Additional considerations for OOP plugin development

Rather than writing plugins (or any PHP code) procedurally, we can leverage Object-Oriented Programming (OOP) to write our plugins. OOP is easier to maintain and extend as the needs of the plugin increase.

The code below is a good starting point for class-based plugins. It doesn't have the full plugin comment but the code should still work (even if it does nothing yet).

```
<?php
if ( ! class_exists( 'Awes'AwesomePlugin'{
  class AwesomePlugin {

    /**
     * Constructor
     */
    public function __construct() {
        $this->setup_actions();
    }

    /**
     * Setting up Hooks
     */
    public function setup_actions() {
        //Main plugin hooks
        register_activation_hook(
          __FILE__,
          array(
            'AwesomePlugin',
            'activate'
          )
        );
```

```
        register_deactivation_hook(
          __FILE__,
          array(
            'AwesomePlugin',
            'deactivate'
          )
        );
    }

    'AwesomePlugin'/**
     * Activate callback
     */
    public static function activate() {
        //Activation code in here
    }

    /**
     * Deactivate callback
     */
    public static function deactivate() {
        //Deactivation code in here
    }
}

  // instantiate the plugin class
  $wp_plugin_template = new AwesomePlugin();
}
```

Constructor differences

One thing that tripped me several times is the difference when working with class methods inside the class. Rather than referencing the methods directly, you must use an array syntax:

```
<?php
    register_activation_hook(
      __FILE__,
```

```
    array(
      'AwesomePlugin',
      'activate'
    )
  );
```

In a normal filter or hook declaration, the second parameter is just the name of the function we want to use as the callback. When working with classes we must use the array syntax.

Visibility Modifiers And The Static Keyword

as you start planning your plugin you might want to consider who can see the code inside your classes. PHP provides the three traditional visibility modifiers: public, protected, and private. The table below shows how they work.

Modifier	Description
public	Allows access from anywhere
protected	Allows access from Class parent and subclasses
private	Allows access from Class only

There's another modifier for class methods and properties: static.

The static keyword allows access to the method or property without having to instantiate the class.

```
<?php
class Foo {
  public static function aStaticMethod() {
    echo 'Inside aStaticMethod';
  }
}

Foo::aStaticMethod();
```

```
$classname = 'Foo';
$classname::aStaticMethod();
```

Composer or no Composer

Composer is a dependency manager for PHP, similar to what NPM does for Node.js or Bundler does for Ruby Gems.

> There are two reasons why I would use Composer in a project:
>
> - I have external dependencies
> - I want classes and libraries, both local and installed with Composer, to autoload

Using composer requires a slightly different approach to writing plugins. We first need to run `composer init`, and answer the questions it asks, to create the project's `composer.json` file and the necessary directories.

Then I change the location of third-party libraries in `composer.json` and run `composer install` to install the libraries in their new location.

```
{
  "config": {
    "vendor-dir": "lib/"
  }
}
```

The final step to get a plugin ready to use Composer autoload is to add the following statement to the main class file:

```
<?php
require __DIR__ . '/lib/autoload.php';
```

Whether you need to use Composer or not is a matter of personal preference and the project's complexity

Embracing Complexity

Coming from Javascript it's always tempting to write all the code for a project in a single file. For some small projects, that's OK, but for larger projects, it may not be.

It's OK, and sometimes it's considered a best practice, to break the project into a set of components, each living on its own file.

require and require_once allow you to load a file from the file system.

Using Composer (as discussed earlier) allows you to load files and external libraries using the autoloader.

It's worth considering the complexity and the number of files you expect to have in your project as you're getting started. Adding Composer from the start may save you many headaches later on.

PHPDoc is your friend, use it

Documenting the code is a good practice and can help you down the road. If you're working on a team and need to onboard people, having document comments in your code will make it easier for the new teammates to get up to speed. another instance where commenting the code is useful is when you leave your project for a while and then have a hard time remembering what it did or how it's supposed to work.

PHPDocumentor) is a combination of comment styles (DocBlocks) and a parser that will generate documentation from the comments on your code.

A DocBlock is a piece of documentation in your source code that informs you what the function of a Structural Element is.

phpDocumentor follows the PHPDoc definition and recognizes the following Structural Elements:

- Function
- Constant
- Class
- Interface
- Trait

- Class constant
- Property
- Method

In addition to the above, the PHPDoc standard also supports DocBlocks for Files and include/require statements, even though PHP itself does recognize this as a language structure.

Each of these elements can have exactly one DocBlock associated with it, which directly precedes it. No code or comments may be between a DocBlock and the start of an element's definition.

WordPress uses a customized version of PHPdoc to annotate the source code. Documentation in WordPress mostly takes the form of either formatted blocks of documentation or inline comments.

The following is a list of what should be documented in WordPress files:

- Functions and class methods
- Classes
- Class members (including properties and constants)
- Requires and includes
- Hooks (actions and filters)
- Inline comments
- File headers
- Constants

Where the core PHPDoc documentation and the WordPress Documentation Standards disagree, the WordPress Documentation Standards take precedence.

Note:

This conversation is only applicable to PHP. If you're working with Javascript, check the JavaScript Documentation Standards and JSDoc 3 Standard for more information.