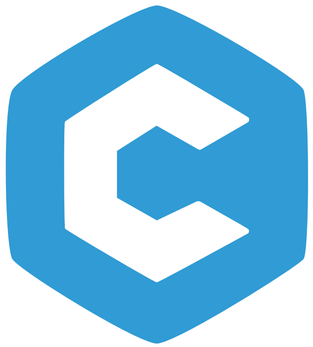


CHROMATIC

http://chromatichq.com

@ChromaticHQ

Code Standards: It's Okay to be Yourself, But Write Your Code Like Everyone Else

Alanna Burke

@aburke626

Twitter, Drupal.org

DrupalCon Baltimore 2017

CHROMATIC

What are Coding Standards?

stand·ard

/ˈstandərd/

noun

1. a level of quality or attainment.

CHROMATIC

Types of Standards

- Style
  - indentation
  - spacing
  - line length
- Substance
  - correct use of functions and components

Why are they important?

"The point of a coding style standard is not to say one style is objectively better than another in the sense of the specific details ... Instead, the point is to set up known expectations on how code is going to look."

- Paul M. Jones

http://paul-m-jones.com/archives/34

Why are they important?

- Keep code readable.
- Spend less time worrying about what your code looks like, and more time coding.
- Help bridge the gap between coders of different backgrounds, languages, locations - especially important in OSS projects such as Drupal where developers from around the world work on the same code.
- Automation (we'll get into this more later).

Who decides coding standards?

We do! Standards are decided by the Drupal community.

How are they decided?

- Drupal's standards are based on the PEAR coding standards.
- They are decided in an issue queue on drupal.org.

How do we implement Coding Standards?

Read the coding standards and keep them handy.

- They’re a living document - they can change!
- Make sure you have them bookmarked for reference.

https://www.drupal.org/coding-standards

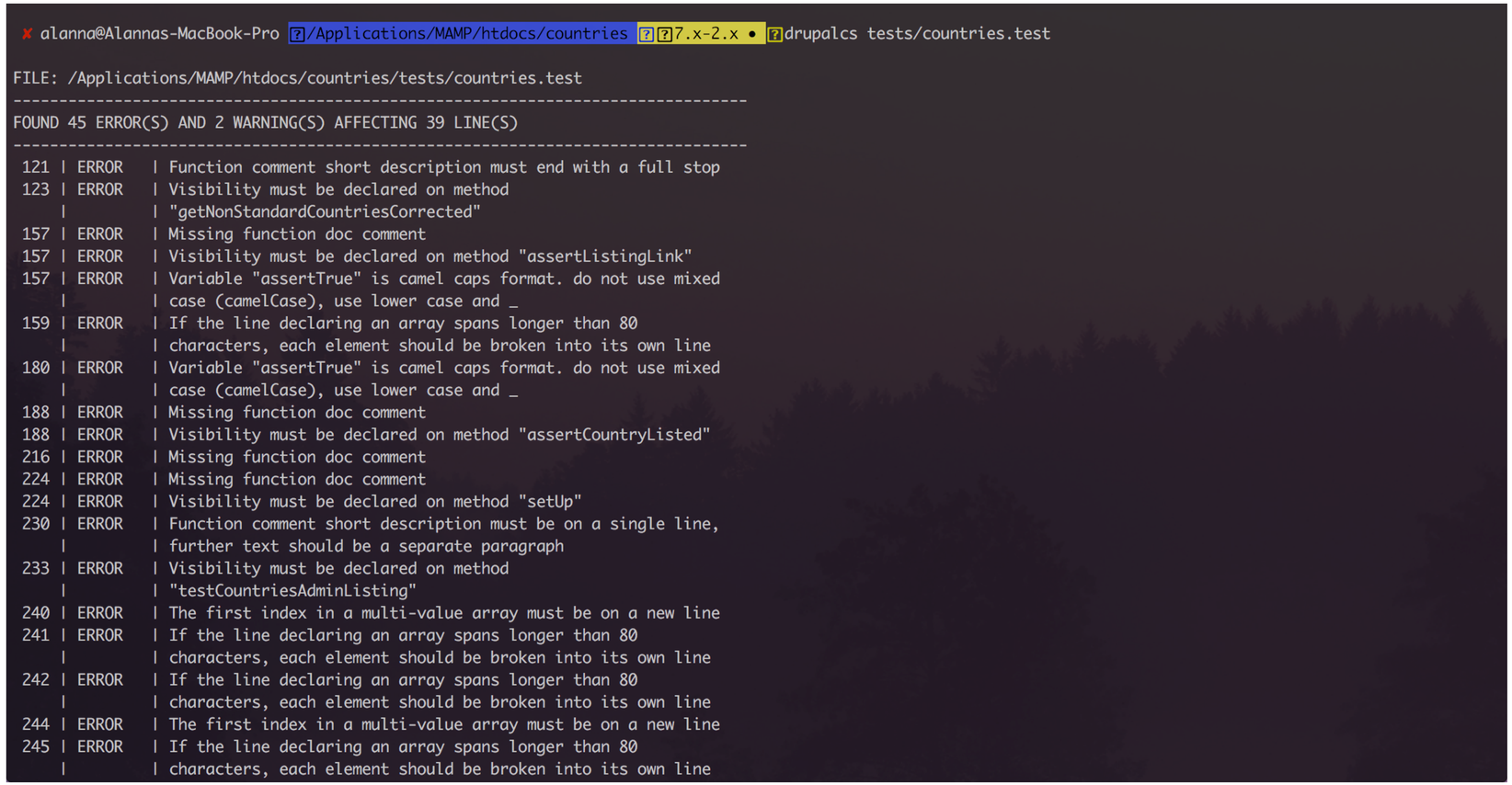
CHROMATIC

Set up your editor for success

Let your editor do the work!

