**Theming section from “The Definitive Guide to**

**Drupal 7”**

<http://themery.com/dgd7>

THEMING

**Drupal’s theme layer, and the themes that use it, are responsible for the look and feel of a Drupal web site**. Good themes consist of all the same elements that you would find on any reputable web site, including **standards-compliant XHTML markup, CSS, and JavaScript**. How it all comes together is what is so special and what makes Drupal themes so flexible and powerful.

Drupal themes can be as simple or as complex as you need them to be. **Themes have the final say and ultimate control over almost every aspect of each page.** Like Drupal itself, themes are flexible and powerful. Admittedly, taking full advantage of Drupal's theme layer means overcoming a rather steep learning curve, and without a general understanding of what’s going on under the hood, it is easy to make mistakes early on.

In this chapter, you will learn about the basic aspects of Drupal's theme layer. You’ll learn how to go about making customizations and changes in a sustainable way and best practices for common tasks. You will be well on your way to creating flexible and sustainable custom Drupal themes in no time! The next chapter will build on this one and will cover the more advanced intricacies of Drupal themes.

Some of the examples you'll find throughout this chapter and the next can be found in the DGD7 theme. It's available at https://github.com/jacine/dgd7 for download if you'd like to follow along.

**THE CORE THEMES DIRECTORY**

When starting out, one of the first things people do is navigate to the **core /themes** directory and take a look at the files in the themes to get an idea of the general structure and contents. Unfortunately, many people make the mistake of starting out by directly customizing core themes. Do not make this mistake! They usually run into roadblocks and frustration shortly thereafter. Drupal has a large and diverse user base, and the main goal of a core Drupal theme is to cater to the masses.

Aesthetics aside, core themes have many requirements and different use cases to satisfy. A few themes support the Color module in order to make it easy for site administrators to change color schemes in the user interface. This is not a bad thing; however, it can easily become confusing and frustrating when trying to customize colorized themes because CSS is generated programmatically and stored outside of the theme directory. Core themes must also function if used as an administration theme and they must support bidirectional text; in general, they can’t stray far from Drupal’s default regions and settings.

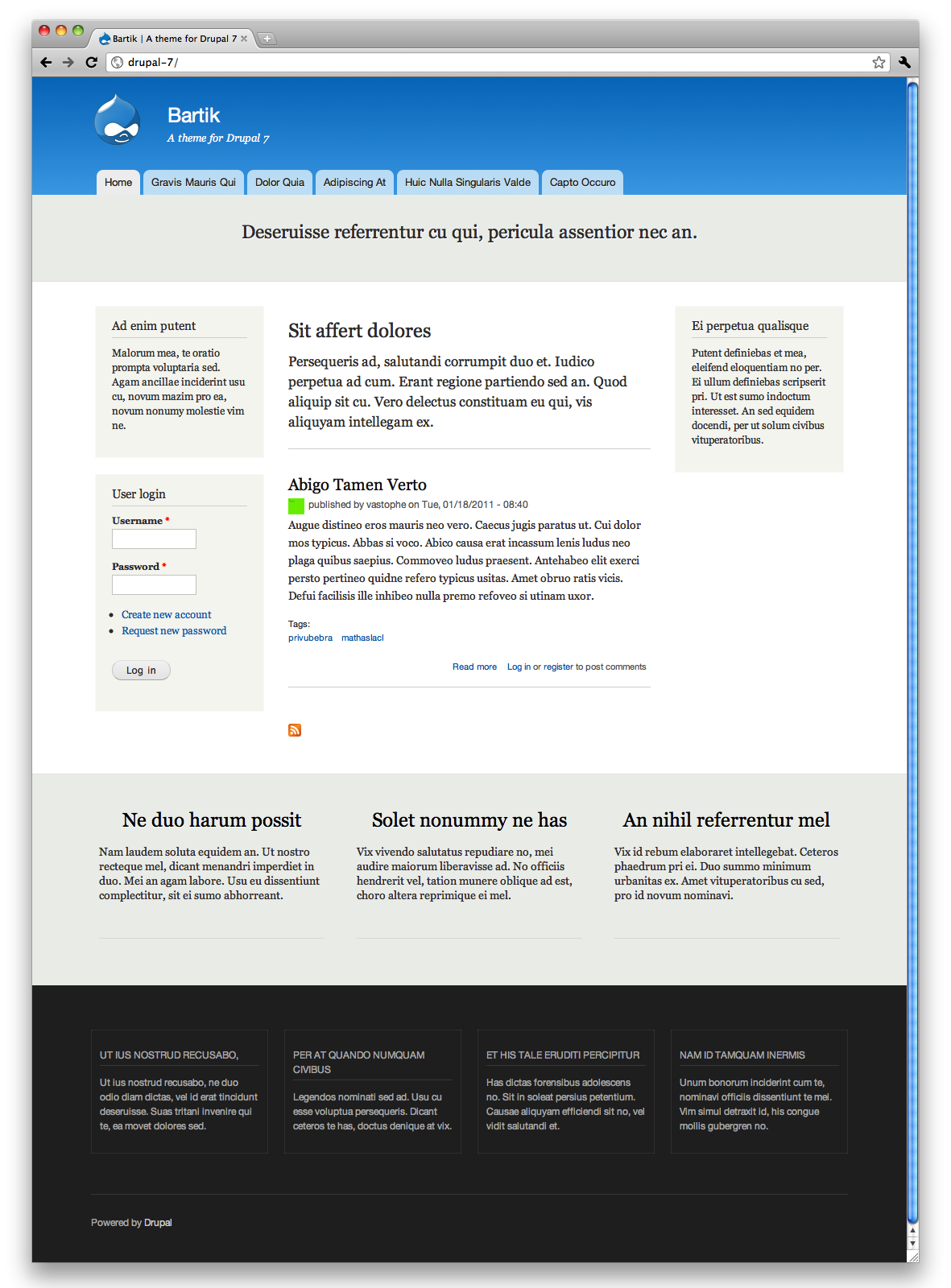
It’s not easy to please everyone, and Drupal core themes have the tough job of trying to do just that. As a result, core themes are nowhere near as flexible or as cutting edge as they could be. Most of the time, your goal and approach will be very different when creating custom themes. You’ll be able to focus on coding your own front-end or back-end focused design, customize the markup, decide which CSS files to use (if any), and other exciting decisions.

