Wordpress naming conventions:

Class Names

When it comes to working with WordPress, you're not likely to encounter classes unless you're doing one of two things:

- Writing a custom library to work alongside a theme or application
- Writing an OOP-based plugin

If you're simply working on the theme, you're more likely to be working with a set of functions - we'll be talking about those momentarily.

But for those who are working with plugins or their own libraries, it's important to remember that classes should typically be nouns - they should represent the purpose that they encapsulate and they should ideally do one thing and do it well.

For example, if you have a class called `Local_File_Operations` then it may be responsible for reading and writing files. It shouldn't be responsible for reading and writing files as well as, say, retrieving remote files.

According to the WordPress Coding Standards, classes should follow the following conventions:

- Class names should use capitalized words separated by underscores.
- Any acronyms should be all upper case.

Simple, right?

Practically speaking, this would look like the following:

- `class Local_File_Operations {}`
- `class Remote_File_Operations {}`
- `class HTTP_Request {}`
- `class SQL_Manager {}`

To reiterate: classes should also be nouns and should describe the single purpose they serve.

Function Names

As mentioned earlier, if classes are nouns that ideally represent a single idea or single purpose, then their methods should be the actions that they are able to take. As such, they should be verbs - they should indicate what action will be taken whenever they are called.

Furthermore, the arguments that they accept should also factor into the name of the function. For example, if a function is responsible for opening a file, then its parameter should be a file name. Since our goal should make it as easy as possible to read code, then it should read something like "have the local file manager read the file having the following file name."

In code, this may look something like this:

```
// The class definition
class Local_File_Manager {

    public function open_file( $filename ) {
        // Function implementation
    }

}
```

```

// How we'd use this code
$file_manager = new Local_File_Manager();
$file_manager->open_file( 'foo.txt' );
```

Of course, this still doesn't cover *how* functions should be written within the context of WordPress development. The Coding Standards state:

> *Use lowercase letters in variable, action, and function names (never `camelCase`). Separate words via underscores. Don't abbreviate variable names un-necessarily; let the code be unambiguous and self-documenting.*

The first part of of the convention is easy enough to understand; however, I think developers have a propensity to take shortcuts when they are able. "Ah," we think, "`$str` makes sense here, and `$number` make sense here."

Of course, there are always worse - some developers resort to using single characters for their variable names (which is generally only acceptable within loops.)

Just as the Coding Standards state: *Don't abbreviate variable names **un-necessarily**. Let the code be unambiguous and self-documenting.*

Now, the truth is, code can only be unambiguous to a point. After all, that's why it's called code, right? This is why I think code comments should be used liberally.

Anyway, the bottom line is to lower case your method names, avoid all camel casing, separate by spacing, and be as specific as possible when naming your variables.

Variable Names

Variable names actually aren't much different from function names other than they represent a single value or a reference to a particular object. The naming conventions still follow what you'd expect:

- Lower case (versus camelCase)
- Separate spaces with underscores

One other convention that some developers use is what's known as Hungarian Notation which is where the type of value the variable stores is prefixed in front of the variable.

For example:

- Strings will often be represented as `$str_firstname`
- Numbers will be written as `$i_tax` or `$num_tax`
- Arrays may be written as `$arr_range`
- ...and so on

Honestly, the coding standards say nothing about this. On one hand, I do think that this makes for cleaner code in the overall scope of code, but there are a lot of developers who dislike Hungarian Notation.

Since the coding conventions say nothing about them, I'm hesitant to recommend them as I want to stay as close to standards as possible. As such, I have to recommend that it's best to follow the coding standards.

File Names

In keeping consistent with the theme of making our code as readable and self-documenting as possible, it makes sense that we pull this through our source code all the way to the files that we're going to make up our theme, plugin, or application.

In keeping consistent with our previous example, let's say that we're working with `Local_File_Operations` then the file would be named `class-local-file-operations.php`.

Easy enough.

Next, if you're working on a plugin called `Instagram_Foo` then the file should be named `instagram-foo.php`; however, it is worth noting that if you use some type of advanced methods for developing your plugins such as keeping the plugin class file in its own file and then loading it using it another file, then your file structure may be:

- `class-instagram-foo.php`
- `instagram-foo.php`

Where `instagram-foo.php` is responsible for loading the `class-instagram-foo.php`. Of course, this only makes sense if you're using OOP when writing your WordPress plugins.