- Sublime Text
- PHPStorm
- Others

Review your own code with Coder/PHPCodeSniffer



https://chromatichq.com/blog/learn-and-enforce-coding-standards-php-codesniffer

Team Code Reviews

- Make the time!
- The most successful teams build in time to review one another's code.
- There's no substitute for code reviews by another person.
- Make sure that you view them as an essential part of your process.
- https://chromatichq.com/blog/code-reviews-are-worth-their-weight-gold

CHROMATIC

Two things to remember:

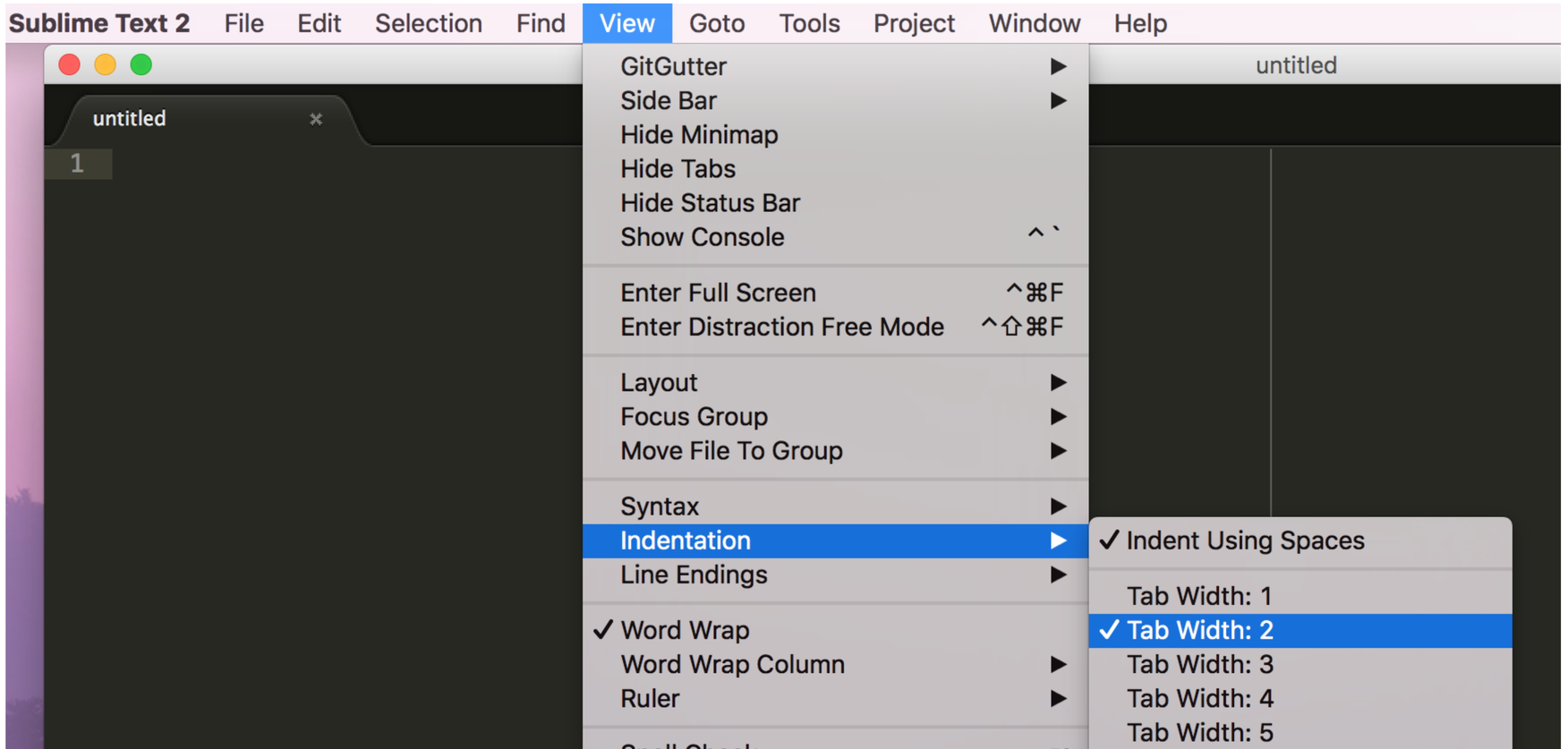
1. Treat others as you would like to be treated - Be kind, courteous, respectful, and constructive. Be aware of your tone.
2. Take everything in stride, and don't take it personally. Those reviewing your code want it to be good, and corrections aren't a personal attack.

Formatting

CHROMATIC

Indentation

- **Two spaces.**
- This is easy to set up in Sublime Text:



Whitespace

- **No trailing whitespace**! (This means no spaces or tabs after the end of a line.)
- This can also be set up in your editor.
- **Use blank lines sparingly** to keep crowded code readable, if necessary.
- But try to **avoid extra blank lines** throughout your files and functions.

File endings

- Drupal uses **Unix file endings**. (The difference between Windows and Unix file endings is what characters are put to indicate the end of the file.)
- There must be **a single blank line** at the end of each file.
- Another thing most **text editors can do for you**!

Arrays

- New short array syntax standards!
  - `["kittens", "puppies", “bunnies”]` ✅
- old arrays:
  - `array("cows", "chicken", “sheep")` ❌

From drupal.org: *Please note, short array syntax is unsupported in versions of PHP prior to 5.4. This means that Drupal 7 core and Drupal 7 contributed projects without an explicit PHP 5.4+ requirement must use long array syntax.*

Line Length - Arrays

If you have an array declaration that's longer than 80 characters, split it into a **multi-line array**, like so:

```
$items['advanced_forum_l'] = [
    'variables' => [
      'text' => NULL,
      'path' => NULL,
     'button_class' => NULL,
    ],
  ];
```

