CGI Programming HTML Forms

Doug Treder
UW Extension

CGI and Forms

- HTML Forms overview
- Redirecting to another URL
- Form Processing

CGI.pm

- Remember,

```
perldoc CGI
```

- is your best friend
- Can use OO or procedural interface to CGI.pm
  - the examples in this set of slides use the procedural interface

CGI Basics

- Three ways to print HTML
  - ordered arguments (need docs)
  - name value pairs (names all start with -)
  - enclosing tag and attribute (first arg is hash reference)

CGI Basics

- An easy way of printing standard HTML
- Easy parsing of incoming form data
- Easy prepopulation of forms for pipelines and reflective forms

In The Beginning...

- Using HTML forms.
- First, declare a form begin and end:

```
<FORM ACTION="path/to/script" METHOD="method">
<!-- form attributes -->
</FORM>
```

Forms and CGI.pm

- Same thing, using CGI.pm:

```
use CGI qw(:standard);

print startform (-action => "path/to/script"
                 -method => "method");

print "<!-- form attributes -->";

print endform;
```

Self Referential Forms

- NOTE: If you pass $query->startform no parameters, it will default to calling itself (ie: call this same script again). This will prove to be very useful in some situations to help preserve state. More on this later...

```
# call this script again
print startform;
print "<!-- form attributes -->";
print endform;
```

Submit and Reset

- Then throw in a submit button and/or reset button. (Note: VALUE field is optional). SUBMIT submits the form, RESET returns it to defaults.

```
<FORM ACTION="path/to/script" METHOD="method">
<!-- form attributes -->
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
```

Submit and Reset

- Using CGI.PM:

```
use CGI qw(:standard);

print startform (-action => "path/to/script"
                 -method => "method");

print "<!-- form attributes -->";

print submit (-value => "Submit");
print reset (-value => "Reset");
print endform;
```

Input Text

- Used for a single line of text. Can control visible size of box with SIZE and maximum string length with MAXLENGTH.

```
<input type=text>
```

Input in HTML

- In HTML:

```
<INPUT TYPE="text" NAME="somename"
 VALUE="somevalue" SIZE="integer"
 MAXLENGTH="integer">

<INPUT TYPE="text" NAME="age"
 VALUE="18" SIZE="5" MAXLENGTH="5">
```

Input with CGI.pm

- Using CGI.pm:

```
print textfield (-name      => "age",
                 -default   => "18",
                 -size      => "5",
                 -maxlength => "5");
```

Reading params

- CGI parses $ENV{QUERY_STRING} and can return entire list or just one at a time

```
use CGI qw(:standard);

my $age = param(‘age’);

# load them all at once into a hash
my %all = param();
$age = $all{age};
```

Hidden Fields

- Won't be displayed on screen (but will show up under View Source). For preserving state...

```
# In HTML:
<INPUT TYPE="HIDDEN" NAME="somename"
VALUE="somevalue">

#Using CGI.pm:
print hidden (-name => "savedstate",
              -value => "homestar");
```

Password Box

- Like TEXT, but echoes * character to hide text from prying eyes. (note: not necessarily secure!)

```
<!-- In HTML: -->
<INPUT TYPE="PASSWORD" NAME="pinnumber" SIZE="15"
MAXLENGTH="20">



#Using CGI.pm:
print password_field (-name => "pinnumber",
                      -size => "15",
                      -maxlength => "20");
```

Checkbox

- Can either be used for simple booleans or attributes that can have several values at the same time.

```
<!-- In HTML: -->
<INPUT TYPE="CHECKBOX" NAME="wantsbeer” VALUE="some"
CHECKED> Have a beer?


# Using CGI.pm:
print checkbox (-name    => "wantsbeer",
                -value   => "some",
                -checked => "CHECKED",
                -label   => "Have a beer?");
```

Checkbox Group

- Creates groups of checkboxes that are related by the same name.

```
my @group_values = ("vitamin", "deworm", "wash");
my %group_labels = ("vitamin" => "Vitamin Shot",
                    "deworm"  => "Deworm",
                    "wash"    => "Deluxe Wash");

print checkbox_group (-name    => "service_type",
                      -values  => \@group_values,
                      -labels  => \%group_labels,
                      -default => ["wash"],
                      -rows    => 2,
```

Checkbox Group

- -name and -values are the checkbox name and values.
  - -values has to be an array reference.
- -labels is a hash reference that relates the checkbox values to the user-visible labels printed next to them.
- If -linebreak is true, line breaks are added
- -default is a reference to an array listing the checkboxes that should be checked by default.

Radio Buttons

- Attributes that can take a single value from a set of alternatives. Each radio button has the same NAME.

```
<!-- In HTML: -->

<INPUT TYPE="RADIO" NAME="payment" VALUE="cash"
CHECKED> Cold, Hard Cash<BR>
<INPUT TYPE="RADIO" NAME="payment" VALUE="check">
Check<BR>
<INPUT TYPE="RADIO" NAME="payment" VALUE="iou"> I.O.U.
```

Radio Buttons

- Similar to checkbox_group.   -default must be scalar since only one can be selected at one time.

```
my @values = ('cash', 'check', 'iou');
my %group_labels = ("cash"  => "Cold, Hard Cash",
                    "check" => "Check",
                    "iou"   => "I.O.U.");

print radio_group (-name    => "payment",
                   -values  => \@values,
                   -labels  => \%group_labels,
                   -default => "cash",
                   -rows => 3, -columns => 2);
```

Dropdown Menu

- Also known as popup or scrolling menu.

