# Introduction to Web Programming

## Lecture 11: Uploading Files

## Including files: `include`

How can we avoid redundantly repeating this content or code?

```
include("filename");
```

```
include("header.html");       # repeated HTML content
include("shared-code.php");   # repeated PHP code
```

- inserts the entire contents of the given file into the PHP script's output page
- encourages modularity
- useful for defining reused functions needed by multiple pages
- related: `include_once`, `require`, `require_once`

## Including a common HTML file

```
<!DOCTYPE html>
<!-- this is top.html -->
<html><head><title>This is some common code</title>
...
```

```
include("top.html");      # this PHP file re-uses top.html's HTML content
```

- including a .html file injects that HTML output into your PHP page at that point
- useful if you have shared regions of pure HTML tags that don't contain any PHP content

## Including a common PHP file

```
<?php
# this is common.php
function useful($x) { return $x * $x; }

function top() {
  ?>
  <!DOCTYPE html>
  <html><head><title>This is some common code</title>
  ...
  <?php
}
```

```
include("common.php");   # this PHP file re-uses common.php's PHP code
$y = useful(42);         # call a shared function
top();                   # produce HTML output
...
```

- including a .php file injects that PHP code into your PHP file at that point
- if the included PHP file contains functions, you can call them
- if you have redundancy of both PHP and HTML content, put redundant HTML into functions

## A form that submits to itself

```
<form action="" method="post">
  ...
</form>
```

- a form can submit its data back to itself by setting the `action` to be blank (or to the page's own URL)
- benefits
  - fewer pages/files (don't need a separate file for the code to process the form data)
  - can more easily re-display the form if there are any errors

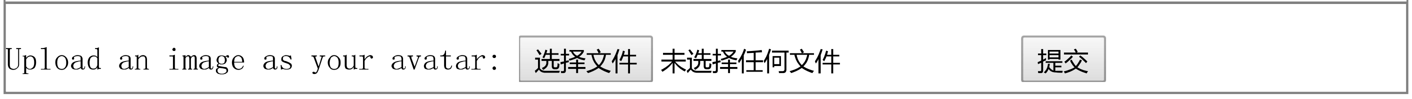
## Processing a self-submitted form

```
if ($_SERVER["REQUEST_METHOD"] == "GET") {
  # normal GET request; display self-submitting form
  ?>
  <form action="" method="post">...</form>
  <?php
} elseif ($_SERVER["REQUEST_METHOD"] == "POST") {
  # POST request; user is submitting form back to here; process it
  $var1 = $_POST["param1"];
  ...
}
```

- a page with a self-submitting form can process both GET and POST requests
- look at the global $_SERVER array to see which request you're handling
- handle a GET by showing the form; handle a POST by processing the submitted form data

## Uploading files

```
<form action="http://www.polytech.unice.fr/~gaetano/params.php"
      method="post" enctype="multipart/form-data">
  Upload an image as your avatar:
  <input type="file" name="avatar" />
  <input type="submit" />
</form>
```


Upload an image as your avatar:

- add a file upload to your form as an `input` tag with `type` of `file`
- must also set the `enctype` attribute of the form

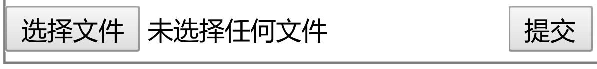
- it makes sense that the form's request method must be `post` (an entire file can't be put into a URL!)
- form's `enctype` (data encoding type) must be set to `multipart/form-data` or else the file will not arrive at the server

## Processing an uploaded file in PHP

- uploaded files are placed into global array `$_FILES`, not `$_POST`
- each element of `$_FILES` is itself an associative array, containing:
  - `name` : the local filename that the user uploaded
  - `type` : the MIME type of data that was uploaded, such as `image/jpeg`
  - `size` : file's size in bytes
  - `tmp_name` : a filename where PHP has temporarily saved the uploaded file
    - to permanently store the file, move it from this location into some other file

## Uploading details

```
<input type="file" name="avatar" />
```



- example: if you upload `borat.jpg` as a parameter named `avatar`,
  - `$_FILES["avatar"]["name"]` will be `"borat.jpg"`
  - `$_FILES["avatar"]["type"]` will be `"image/jpeg"`
  - `$_FILES["avatar"]["tmp_name"]` will be something like `"/var/tmp/phpZtR4TI"`

## Processing uploaded file, example

```
$username = $_POST["username"];
if (is_uploaded_file($_FILES["avatar"]["tmp_name"])) {
  move_uploaded_file($_FILES["avatar"]["tmp_name"], "$username/avatar.jpg");
  print "Saved uploaded file as $username/avatar.jpg\n";
} else {
  print "Error: required file not uploaded";
}
```

- functions for dealing with uploaded files:
  - `is_uploaded_file(`*filename*`)`
    returns TRUE if the given filename was uploaded by the user
  - `move_uploaded_file(`*from*`, `*to*`)`
    moves from a temporary file location to a more permanent file
- proper idiom: check `is_uploaded_file`, then do `move_uploaded_file`

## More about associative arrays

## Creating an associative array

```
$name = array();
$name["key"] = value;
...
$name["key"] = value;
```

```
$name = array(key => value, ..., key => value);
```

```
$blackbook = array("marc"   => "206-685-2181",
                   "stuart" => "206-685-9138",
                   "jenny"  => "206-867-5309");
```

- can be declared either initially empty, or with a set of predeclared key/value pairs

## Printing an associative array

```
print_r($blackbook);
```

```
Array
(
    [jenny] => 206-867-5309
    [stuart] => 206-685-9138
    [marc] => 206-685-2181
)
```