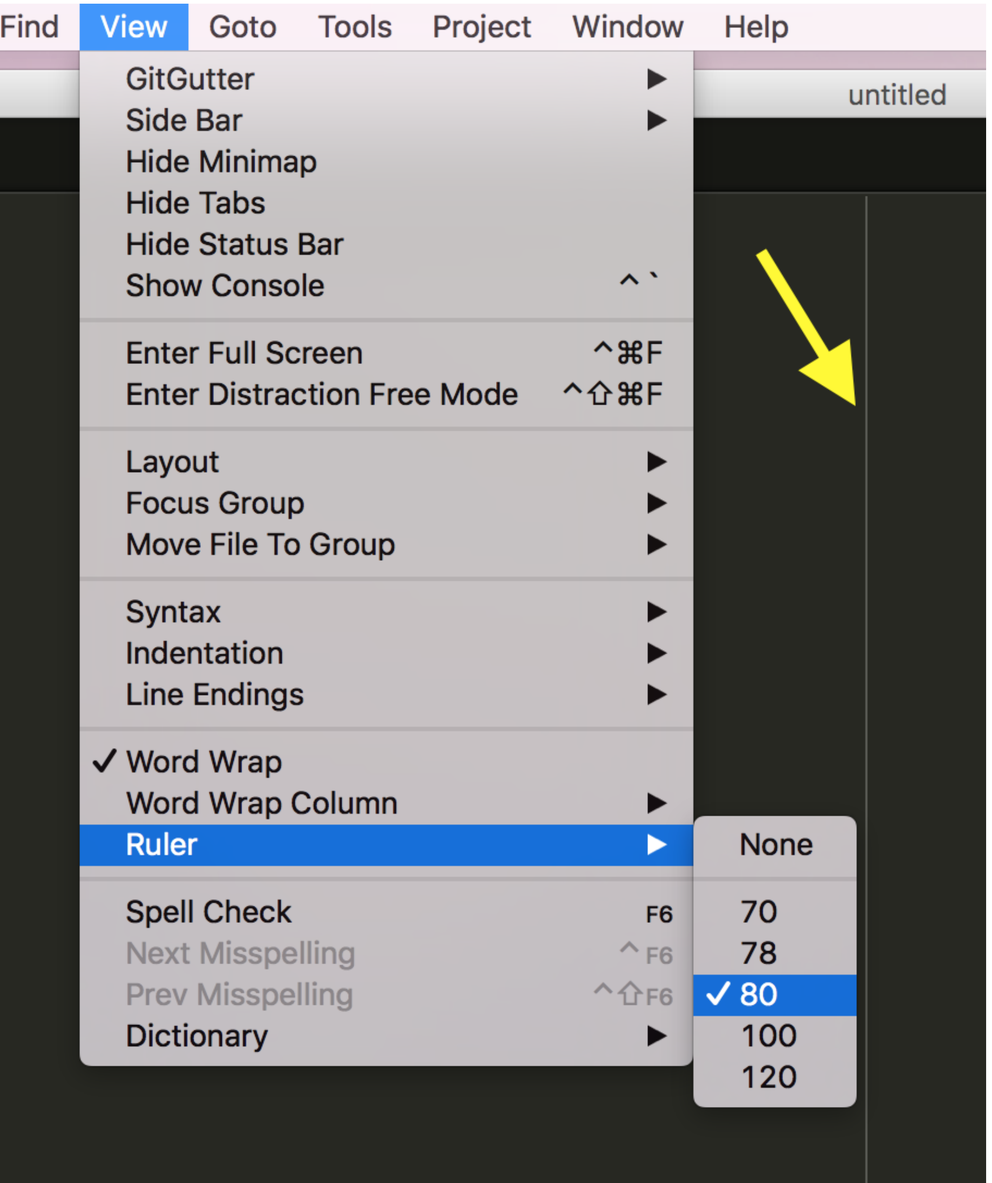
Line Length - Arrays

- Each item is **on its own line**.
- Each item is **followed by a comma**, even the last item.
- This is Drupal best practice regarding arrays in PHP.

Line Length - Arrays

- If you have a super long array (100s of items), you could break each item into its own line.
- That would be very long, but very readable.
- If the programmer who looks at this code next is unlikely to need this info at their fingertips, consider importing it from a csv file or similar, and keeping it out of your code.

Line Length



- Lines should be 80 characters long.
- If breaking up your code over multiple lines makes it less readable, reconsider!
- The goal is **readability.**
- Comment and documentation text, should **always** be 80 characters or under.
- Set up a ruler in your editor.

Operators

- There should always be **one space around operators.**

  (**=**, **-**, **+**, *****, **=>**, **.**, etc).

- You do not need spaces just inside of parentheses.

Operators

Here's an example without proper spaces, to show how difficult it is to read:

```
if($a='system'||$b=='system'){
  return $a=='system'?-1:1;
}
```

And properly formatted:

```
if ($a == 'system' || $b == 'system') {
  return $a == 'system' ? -1 : 1;
}
```

Function Calls & Declarations

- When declaring a function, there should always be **a single space** after the argument list and before the opening curly brace.
- The function then **begins on the next line**, indented with 2 spaces.
- The closing brace **goes on its own line**.

Function Calls & Declarations

- A function call always has **a set of parentheses**, with **no spaces** on either side of them, whether or not there are parameters.
- If there are parameters, they should be **separated by a comma**, **followed by a space.**

Function Calls & Declarations

This update hook from the Advanced Forum contrib module is a simple example of both a function declaration and function call:

```
function advanced_forum_update_7200() {
  if (variable_get('advanced_forum_forum_disabled') == NULL) {
    variable_set('advanced_forum_forum_disabled', FALSE);
  }
```

Constants

- Notice the all caps in the code on the previous slide?
- `TRUE`, `FALSE`, and `NULL` are **always capitalized** in Drupal code.
- They are constants, which are always in all caps in Drupal.
- Custom constants **must be prefixed with the module name**.

Constants

Here's an example from the CKEditor module:

```
define('CKEDITOR_FORCE_SIMPLE_TOOLBAR_NAME', 'DrupalBasic');
define('CKEDITOR_ENTERMODE_P', 1);
define('CKEDITOR_ENTERMODE_BR', 2);
define('CKEDITOR_ENTERMODE_DIV', 3);
```

Control Structures

- When using control structures like `if`, `else`, `elseif`, `case`, `switch`, `foreach`, `while`, `do`, etc., **there should always be a space** after the control structure term.
- Also, there should **always be a space** before the opening curly brace.
- The **statement is indented** on the next line.
- The closing brace is **on its own line**.
- (much like functions)

Control Structures

- Inline control structures are not permitted in Drupal, although they are valid PHP. You **should not use either** of the following structures in Drupal:

```
if($foo) echo bar();
```

OR

```
if($foo)
  echo bar();
```

Control Structures

- Control structures **must always have braces**, and the statement(s) must always be **on the next line**.
- Note that in Drupal, **the standard is to use `elseif` as one word**, not `else if`. Both are valid PHP, but the Drupal standards specify it as one word.

Alternate control statement syntax for theme templates

- Use **`if ():`** and **`endif;`** without braces.
- Statements must still be **on their own line**, as must the **`endif`** statement.

Alternate control statement syntax for theme templates

Here's an example from the Zen subtheme:

```
<?php if ($comments && $node->type != 'forum'): ?>
    <h2 class="comments__title title"><?php print t('Comments'); ?></h2>
<?php endif; ?>
```

Casting

- The **type is wrapped in parentheses**.
- Always **put a space** between the type and the variable.
- Example from the Big Menu contrib module:

```
$p_depth = 'p' . (string) ((int) $depth + 3)
```

- Note that there is a space after `(string)` and after `(int)`.

Semicolons

- Every PHP statement ends with a semicolon. Always!