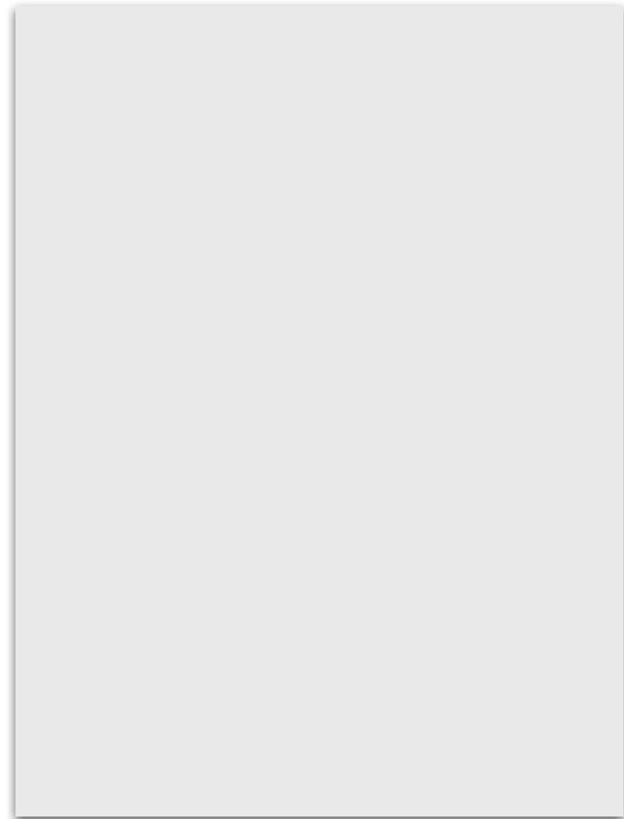
```
<!-- In HTML: -->
<SELECT NAME="payment">

<OPTION VALUE="cash" SELECTED>Cold, Hard Cash
<OPTION VALUE="check">Check
<OPTION VALUE="iou">I.O.U.

</SELECT>
```

Break

- After this break, we just have
    - some other kinds of form fields
    - Page redirects
    - Form validation



Dropdown Menu

- Similar to checkbox_group and radio_group

```
my @values = ('cash', 'check', 'iou');
my %group_labels = ("cash"  => "Cold, Hard Cash",
                    "check" => "Check",
                    "iou"   => "I.O.U.");

print popup_menu (-name    => "payment",
                  -values  => \@values,
                  -labels  => \%group_labels,
                  -default => "cash",
                  -rows    => 3,
                  -columns => 2);
```

MultiSelect

- Selecting multiple items from a scrolling list.

```
<!-- In HTML, just add MULTIPLE to SELECT (and
optional SIZE):  -->

<SELECT MULTIPLE NAME="payment" SIZE="3">
<OPTION VALUE="cash" SELECTED>Cold, Hard Cash
<OPTION VALUE="check">Check
<OPTION VALUE="iou">I.O.U.
</SELECT>
```

MultiSelect

- XHTML requires all attributes to have values.

```
<!-- In HTML, just add MULTIPLE to SELECT (and
optional SIZE):  -->

<SELECT MULTIPLE=”multiple” NAME="payment" SIZE="3">
<OPTION VALUE="cash" SELECTED>Cold, Hard Cash
<OPTION VALUE="check">Check
<OPTION VALUE="iou">I.O.U.
</SELECT>
```

MultiSelect

- Using CGI.pm, it's called scrolling_list:

```
my @values = ('cash', 'check', 'iou');
my %group_labels = ("cash"  => "Cold, Hard Cash",
                   "check" => "Check",
                   "iou"   => "I.O.U.");

print scrolling_list (-name     => "payment",
                      -values   => \@values,
                      -labels   => \%group_labels,
                      -default  => ["cash"],
                      -multiple => "true",
                      -size     => "3" );
```

Textarea

- Used to input multiple lines of text.

```
<!-- In HTML:  -->
<TEXTAREA NAME="comments" ROWS="4" COLUMNS="25">
Your default text goes here</TEXTAREA>

# Using CGI.pm:

print textarea (-name => "comments",
           -default => "Your default text goes here",
           -rows    => "4",
           -columns => "25" );
```

Sticky

- CGI has built in stickyness
- This means that
  - if a CGI prints form fields with the same name as fields it received,
  - the values in the script are replaced with what came to the script

CGI Sticky

- Especially annoying for “action” or “dispatch” fields
-

```
use CGI;

my $q = CGI->new();

print $q->startform();



print $q->hidden(-name => “action”,
                 -value => “choose”);
```

CGI Sticky

- Especially annoying for “action” or “dispatch” fields
- Override this with

```
use CGI;

my $q = CGI->new();

print $q->startform();

$q->param( action => “choose” );

print $q->hidden(-name => “action”,
                 -value => “choose”);
```

Page Redirects

- HTTP Redirects can't be done in HTML

```
use strict;
my $url = "http://www.perl.com/CPAN/";
# do whatever processing you need to do here
print "Location: $url\n\n"; # HTTP redirect
exit;
```

Page Redirects

```
use CGI;
use strict;
my $url = "http://www.yahoo.com";
# do processing you need to do here
print redirect (-URL => $url);
exit;
```

Validating Form Data

- numeric fields
- date fields
- range fields
- credit card number fields
- fields that must be consistent with other fields

Validation

- One trick is to have a check_errors subroutine where you return any errors -- if there are any, print them out.

```
use strict;
use CGI;

# return a list of errors
my @errors = check_errors();
print_errors (@errors) if @errors;
```

Validation

- Have to roll your own validation routines. Examples:

```
if ( $input =~ /[\D-\.]/ ) {
   push @errors, 'Contains non-digits';
}
if ( length ($input) != 8 ) {
   push @errors, 'Invalid length';
}
if ($input !~ /^[\w\.]+\@\w+\.\w+/) {
   push @errors, 'Bad email address';
   # NOTE, many valid email addresses will fail in
   # this regex!!
}
```