Function Arguments

When it comes to passing function arguments, it's important to remember that if function names describe the actions that are being taken by the class, then the argument should represent on what the function is actually operating.

From the Coding Standards:

Prefer string values to just `true` and `false` when calling functions.

Since boolean values can be unclear when passing values into a function, it makes it difficult to ascertain exactly what the function is doing.

For example, let's use the above example in a slightly different manner:

```php
// The class definition
class Local_File_Manager {

    public function manage_file( $filename, true ) {

        if ( true ) {
            // Open the file
        } else {
            // Delete the file
        }

    }

}

// How we'd use this code
$file_manager = new Local_File_Manager();
$file_manager->manage_file( 'foo.txt', true );
```

Is more difficult to understand than, say, something like this:

```php
// The class definition
class Local_File_Manager {

    public function open_file( $filename ) {
        // open the file
    }

    public function delete_file( $filename ) {
        // delete the file
    }

}

// How we'd use this code
$file_manager = new Local_File_Manager();
$file_manager->open_file( 'foo.txt' );
$file_manager->delete_file( 'foo.txt' );
```

On top of that, remember that arguments being passed into functions are still variables in and of themselves so they are subject to the variable naming conventions that we've detailed above.

Overview #Overview

WordPress uses the query string to decide which template or set of templates should be used to display the page. The query string is information that is contained in the link to each part of your website.
Put simply, WordPress searches down through the template hierarchy until it finds a matching template file. To determine which template file to use, WordPress:

1. Matches every query string to a query type to decide which page is being requested (for example, a search page, a category page, etc);
2. Selects the template in the order determined by the template hierarchy;
3. Looks for template files with specific names in the current theme's directory and uses the **first matching template file** as specified by the hierarchy.

With the exception of the basic `index.php` template file, you can choose whether you want to implement a particular template file or not.

If WordPress cannot find a template file with a matching name, it will skip to the next file in the hierarchy. If WordPress cannot find any matching template file, the theme's `index.php` file will be used.

When you are using a child theme, any file you add to your child theme will over-ride the same file in the parent theme. For example, both themes contain the same template `category.php`, then child theme's template is used.

If a child theme contains the specific template such as `category-unicorns.php` and the parent theme contains lower prioritized template such as `category.php`, then child theme's `category-unicorns.php` is used.

Contrary, if a child theme contains general template only such as `category.php` and the parent theme contains the specific one such as `category-unicorns.php`, then parent's template `category-unicorns.php` is used.