;

PHP tags

- All PHP files begin with an opening tag: <?php but **never, ever use a closing tag!**
- Also, **never use php short tags** (<? ?>)
- Why?
- Because whitespace or other characters after a closing tag can cause errors, so allowing PHP to close it on its own eliminates those errors.

Documentation

Why is documentation important?

It tells us what to expect from our code.

But I know what my code does!

But my code is so good, it's self-documenting!

Doc Blocks

- Specially formatted blocks of information that go both at the top of each PHP file and before each function.

File Doc Blocks

- Go **at the top of a file** to give you an overview of what the file contains.
- There should be **one blank line** between the opening `<?php` tag and the doc block.
- First line: `@file`.
- On the next line after the `@file` tag: short description.
- The `@file` doc block may also commonly include `@author` and `@version`.

File Doc Blocks

- Example from the Backup and Migrate module:

```
<?php

/**
 * @file
 * Create (manually or scheduled) and restore backups of your Drupal MySQL
 * database with an option to exclude table data (e.g. cache_*).
 */
```

File Doc Blocks

- In something like a template file, you'll see more detail in the `@file` doc block, because the rest of the file may not have as much documentation.
- The `@file` doc block may often spell out available variables.

Function Doc Blocks

- A function doc block goes **just before every function** in every PHP file.
- **No exceptions**.
- Ever.
- Only a **one-line description** is required.
- You should include any **parameters and return types**.

Function Doc Blocks

```
/**
 * Restore from a file in the given destination.
 *
 * @param string $destination_id
 *   The machine-readable path of the backup destination.
 * @param object|string $file
 *   The file object, or its name.
 * @param object|array $settings
 *   A settings object, or array to create a settings object.
 *
 * @return object|bool
 *   Returns the file, or FALSE if the restore had any errors.
 */
```

Tags

- There are a variety of tags you can use in your doc blocks.
- They go in a certain order:

1. One-line summary, ending in a period.
2. Additional paragraph(s) of explanation.
3. `@var`
4. `@param`
5. `@return`
6. `@throws`
7. `@ingroup`
8. `@deprecated`
9. `@see`
10. `@todo`
11. `@Plugin` and other annotations

Tags

- Each type of tag should be **separated by a blank line**.
- The **most-used tags** are probably `@param` and `@return`.

```
/**
 * Generate a country form.
 *
 * @ingroup forms
 *
 * @see countries_admin_form_validate()
 * @see countries_admin_form_submit()
 */
```

Here's an example of how the Countries module uses some of the other tags.

Implements hook_xyz().

When implementing a hook (like hook_menu) simply put:

```
/**
 * Implements hook_menu().
 */
```

Implements hook_xyz().

If you put more than this, coder will give you a warning, like:

```
8 | WARNING | Format should be "* Implements hook_foo().", "* Implements
  |         | hook_foo_BAR_ID_bar() for xyz_bar().",, "* Implements
  |         | hook_foo_BAR_ID_bar() for xyz-bar.html.twig.", or "* Implements
  |         | hook_foo_BAR_ID_bar() for xyz-bar.tpl.php.".
```

CHROMATIC

Implements `hook_xyz()`.

Don’t duplicate function documentation. You’ll get a warning like:

```
11 | WARNING | Hook implementations should not duplicate @param documentation
```

CHROMATIC

API Module

- **Why are these docblocks so important?**
- Why do they have to be **formatted so exactly**?
- The API Module parses the information in doc blocks into **human-readable documentation**.
- The documentation found at https://api.drupal.org/ is all generated this way.

Inline Comments

- Drupal generally uses the C++-style `//` notation.
- C-style comments (`/* */`) are allowed, but discouraged within functions.
- Inline comments shouldn't follow a statement - this means they **must get their own line**.
- Inline comments must always **end in a full stop**.
- Must **never be longer than 80 characters**.

CHROMATIC

Content Style Guide

- Drupal.org has a style guide for content on the site.
- Style of various industry-related terms, along with Drupal specific terms.
- https://www.drupal.org/drupalorg/style-guide/content

CHROMATIC

The `t()` Function

What does the `t()` function do?

- **Translates a given string to a given language** at run-time if you have more than one language enabled.
- Allows for **localization**.
- Wrap your **user-facing strings** in this function so that they can be translated.
- Depending on which placeholder you use, it runs different **sanitization** functions.

When/where do I use it?

- Pretty much everywhere!
- Every user-facing string.
- This ensures your site can be localized.
- **When in doubt, translate everything.**

Parameters

1. **The string to be translated.**
2. (optional) Array of replacements, if any.
3. (optional) Array of options.

`$options` array

from drupal.org:

`$options`: An associative array of additional options, with the following elements:

* `langcode` (defaults to the current language): The language code to translate to a language other than what is used to display the page.
* `context` (defaults to the empty context): A string giving the context that the source string belongs to.

What is string context?

String context (or translation context) is **a way to organize translations when words have 1 to many translations**.

From the handbook page:

- *Each original (English) string can have only one translation.*
- *This is a problem when one English word has several meanings, like "Order", which can mean the order of elements in a list, to order something in a shop, or an order someone has placed in a shop.*
- *For many languages, the string "Order" needs **a different translation for each of these meanings**.*

Read More: https://www.drupal.org/node/1369936

CHROMATIC

Using Placeholders

- **Placeholders** come from the `format_string` function, which is called by `t()`.
- The most common placeholder is probably `@variable`.
- This placeholder runs `check_plain()` on the text before replacing it.
- **Never pass a variable through `t()` directly** - only string literals.
  - The short explanation for this is that the string to be translated needs to be available at runtime, and a variable may not be available and may change its value. You can find an in-depth explanation on StackExchange: http://drupal.stackexchange.com/questions/9362/is-it-always-bad-to-pass-a-variable-through-t.

Using Placeholders

Use a placeholder to insert a value into the translated text, like in this example from the Advanced Forum contrib module:

```
$block->title = t(
  'Most active poster in @forum', array('@forum' => $forum->name)
);
```

`%variable` Placeholder

- Runs `drupal_placeholder()` on the text.
- **Escapes** the text.
- Formats it as **emphasized** text.

!variable Placeholder

Drupal 7

- Inserts your value **exactly as is**.
- **Without** running any sanitization functions.
- Never use this on user-entered text.

Drupal 8

- Deprecated.

