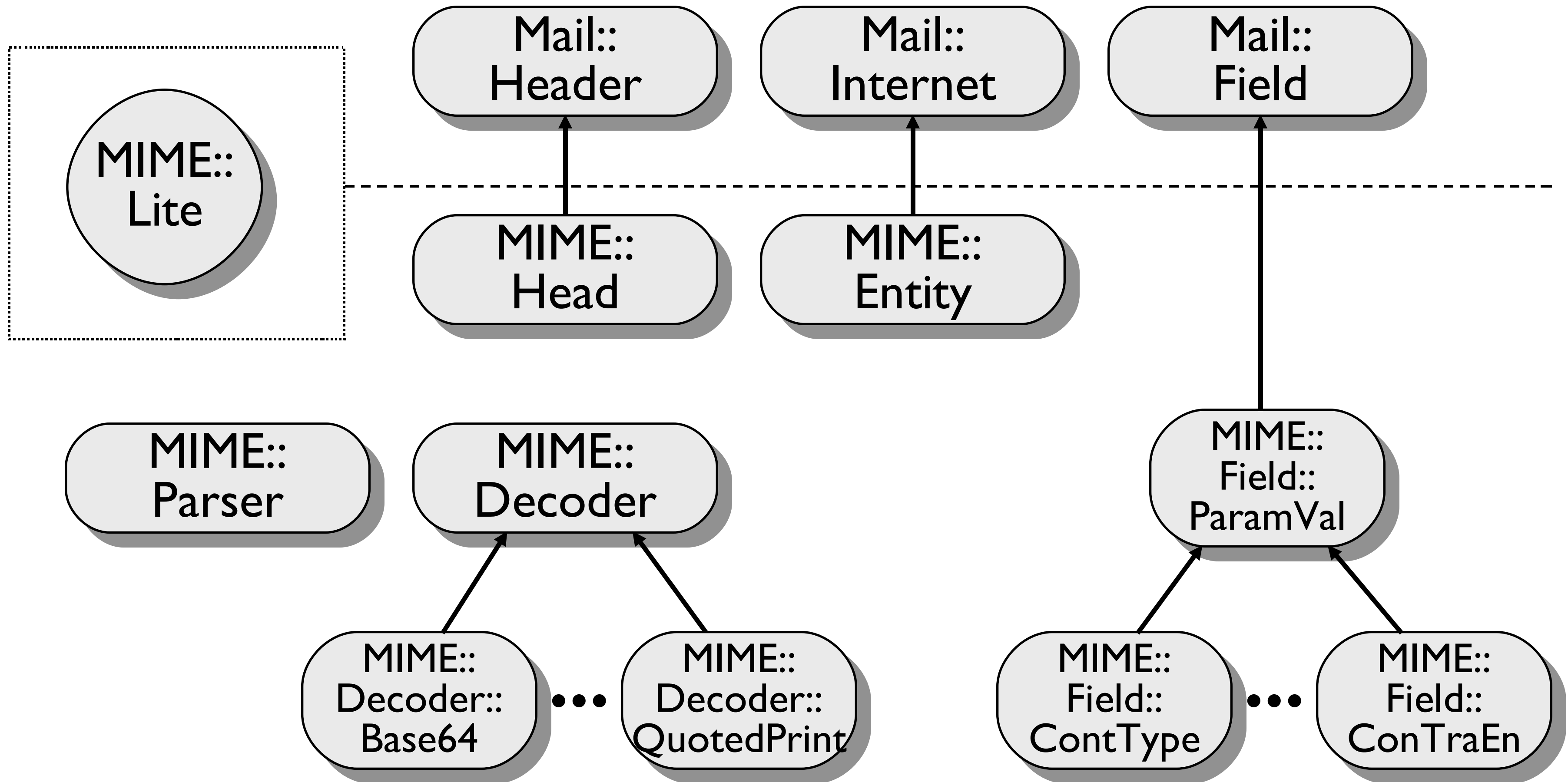



# MIME-Tools

- Tool kit providing almost all you need to be able to...
  - Parse, decode, and save single- or multipart MIME messages.
  - Construct and output single- or multipart MIME messages (yes, with binary attachments).
- Newest release (2.0) the result of collaboration with Mail-Tools author on new Mail-Tools modules (1.06) that were more amenable to subclassing.

Copyright © 1996, 2000 Zeegee Software Inc.

# The Mail/MIME Hierarchy




Copyright © 1996, 2000 Zeegee Software Inc.

# Parsing MIME messages

The MIME::Parser module can be used out of the box to parse a message. Parts can go to file or stay in core:

```
use MIME::Parser;

# Create parser, and set the output directory:
my $parser = new MIME::Parser;
$parser->output_dir("$ENV{HOME}/mimemail");

# Parse input, creating a MIME entity:
$entity = $parser->parse(\*STDIN) || die "no MIME";

# Take a look at the top-level entity:
$entity->dump_skeleton;
```

Copyright © 1996, 2000 Zeegee Software Inc.

# Other MIME components

- The MIME::Base64 and MIME::QuotedPrint modules (by Gisle Aas) do the encoding/decoding work of this toolkit, and can be used separately.
- The MIME::Decoder class may be used to encode/decode streams:

```
# A filter to encode in base64:
use MIME::Decoder;
$decoder = new MIME::Decoder 'base64'
       or die "unsupported";
$decoder->encode(\*STDIN, \*STDOUT);
```

Copyright © 1996, 2000 Zeegee Software Inc.

# Creating MIME messages

```
# Create the top-level, and set up the mail headers:
$top = build MIME::Entity Type=>"multipart/mixed";
$top->head->add('from',    "me\@myhost.com");
$top->head->add('to',      "you\@yourhost.com");
$top->head->add('subject', "Hello, nurse!");

# Attachment #1: a simple text document:
$top->attach(Path=>"./testin/short.txt");

# Attachment #2: a GIF file:
$top->attach(Path     => "./docs/mime-sm.gif",
             Type     => "image/gif",
             Encoding => "base64");

# Attachment #3: some literal text:
$top->attach(Data=>$message);

# Send it:
open MAIL, "| /usr/lib/sendmail -t -oi -oem" or die "open: $!";
$top->print(\*MAIL);
close MAIL
```

# *FINAL PROJECT*

Go to the Perl5 Module List and scan down it, keeping in mind the projects you are currently working on.

Somewhere in that list is a hole... a module that you feel should be there, but isn't. A module you could really use. One which many of your colleagues could use, as well.

> Propose that module to comp.lang.perl.modules.
>
> Then write it.
>
> Then release it.

Help save the world.