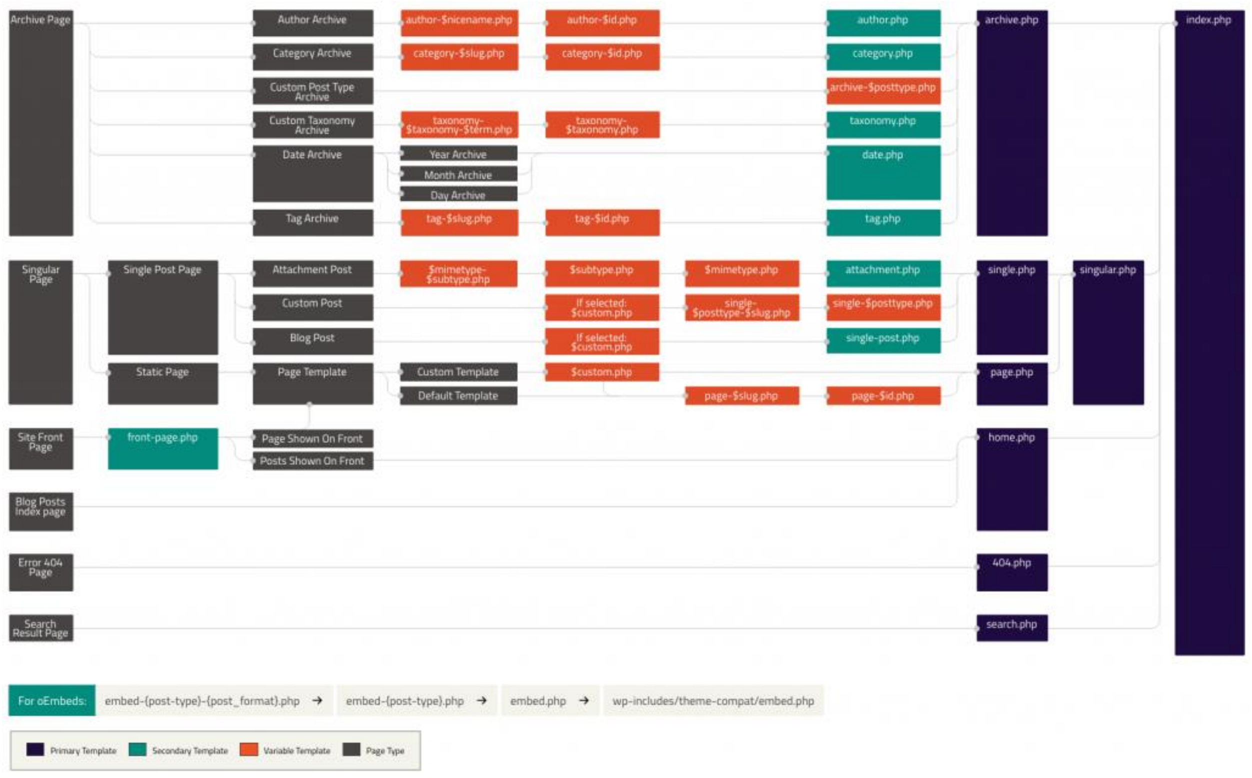
Examples #Examples

If your blog is at `http://example.com/blog/` and a visitor clicks on a link to a category page such as `http://example.com/blog/category/your-cat/`, WordPress looks for a template file in the current theme's directory that matches the category's ID to generate the correct page. More specifically, WordPress follows this procedure:

1. Looks for a template file in the current theme's directory that matches the category's slug. If the category slug is "unicorns," then WordPress looks for a template file named `category-unicorns.php`.
2. If `category-unicorns.php` is missing and the category's ID is 4, WordPress looks for a template file named `category-4.php`.
3. If `category-4.php` is missing, WordPress will look for a generic category template file, `category.php`.
4. If `category.php` does not exist, WordPress will look for a generic archive template, `archive.php`.
5. If `archive.php` is also missing, WordPress will fall back to the main theme template file, `index.php`.

Visual Overview #Visual Overview

The following diagram shows which template files are called to generate a WordPress page based on the WordPress template hierarchy.



You can also interact with this diagram.

The Template Hierarchy In Detail #The Template Hierarchy In Detail

While the template hierarchy is easier to understand as a diagram, the following sections describe the order in which template files are called by WordPress for a number of query types.

Home Page display #Home Page display

By default, WordPress sets your site's home page to display your latest blog posts. This page is called the blog posts index. You can also set your blog posts to display on a separate static page. The template file `home.php` is used to render the blog posts index, whether it is being used as the front page or on separate static page. If `home.php` does not exist, WordPress will use `index.php`.

1. `home.php`
2. `index.php`

Note:If `front-page.php` exists, it will override the `home.php` template.

Front Page display #Front Page display

The `front-page.php` template file is used to render your site's front page, whether the front page displays the blog posts index (mentioned above) or a static page. The front page template takes precedence over the blog posts index (`home.php`) template. If the `front-page.php` file does not exist, WordPress will either use the `home.php` or `page.php` files depending on the setup in Settings → Reading. If neither of those files exist, it will use the `index.php` file.

1. `front-page.php` – Used for both "**your latest posts**" or "**a static page**" as set in the **front page displays** section of Settings → Reading.
2. `home.php` – If WordPress cannot find `front-page.php` and "**your latest posts**" is set in the **front page displays** section, it will look for `home.php`. Additionally, WordPress will look for this file when the **posts page** is set in the **front page displays** section.
3. `page.php` – When "**front page**" is set in the **front page displays** section.
4. `index.php` – When "**your latest posts**" is set in the **front page displays** section but `home.php` does not exist *or* when **front page** is set but `page.php` does not exist.