`:variable` Placeholder

New in Drupal 8

- For use specifically with **urls**.
- Escaped with `\Drupal\Component\Utility\Html::escape()`.
- **Filtered for dangerous protocols** using `UrlHelper::stripDangerousProtocols()`.

When don't I use `t()`?

In **Drupal 7**, there are some instances where `t()` is not available.

- During the installation phase, `t()` isn't available, so you must use `get_t()`. You can do something like this:

```
$t = get_t();
$t('my string');
```

- Translation is also not used inside of `hook_schema()` or `hook_menu()`.
- In **Drupal 8**, `t()` is always available, so you can always use it.

t() and links - Bad Examples

- **Do not concatenate** t() strings around the link.

```
$do_not_do_this = t('Do not ') . "<a href="api.drupal.org">" . t('link
') . "</a>"  . t('to something like this.');
```

- **Do not use a variable to insert the url & HTML markup** into the text.

```
$bad = t('This is not a good way to make a @link.', array('@link' =>
'<a href=“https://api.drupal.org">' . t('link') . '</a>'));
```

t() and links - Bad Examples

- **Do not insert the entire link markup and url** directly into t().

```
$dreadful = t('This is a dreadful way to make a link pointing to the
<a href="https://api.drupal.org">Drupal API t() documentation</a>.');
```

- Do not insert the l() function into the t() function. It might seem good, but it's redundant.

```
$awful = t('This may seem good, but it's an awful way to link to this
@doc.', array('@doc => l(t(‘documentation'), 'https://
api.drupal.org'));
```

t() and links - Good Examples

- Use t() to **insert** the url.

```
$good = t('Read about the t() function <a href="@api">here</a>',
array('@api' => 'https://api.drupal.org'));
```

t() and links - Good Examples

Here's an example from Drupal 8 Core using `%variable` and `:variable`, in the function `install_check_translations()` in `install.core.inc`:

```
'description' => t('The installer requires read permissions to
%translations_directory at all times. The <a
href=":handbook_url">webhosting issues</a> documentation section offers
help on this and other topics.', array('%translations_directory' =>
$translations_directory, ':handbook_url' => 'https://www.drupal.org/
server-permissions')),
```

It's okay to put a little html in your `t()` function to simplify like this.

Translation Best Practices

- **Think from the point of view of a translator.**
- Try not to abstract out pieces of content too much.

Example:

- In **English**, you may have a blog titled "**Bob's Homepage**."
- Your instinct may be to abstract it like so:

```
$username . "'s " . t('Homepage.');
```

Translation Best Practices

- What's the problem here?
- In other languages, **this phrase may be re-arranged**.
- For example, in French or Spanish, it would be "**Homepage de Bob**."
- This example would require a translator to **change code.**

```
$username . "'s " . t('Homepage.');
```

CHROMATIC

Translation Best Practices

- What's the solution?
- **Less abstraction:**

```
t('@user's Homepage.', array('@username' => 'Bob'));
```

- Can easily be changed without coding to:

```
t('Homepage de @user.', array('@username' => 'Bob'));
```

Concatenation Dos and Don’ts

- **Don’t concatenate strings within** `t()` - Even if you think you have to, there is a better way.

```
t(‘Don’t try to join’ . ‘ ‘ . @num . ‘ ‘ . ‘strings.’, array(‘@num’ =>
‘multiple’));
```

- And don’t concatenate `t()` strings and variables - you don’t need to!

```
t(‘This is a complicated way to join ’) . $mystring . t(‘ and
translated strings’);
```

This would also give you a codesniffer error because **you should not have leading or trailing whitespace in a translatable string**.

Concatenation Dos and Don’ts

- Do this:

```
t(‘This is a simple way to join @mystring and translated strings’,
array(‘@mystring’ => ‘whatever my string is’));
```

- This is how the `t()` function is designed to be used!

Drupal 8 & Twig

- With Drupal 8, we have **the Twig templating engine**.
- This means **new ways to format our text for translation** in templates.
- The simplest way is to pipe your text through `|t`. Here's an example from the Devel contrib module:

```
<thead>
 <tr>
    <th>{{ 'Name'|t }}</th>
    <th>{{ 'Path'|t }}</th>
    <th>{{ 'Info file'|t }}</th>
  </tr>
</thead>
```

CHROMATIC

Drupal 8 & Twig

- The text is **piped into the translation function**.
- Just as it would be passed through `t()` in Drupal 7.
- You can also use `|trans` **interchangeably** with `|t`.
- You can use a `{% trans %}` block to **translate a larger chunk of text** or use placeholders.
- These blocks can also **handle logic for plurals**.

Drupal 8 & Twig

- Here's an example from Drupal 8 Core:

```
<h3 class="views-ui-view-title" data-drupal-selector="views-table-filter-text-source">{{ view.label }}</h3>
<div class="views-ui-view-displays">
  {% if displays %}
    {% trans %}
      Display
    {% plural displays %}
      Displays
    {% endtrans %}:
    <em>{{ displays|safe_join(', ') }}</em>
  {% else %}
    {{ 'None'|t }}
  {% endif %}
</div>
```

Wrapping Up t()

- A lot of what-not-to-do, but now you know!
- Don’t get too creative!
- There is more to dig into with Twig & translations & logic.
  - https://www.drupal.org/developing/api/8/localization

Object Oriented Coding & Drupal 8

CHROMATIC

What is Object Oriented Programming?

- A way of programming that is based on the concept of **objects**, which **represent data in a program**.
- Objects have **properties**, which hold data, and **methods**, which execute functions.
- After an object is created, or **instantiated**, it can be used over and over again.
- Allows for a lot of **reuse and convenience** that procedural programming does not.
- If you're not familiar, check out the OOP Examples project.
  - https://www.drupal.org/project/oop_examples

Note

- All of these examples are from **Drupal 8**.
- While you can certainly use object-oriented code in Drupal 7, and many people have, it's now **mandatory**, so it's best to get used to it.

Declaring Classes

- There should only be one **class**, **interface**, or **trait** per file.
- The file should be **named after the class or interface**.
- Here's an example from the ctools contrib module →

EntityFormWizardBase.php

```php
<?php

/**
 * @file
 * Contains \Drupal\ctools\Wizard\EntityFormWizardBase.
 */

namespace Drupal\ctools\Wizard;

use Drupal\Core\DependencyInjection\ClassResolverInterface;
use Drupal\Core\Entity\EntityManagerInterface;
use Drupal\Core\Form\FormBuilderInterface;
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Routing\RouteMatchInterface;
use Drupal\ctools\Event\WizardEvent;
use Drupal\user\SharedTempStoreFactory;
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

/**
 * The base class for all entity form wizards.
 */

abstract class EntityFormWizardBase extends FormWizardBase implements EntityFormWizardInterface {
```

Declaring Classes

- Class naming is important for **autoloading**. Autoloading allows for classes to be loaded on demand, instead of a long list of require statements. From drupal.org:

  "In Drupal 8, classes will be autoloaded based on the PSR-4 namespacing convention.