# CORE THEMES

Drupal core contains four themes. They are introduced in the following sections.

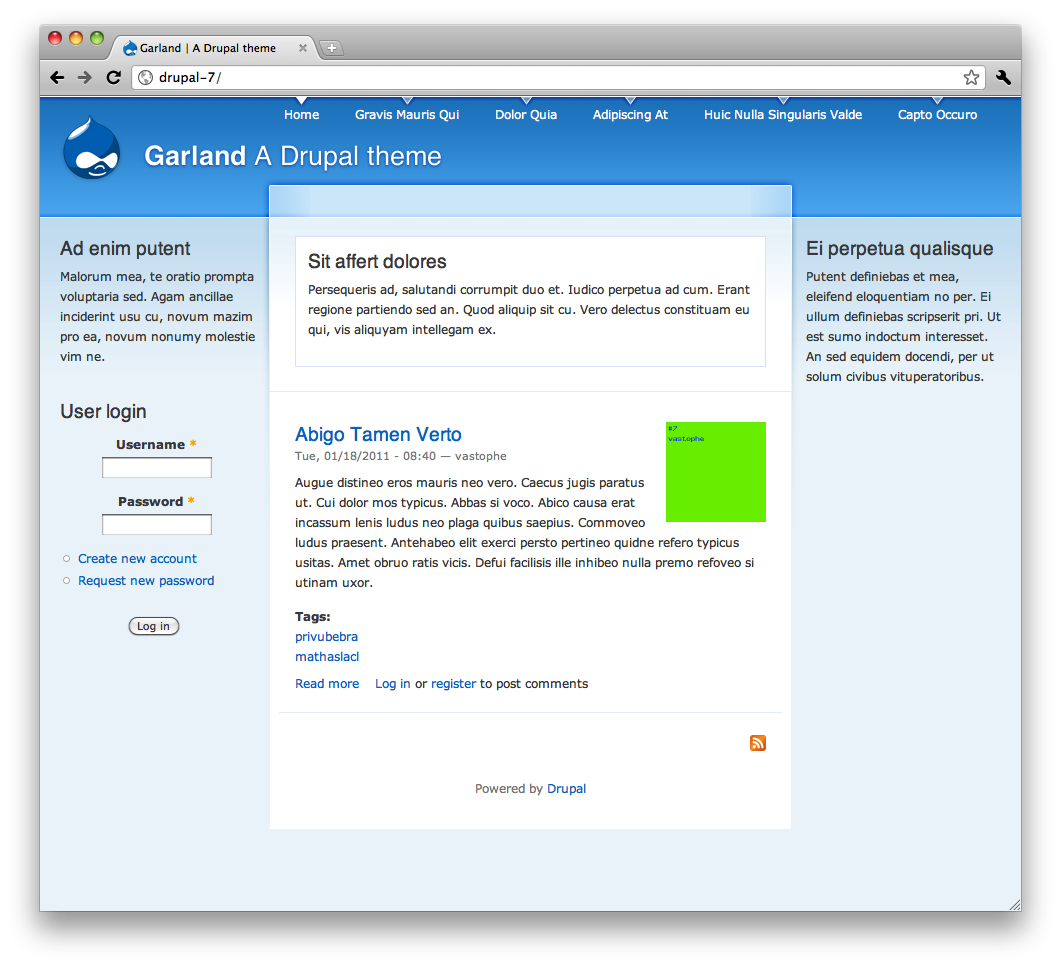
### BARTIK

Bartik is a new and welcome addition to Drupal 7. Drupal enables Bartik as the default user-facing theme upon installation. **It is a clean and simple theme that supports the color module and makes excellent use of regions.** In addition to the default regions Drupal recommends, the Bartik theme has seven custom regions for laying out blocks in the footer and sub-footer.