- `print_r` function displays all keys/values in the array
- `var_dump` function is much like `print_r` but prints more info
- unlike `print`, these functions require parentheses

## Associative array functions

```
if (isset($blackbook["marc"])) {
  print "Marc's phone number is {$blackbook['marc']}\n";
} else {
  print "No phone number found for Marc.\n";
}
```

| name(s) | category |
|---|---|
| `isset`, `array_key_exists` | whether the array contains value for given key |
| `array_keys`, `array_values` | an array containing all keys or all values in the assoc.array |
| `asort`, `arsort` | sorts by value, in normal or reverse order |
| `ksort`, `krsort` | sorts by key, in normal or reverse order |

## `foreach` loop and associative arrays

```php
foreach ($blackbook as $key => $value) {
  print "$key's phone number is $value\n";
}
```

```
jenny's phone number is 206-867-5309
stuart's phone number is 206-685-9138
marc's phone number is 206-685-2181
```

- both the key and the value are given a variable name
- the elements will be processed in the order they were added to the array

## 15.1: Form Validation

## What is form validation?

- **validation**: ensuring that form's values are correct
- some types of validation:
    - preventing blank values (email address)
    - ensuring the type of values
        - integer, real number, currency, phone number, Social Security number, postal address, email address, date, credit card number, ...
    - ensuring the format and range of values (ZIP code must be a 5-digit integer)
    - ensuring that values fit together (user types email twice, and the two must match)

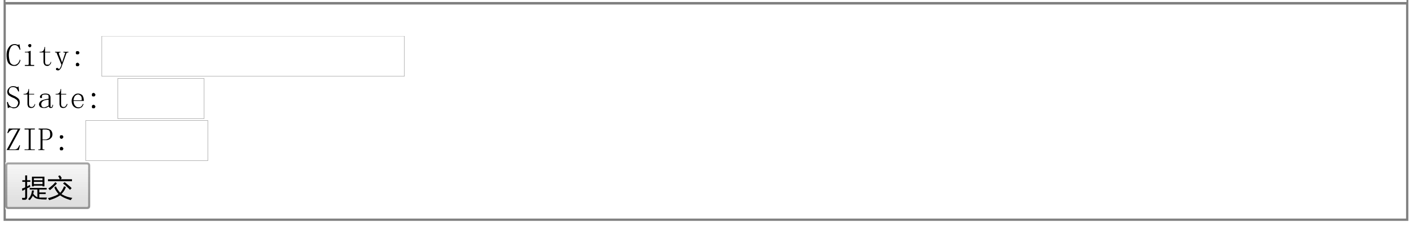
## Client vs. server-side validation

Validation can be performed:

- **client-side** (before the form is submitted)
    - can lead to a better user experience, but not secure (why not?)
- **server-side** (in PHP code, after the form is submitted)
    - needed for truly secure validation, but slower
- both
    - best mix of convenience and security, but requires most effort to program

## An example form to be validated

```
<form action="http://foo.com/foo.php" method="get">
  <div>
    City:  <input name="city" /> <br />
    State: <input name="state" size="2" maxlength="2" /> <br />
    ZIP:   <input name="zip" size="5" maxlength="5" /> <br />
    <input type="submit" />
  </div>
</form>
```


City:
State:
ZIP:
提交

- Let's validate this form's data on the server...

## Basic server-side validation code

```
$city  = $_POST["city"];
$state = $_POST["state"];
$zip   = $_POST["zip"];
if (!$city || strlen($state) != 2 || strlen($zip) != 5) {
  print "Error, invalid city/state/zip submitted.";
}
```

- *basic idea:* Examine parameter values, and if they are bad, show an error message and abort.
- What should we do if the data submitted is missing or invalid?
  - simply `printing` an error message is not a very graceful result

## The `die` function

```
die("error message text");
```

- PHP's `die` function prints a message and then completely stops code execution
- it is sometimes useful to have your page "die" on invalid input
- *problem:* poor user experience (a partial, invalid page is sent back)

## The `header` function

```
header("HTTP header text");    # in general
header("Location: url");      # for browser redirection
```

- PHP's `header` function can be used for several common HTTP messages
  - sending back HTTP error codes (404 not found, 403 forbidden, etc.)
  - redirecting from one page to another
  - indicating content types, languages, caching policies, server info, ...
- you can use a `Location` header to tell the browser to redirect itself to another page
  - useful to redirect if the user makes a validation error
  - **must** appear before *any* other HTML output generated by the script

## Using `header` to redirect between pages

```
header("Location: url");
```

```
$city  = $_POST["city"];
$state = $_POST["state"];
$zip   = $_POST["zip"];
if (!$city || strlen($state) != 2 || strlen($zip) != 5) {
  header("Location: start-page.php");   # invalid input; redirect
}
```

- *one problem:* User is redirected back to original form without any clear error message or understanding of why the redirect occurred. (We can improve this later.)

## Another problem: Users submitting HTML content

```
<h1>hack</h1>
```

- A user might submit information to a form that contains HTML syntax
- If we're not careful, this HTML will be inserted into our pages (why is this bad?)

## The `htmlspecialchars` function

| | |
|---|---|
| `htmlspecialchars` | returns an HTML-escaped version of a string |

- text from files / user input / query params might contain <, >, &, etc.
- we could manually write code to strip out these characters
- better idea: allow them, but **escape** them

```
$text = "<p>hi 2 u & me</p>";
$text = htmlspecialchars($text);   # "&lt;p&gt;hi 2 u &amp; me&lt;/p&gt;"
```