  In core, the PSR-4 'tree' starts under `core/lib/`.

  In modules, including contrib, custom and those in core, the PSR-4 'tree' starts under `modulename/src`."

http://www.php-fig.org/psr/psr-4/

https://www.drupal.org/node/608152#declaring

CHROMATIC

Declaring Classes

- From the PSR-4 Autoloader documentation (which is quite brief and worth looking over):

  "This PSR describes a specification for autoloading classes from file paths… This PSR also describes where to place files that will be autoloaded according to the specification."

- So all that's going on here is that PSR-4 is telling you **how to create your file paths so that classes can be autoloaded**.

http://www.php-fig.org/psr/psr-4/

A note on the file docblock

- The current Drupal standards state:

  "The `@file` *doc block MUST be present for all PHP files, with one exception:* ***files that contain a namespaced class/interface/trait, whose file name is the class name with a .php extension****, and whose file path is closely related to the namespace (under PSR-4 or a similar standard), SHOULD NOT have a* `@file` *documentation block."*

- This was adopted **after** most Drupal 8 code was written.
- This is why you are still seeing `@file` blocks in php files that don't require them.
- I have not edited any of the snippets I am quoting here.
- Be aware that **this is a new standard** that you will probably be seeing adopted in module code.

Namespaces

- Namespaces are a way of organizing codebases.
- First, let's look at an example of a namespace from the Metatag contrib module →

```php
<?php

/**
 * @file
 * Contains Drupal\metatag\Command\GenerateGroupCommand.
 */

namespace Drupal\metatag\Command;

use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;
use Drupal\Console\Command\GeneratorCommand;
use Drupal\Console\Command\Shared\ContainerAwareCommandTrait;
use Drupal\Console\Command\Shared\ModuleTrait;
use Drupal\Console\Command\Shared\FormTrait;
use Drupal\Console\Command\Shared\ConfirmationTrait;
use Drupal\Console\Style\DrupalStyle;
use Drupal\metatag\Generator\MetatagGroupGenerator;

/**
 * Class GenerateGroupCommand.
 *
 * Generate a Metatag group plugin.
 *
 * @package Drupal\metatag
 */

class GenerateGroupCommand extends GeneratorCommand {

  use ContainerAwareCommandTrait;
  use ModuleTrait;
  use FormTrait;
  use ConfirmationTrait;
```

Namespaces

- Now look at the directory structure and that's what we'll see:
- Remember, the PSR-4 directory tree starts **under** `src/`, which is why it's not included in the namespace itself.
- When creating a Drupal module, you should **follow this directory structure** - `<modulename>/src/<namespace>`.

Namespaces

- In the previous example, there is a list of classes to be used in this file.
- Any class or interface with a backslash in it **must be declared** like this at the top of the file.
- These are called "**fully-qualified namespaces**" and they now can be referred to by just the last part of the namespace - the fully-qualified namespace may no longer be used inside the code.
- In the class `GenerateGroupCommand`:
  - The use statements there refer to the same namespaces used at the top of the file, but here we don't use the entire name (no backslash).

Namespaces & Collisions

- If you have two classes with the same name, that's a **collision**.
- Fix it by **aliasing** the namespace.
- Use the **next higher portion** of the namespace to create the alias.
- Here's an example from Drupal core:

```
namespace Drupal\Component\Bridge;

use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Zend\Feed\Reader\ExtensionManagerInterface as ReaderManagerInterface;
use Zend\Feed\Writer\ExtensionManagerInterface as WriterManagerInterface;
```

Global Classes

- An exception to `use` statements is if you are using a global class - in that case, you don't need to `use` anything.
- Here's an example from the Devel contrib module →

```
/**
 * Formats a time.
 *
 * @param integer|float $time A raw time
 *
 * @return float The formatted time
 *
 * @throws \InvalidArgumentException When the raw time is not valid
 */

private function formatTime($time) {
  if (!is_numeric($time)) {
    throw new \InvalidArgumentException('The time must be a numerical value');
  }

  return round($time, 1);
}
```

Indenting and Whitespace

- Formatting basics don't change in OOP, but **there are some OOP-specific conventions.**
- There should be **an empty line between the start of a class or interface definition and a property or method definition**.
- Here's an example from the Token contrib module →

```php
<?php

/**
 * @file
 * Contains \Drupal\token\TokenEntityMapperInterface.
 */

namespace Drupal\token;

interface TokenEntityMapperInterface {

  /**
   * Return an array of entity type to token type mappings.
   *
   * @return array
   *   An array of mappings with entity type mapping to token type.
   */
  public function getEntityTypeMappings();
```

Indenting and Whitespace

- There should be **an empty line between a property definition and a method definition**.
- Here's another example from the Token contrib module →

```
/**
 * @var array
 */
protected $entityMappings;

public function __construct(EntityTypeManagerInterface
$entity_type_manager, ModuleHandlerInterface $module_handler) {
  $this->entityTypeManager = $entity_type_manager;
  $this->moduleHandler = $module_handler;
}
```

Indenting and Whitespace

- There should also be a blank line **between the end of a method definition and the end of a class definition** - a blank line between the ending curly braces.
- Again from the Token module, here we can see that there is a blank line after the last function:

```
/**
   * Resets metadata describing supported tokens.
   */
  public function resetInfo() {
    $this->tokenInfo = NULL;
    $this->cacheTagsInvalidator->invalidateTags([static::TOKEN_INFO_CACHE_TAG]);
  }

}
```

Naming Conventions

- When declaring a **class or interface, use UpperCamel**.
- When declaring a **method or class property, use lowerCamel**.
- Class names shouldn't include "drupal" or "class."
- Interfaces should end with "Interface."
- Test classes should end with "Test."