As you can see, there are a lot of rules to what path WordPress takes. Using the chart above is the best way to determine what WordPress will display.

Privacy Policy Page display #Privacy Policy Page display

The `privacy-policy.php` template file is used to render your site's Privacy Policy page. The Privacy Policy page template takes precedence over the static page (`page.php`) template. If the `privacy-policy.php` file does not exist, WordPress will either use the `page.php` or `singular.php` files depending on the available templates. If neither of those files exist, it will use the `index.php` file.

1. `privacy-policy.php` – Used for the Privacy Policy page set in the **Change your Privacy Policy page** section of Settings → Privacy.
2. `custom template file` – The page template assigned to the page. See `get_page_templates()`.
3. `page-{slug}.php` – If the page slug is `privacy`, WordPress will look to use `page-privacy.php`.
4. `page-{id}.php` – If the page ID is 6, WordPress will look to use `page-6.php`.

5. `page.php`
6. `singular.php`
7. `index.php`

Single Post #Single Post

The single post template file is used to render a single post. WordPress uses the following path:

1. `single-{post-type}-{slug}.php` – (Since 4.4) First, WordPress looks for a template for the specific post. For example, if post type is `product` and the post slug is `dmc-12`, WordPress would look for `single-product-dmc-12.php`.
2. `single-{post-type}.php` – If the post type is `product`, WordPress would look for `single-product.php`.
3. `single.php` – WordPress then falls back to `single.php`.
4. `singular.php` – Then it falls back to `singular.php`.
5. `index.php` – Finally, as mentioned above, WordPress ultimately falls back to `index.php`.

Single Page #Single Page

The template file used to render a static page (`page` post-type). Note that unlike other post-types, `page` is special to WordPress and uses the following path:

1. `custom template file` – The page template assigned to the page. See get_page_templates().
2. `page-{slug}.php` – If the page slug is `recent-news`, WordPress will look to use `page-recent-news.php`.
3. `page-{id}.php` – If the page ID is 6, WordPress will look to use `page-6.php`.
4. `page.php`
5. `singular.php`
6. `index.php`

Category #Category

Rendering category archive index pages uses the following path in WordPress:

1. `category-{slug}.php` – If the category's slug is `news`, WordPress will look for `category-news.php`.
2. `category-{id}.php` – If the category's ID is `6`, WordPress will look for `category-6.php`.
3. `category.php`

4. `archive.php`
5. `index.php`

Tag [#Tag]

To display a tag archive index page, WordPress uses the following path:

1. `tag-{slug}.php` – If the tag's slug is `sometag`, WordPress will look for `tag-sometag.php`.
2. `tag-{id}.php` – If the tag's ID is `6`, WordPress will look for `tag-6.php`.
3. `tag.php`
4. `archive.php`
5. `index.php`

Custom Taxonomies [#Custom Taxonomies]

[Custom taxonomies] use a slightly different template file path:

1. `taxonomy-{taxonomy}-{term}.php` – If the taxonomy is `sometax`, and taxonomy's term is `someterm`, WordPress will look for `taxonomy-sometax-someterm.php.` In the case of [post formats], the taxonomy is 'post_format' and the terms are 'post-format-{format}. i.e. `taxonomy-post_format-post-format-link.php` for the link post format.
2. `taxonomy-{taxonomy}.php` – If the taxonomy were `sometax`, WordPress would look for `taxonomy-sometax.php`.
3. `taxonomy.php`
4. `archive.php`
5. `index.php`

Custom Post Types [#Custom Post Types]

[Custom Post Types] use the following path to render the appropriate archive index page.

1. `archive-{post_type}.php` – If the post type is `product`, WordPress will look for `archive-product.php`.
2. `archive.php`
3. `index.php`

(For rendering a single post type template, refer to the [single post display] section above.)