More detailed conventions: https://www.drupal.org/node/608152#naming

- Here's a good example from the Google Analytics contrib module →

```
/**
 * @file
 * Contains \Drupal\google_analytics\Tests\GoogleAnalyticsBasicTest.
 */

namespace Drupal\google_analytics\Tests;

use Drupal\Core\Session\AccountInterface;
use Drupal\simpletest\WebTestBase;

/**
 * Test basic functionality of Google Analytics module.
 *
 * @group Google Analytics
 */

class GoogleAnalyticsBasicTest extends WebTestBase {
```

Interfaces

- For **flexibility** reasons, it is strongly encouraged that you create interface definitions and implement them in separate classes.
- If there is even a *remote* possibility of a class being swapped out for another implementation at some point in the future, **split the method definitions off into a formal Interface.**
- A class that is intended to be extended **must always provide an Interface** that other classes can implement rather than forcing them to extend the base class.

```php
<?php

/**
 * @file
 * Contains \Drupal\ctools\ConstraintConditionInterface.
 */

namespace Drupal\ctools;

interface ConstraintConditionInterface {

  /**
   * Applies relevant constraints for this condition to the injected contexts.
   *
   * @param \Drupal\Core\Plugin\Context\ContextInterface[] $contexts
   *
   * @return NULL
   */

  public function applyConstraints(array $contexts = array());

  /**
   * Removes constraints for this condition from the injected contexts.
   *
   * @param \Drupal\Core\Plugin\Context\ContextInterface[] $contexts
   *
   * @return NULL
   */

  public function removeConstraints(array $contexts = array());

}
```

Visibility

- All methods and properties of classes must have their visibility declared.
- They can be **public**, **protected**, or **private**.
- Public properties are strongly discouraged.
- Here's an example from the Metatag contrib module →

```php
/**
 * Token handling service. Uses core token service or contributed Token.
 */

class MetatagToken {
  /**
   * Token service.
   *
   * @var \Drupal\Core\Utility\Token
   */
  protected $token;

  /**
   * Constructs a new MetatagToken object.
   *
   * @param \Drupal\Core\Utility\Token $token
   *   Token service.
   */
  public function __construct(Token $token) {
    $this->token = $token;
  }
```

Type Hinting

- Type-hinting is optional, but recommended - can be a **great debugging tool**.
- If an object of the incorrect type is passed, an error will be thrown.
- If a method's parameters expects a certain interface, **specify it**.
- **Do not specify a class as a type**, only an interface.
  - This ensures that **you are checking for a type**, but **keeps your code fluid** by not adhering rigidly to a class.
- **Keep code reusable** by checking only for the interface and not the class, allowing the classes to differ.

```
/**
 * Extends the default PathItem implementation to generate aliases.
 */
class PathautoItem extends PathItem {

  /**
   * {@inheritdoc}
   */
  public static function propertyDefinitions(FieldStorageDefinitionInterface $field_definition)
{
    $properties = parent::propertyDefinitions($field_definition);
    $properties['pathauto'] = DataDefinition::create('integer')
      ->setLabel(t('Pathauto state'))
      ->setDescription(t('Whether an automated alias should be created or not.'))
      ->setComputed(TRUE)
      ->setClass('\Drupal\pathauto\PathautoState');
    return $properties;
  }
```

Chaining

- Chaining allows you to **immediately call a function on a returned object**.
- You've probably most often seen or used this with **database objects**.
- Here's an example from the Devel Node Access contrib module:

```
// How many nodes are not represented in the node_access table?
  $num = db_query('SELECT COUNT(n.nid) AS num_nodes FROM {node} n LEFT JOIN {node_access} na
ON n.nid = na.nid WHERE na.nid IS NULL')->fetchField();
```

- Without chaining, you'd have set the results of that query into an object like `$result`, and then you could use the `fetchField()` function in another statement.

Chaining

- Additionally, to allow for chaining whenever possible, **methods that don't return a specific value should return** `$this`.
- Especially in methods that set a state or property on an object, **returning the object itself is more useful** than returning a boolean or NULL.

Constructors & Instantiation

- Drupal's coding standards discourage directly creating classes.
- Instead, it is ideal to **create a function to instantiate the object and return it**. Two reasons are given for this:

1. The function can be written to be re-used to return different objects with the same interface as needed.
2. You cannot chain constructors in PHP, but you can chain the returned object from a function, and chaining is very useful.

Constructors & Instantiation

- Here's an example from the Metatag contrib module:

```
/**
 * {@inheritdoc}
 */

protected function createGenerator() {
  return new MetatagTagGenerator();
}
```

CHROMATIC

Wrapping Up OOP

If you need more resources, this Object Oriented Programming 101 post:

http://www.drupalwatchdog.com/volume-3/issue-1/object-oriented-programming-101

and this Introduction to Drupal 8 Object-Oriented Concepts:

https://www.acquia.com/resources/webinars/introduction-drupal-8-object-oriented-concepts#animated

have been helpful to our team.

CHROMATIC

Twig in Drupal 8

CHROMATIC

The DocBlock

- Twig files should include a **docblock** like any other Drupal file.
- Sections such as `@see`, `@ingroup`, etc, still apply as they did before Twig, so use as appropriate.
- A note on `@ingroup` themeable from Drupal.org:

  Twig template docblocks should only include @ingroup themeable if the template is providing the default themeable output. For themes overriding default output the @ingroup themeable line should not be included.

Comments

- Comments are wrapped in **the Twig comment indicator**, `{# ... #}`.
- Short and long comments use the same indicator.
- Long comments should be wrapped so that they **do not exceed 80 characters**.
- Comments that span several lines should have the indicators on separate lines.

Comments

- Here's an example of a short comment from Drupal 8 core, the Field UI module:

```
{# Add Ajax wrapper. #}
```

Comments

- Here's an example of a long comment from Drupal 8 core, the Book module:

```
{#
 The given node is embedded to its absolute depth in a top
 level section. For example, a child node with depth 2 in the
 hierarchy is contained in (otherwise empty) div elements
 corresponding to depth 0 and depth 1. This is intended to
 support WYSIWYG output - e.g., level 3 sections always look like
 level 3 sections, no matter their depth relative to the node
 selected to be exported as printer-friendly HTML.
#}
```

Variables

- Variables should **be referenced only by name in the docblock**, with no prefix.
- For example, `foo` instead of `$foo` or `{{ foo }}`.
- The **type should not be included**, as this does not affect theming.
- Variables referenced inline in a docblock should be **wrapped in single quotes**.
- Here's an example from the Token contrib module →

```
{#
/**
 * @file
 * Default theme implementation for the token tree link.
 *
 * Available variables:
 * - url: The URL to the token tree page.
 * - text: The text to be displayed in the link.
 * - attributes: Attributes for the anchor tag.
 * - link: The complete link.
 *
 * @see template_preprocess_token_tree_link()
 *
 * @ingroup themeable
 */
#}
{% if link -%}
  {{ link }}
{%- endif %}
```

Variables

- Variables referenced inline in a docblock should be **wrapped in single quotes**.
- Here's an example from Drupal 8 core, the Comment module→

```
  /*
...
 * - created: Formatted date and time for when the comment was created.
 *   Preprocess functions can reformat it by calling format_date() with the
 *   desired parameters on the 'comment.created' variable.
 * - changed: Formatted date and time for when the comment was last changed.
 *   Preprocess functions can reformat it by calling format_date() with the
 *   desired parameters on the 'comment.changed' variable.
...
*/
```

Expressions

- Twig makes it easy to **check if a variable is available** before using it.
- Just use `if`.
- Here's an example from Drupal 8 core:

```
{% if label %}
   <h2{{ title_attributes }}>{{ label }}</h2>
 {% endif %}
```

Expressions

- Loops are also much simpler in Twig!
- You can **easily use for loops** in Twig.
- Here's an example from the Devel contrib module:

```
 {% for item in collector.links %}
  <div class="sf-toolbar-info-piece">
    <span><a href="{{ item.url }}" title="{{ item.description|
default(item.title) }}">{{ item.title }}</a></span>
  </div>
{% endfor %}
```

Expressions

- Another simple expression that can be done in Twig is **variable assignment**.
- If a variable is needed only in the template, it can be declared directly, as you would anywhere else. Like this:

```
{% myvariable = 'myvariable value' %}
```

HTML attributes

- HTML attributes in Drupal 8 are **drillable**.
- They can **be printed all at once**, or one at a time, using **dot notation**.
- If you do not print them all, they **should all be included at the end**, so that any other attributes added by other modules will be included.
- If you're not familiar with HTML attributes in Drupal 8, here's some Drupal.org documentation:

https://www.drupal.org/docs/8/theming-drupal-8/using-attributes-in-templates

- and the Drupal Twig documentation on HTML attributes:

https://www.drupal.org/node/1823416#attributes

HTML attributes

- Here's an example from Drupal 8 core:

```
 {% if link_path -%}
  <a{{ attributes }}>{{ name }}{{ extra }}</a>
{%- else -%}
  <span{{ attributes }}>{{ name }}{{ extra }}</span>
{%- endif -%}
```

- In this code, the full attributes are printed for the <a> and tags.

CHROMATIC

HTML attributes

- You can also **add or remove a class** in Twig.
- Here's an example from the Views module:

```
{%
  set classes = [
    dom_id ? 'js-view-dom-id-' ~ dom_id,
  ]
%}
<div{{ attributes.addClass(classes) }}>
```

Read more here: https://www.drupal.org/node/2315471

Whitespace Control

- The `{% spaceless %}` tag **removes whitespace between HTML tags**.
- Wrap your code in this tag wherever you want to remove whitespace.
- Here's an example from the Devel contrib module →

```
{% spaceless %}
  <div class="sf-toolbar-info-piece">
    <b>{{ 'Status'|t }}</b>
    <span class="sf-toolbar-status sf-toolbar-status-
{{ request_status_code_color }}">{{ collector.statuscode }}</span> {{ collector.statustext }}
  </div>
  <div class="sf-toolbar-info-piece">
    <b>{{ 'Controller'|t }}</b>
    {{ request_handler }}
  </div>
  <div class="sf-toolbar-info-piece">
    <b>{{ 'Route name'|t }}</b>
    <span>{{ request_route }}</span>
  </div>
{% endspaceless %}
```

Whitespace Control

- The whitespace control character (**-**) **removes whitespace at the tag level**.
- Here's an example from Drupal 8 core:

```
<span{{ attributes }}>
 {%- for item in items -%}
   {{ item.content }}
 {%- endfor -%}
</span>
```

- This can also be used to **remove the space from both sides or just one**, if the character is only on one side.

Caveat regarding newlines at the end of files

- Drupal coding standards require that **all files have a newline at the end of files**.
- If you have PHP codesniffer or any other tests set up for Drupal standards, it will require this.
- However, in Twig, **this may not be wanted** in your template output.
- Until a better community-wide solution is reached, you can alter your tests if you need them to pass, or **add a twig template tag to the end of the file** - you can read this ongoing issue for more clarification:

https://www.drupal.org/node/2082845

Filters

- A filter in twig uses the pipe character, `|`.
- With the `t()` function, we talked about using the new `|t` filter to translate text, but there are other filters that you can use as well.

Filters

- There are a variety of Twig filters and Drupal-specific filters.
- Here's an example of a Twig filter, **join**, from Drupal 8 core:

```
<div class="sf-toolbar-info-piece">
  <b>{{ 'Roles'|t }}</b>
  <span>{{ collector.roles|join(', ') }}</span>
</div>
```

CHROMATIC

Syntax

- These standards are taken from the Twig Documentation:

  twig.sensiolabs.org/doc/coding_standards.html

1. Put one (and only one) space after the start of a delimiter (`{{`, `{%`, and `{#`) and before the end of a delimiter (`}}`, `%}`, and `#}`).

   1.a) When using the whitespace control character, do not put any spaces between it and the delimiter.

Syntax

2. Put one (and only one) space before and after the following operators:
   - comparison operators (==, !=, <, >, >=, <=)
   - math operators (+,-, /, *, %, //, **)
   - logic operators (`not`, `and`, `or`)
   - `~`
   - `is`
   - `in`
   - ternary operator (`?:`)

Syntax

3. Put one (and only one) space after the `:` sign in hashes and `,` in arrays and hashes.
4. Do not put any spaces after an opening parenthesis and before a closing parenthesis in expressions.
5. Do not put any spaces before and after string delimiters.
6. Do not put any spaces before and after the following operators:
    - `|`
    - `.`
    - `..`
    - `[]`

Syntax

7. Do not put any spaces before and after the parenthesis used for filter and function calls.
8. Do not put any spaces before and after the opening and the closing of arrays and hashes.
9. Use **lowercase and underscored variable names** (not camel case).
10. Indent your code inside tags using **two spaces**, as throughout Drupal.

Wrapping Up

- Remember to keep coding standards handy.
- Refer to them often - they change!
- Check out the coding standards issue queue.
- Keep your code clean!

http://chromatichq.com

@ChromaticHQ