Author display #Author display

Based on the above examples, rendering author archive index pages is fairly explanatory:

1. `author-{nicename}.php` – If the author's nice name is `matt`, WordPress will look for `author-matt.php`.
2. `author-{id}.php` – If the author's ID were `6`, WordPress will look for `author-6.php`.
3. `author.php`
4. `archive.php`
5. `index.php`

Date #Date

Date-based archive index pages are rendered as you would expect:

1. `date.php`
2. `archive.php`
3. `index.php`

Search Result #Search Result

Search results follow the same pattern as other template types:

1. `search.php`
2. `index.php`

404 (Not Found) #404 (Not Found)

Likewise, 404 template files are called in this order:

1. `404.php`
2. `index.php`

Attachment #Attachment

Rendering an attachment page (`attachment` post-type) uses the following path:

1. `{MIME-type}.php` – can be any MIME type (For example: `image.php`, `video.php`, `pdf.php`). For `text/plain`, the following path is used (in order):
    1. `text-plain.php`
    2. `plain.php`
    3. `text.php`
2. `attachment.php`
3. `single-attachment-{slug}.php` – For example, if the attachment slug is `holiday`, WordPress would look for `single-attachment-holiday.php`.
4. `single-attachment.php`
5. `single.php`
6. `singular.php`
7. `index.php`

Embeds #Embeds

The embed template file is used to render a post which is being embedded. Since 4.5, WordPress uses the following path:

1. `embed-{post-type}-{post_format}.php` – First, WordPress looks for a template for the specific post. For example, if its post type is `post` and it has the audio format, WordPress would look for `embed-post-audio.php`.
2. `embed-{post-type}.php` – If the post type is `product`, WordPress would look for `embed-product.php`.
3. `embed.php` – WordPress then falls back to embed`.php`.
4. Finally, WordPress ultimately falls back to its own `wp-includes/theme-compat/embed.php` template.

Non-ASCII Character Handling #Non-ASCII Character Handling

Since WordPress 4.7, any dynamic part of a template name which includes non-ASCII characters in its name actually supports both the un-encoded and the encoded form, in that order. You can choose which to use.

Here's the page template hierarchy for a page named "Hello World " with an ID of `6`:

- `page-hello-world- .php`
- `page-hello-world-%f0%9f%98%80.php`
- `page-6.php`
- `page.php`
- `singular.php`

The same behaviour applies to post slugs, term names, and author nicenames.

Filter Hierarchy #Filter Hierarchy

The WordPress template system lets you filter the hierarchy. This means that you can insert and change things at specific points of the hierarchy. The filter (located in the `get_query_template()` function) uses this filter name: `"{$type}_template"` where `$type` is the template type.
Here is a list of all available filters in the template hierarchy:

- `embed_template`
- `404_template`
- `search_template`
- `frontpage_template`
- `home_template`
- `privacypolicy_template`
- `taxonomy_template`
- `attachment_template`
- `single_template`
- `page_template`
- `singular_template`
- `category_template`
- `tag_template`
- `author_template`
- `date_template`
- `archive_template`
- `index_template`

Example #Example

For example, let's take the default author hierarchy:

- `author-{nicename}.php`
- `author-{id}.php`
- `author.php`

To add `author-{role}.php` before `author.php`, we can manipulate the actual hierarchy using the 'author_template' template type. This allows a request for /author/username where username has the role of editor to display using author-editor.php if present in the current themes directory.

```
function author_role_template( $templates = '' ) {
    $author = get_queried_object();
    $role = $author->roles[0];

    if ( ! is_array( $templates ) && ! empty( $templates ) ) {
        $templates = locate_template( array( "author-$role.php",
$templates ), false );
    } elseif ( empty( $templates ) ) {
        $templates = locate_template( "author-$role.php", false );
    } else {
        $new_template = locate_template( array( "author-$role.php" )
);

        if ( ! empty( $new_template ) ) {
            array_unshift( $templates, $new_template );
```

Conclusion

We've taken an extended look at Naming Conventions and Function arguments in the Coding Standards. Hopefully this has helped to provide not only a guide for how to improve certain aspects of your WordPress code, but also to explain the rationale behind some of the practices.

In the next article, we're going to take a look at the significance of single quotes and double quotes within the context of working with strings in WordPress development.

There *is* a difference into how they are interpreted by PHP and there are conditions in which you should use one over the other and we'll be reviewing that in the next article.