 Bartik is a clean and simple theme.

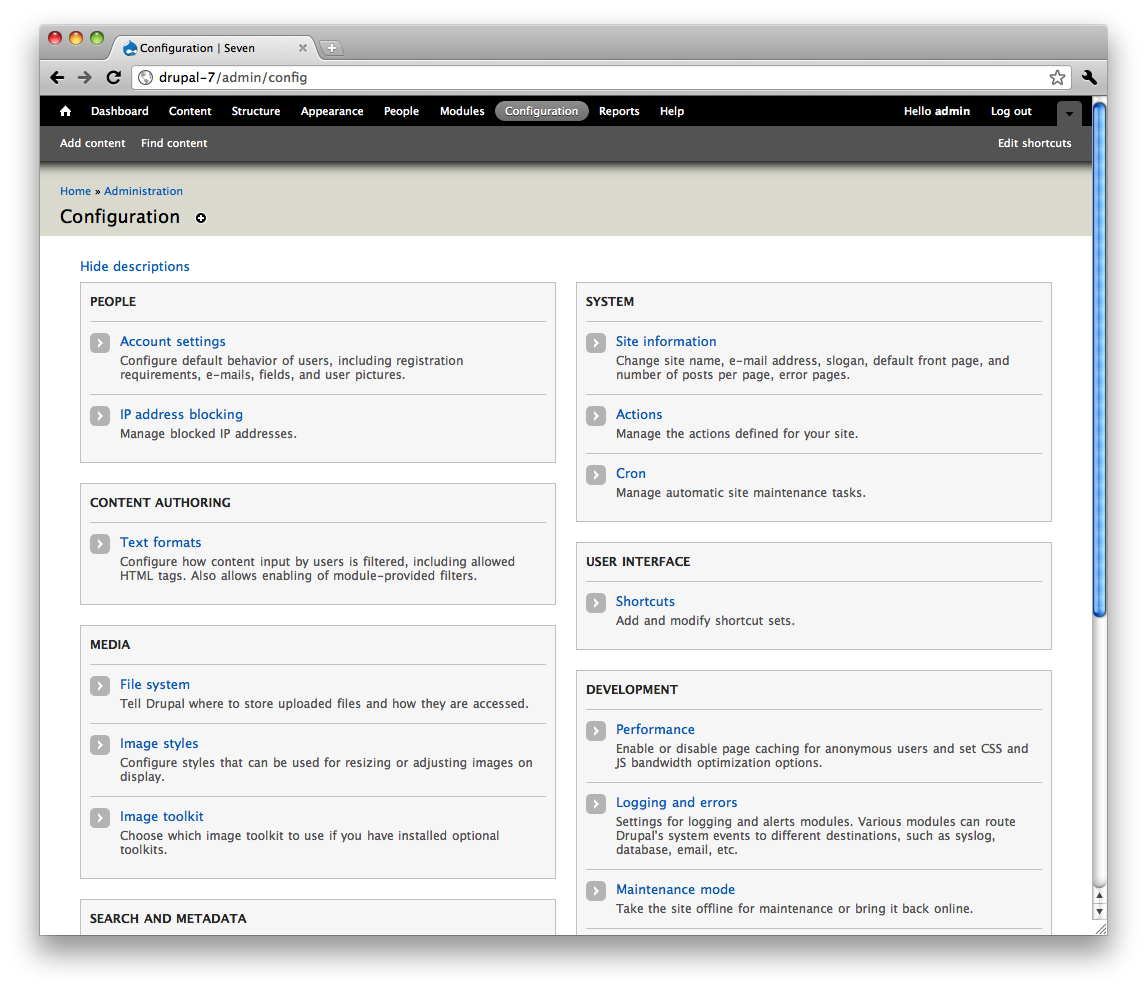
### GARLAND

Garland originally made its debut as a core theme in Drupal 5. It is a more complex theme with excellent color module support. It contains fifteen color schemes and **provides an option to toggle between a fixed or fluid layout**.

 Garland is a more complex theme with excellent color module support.

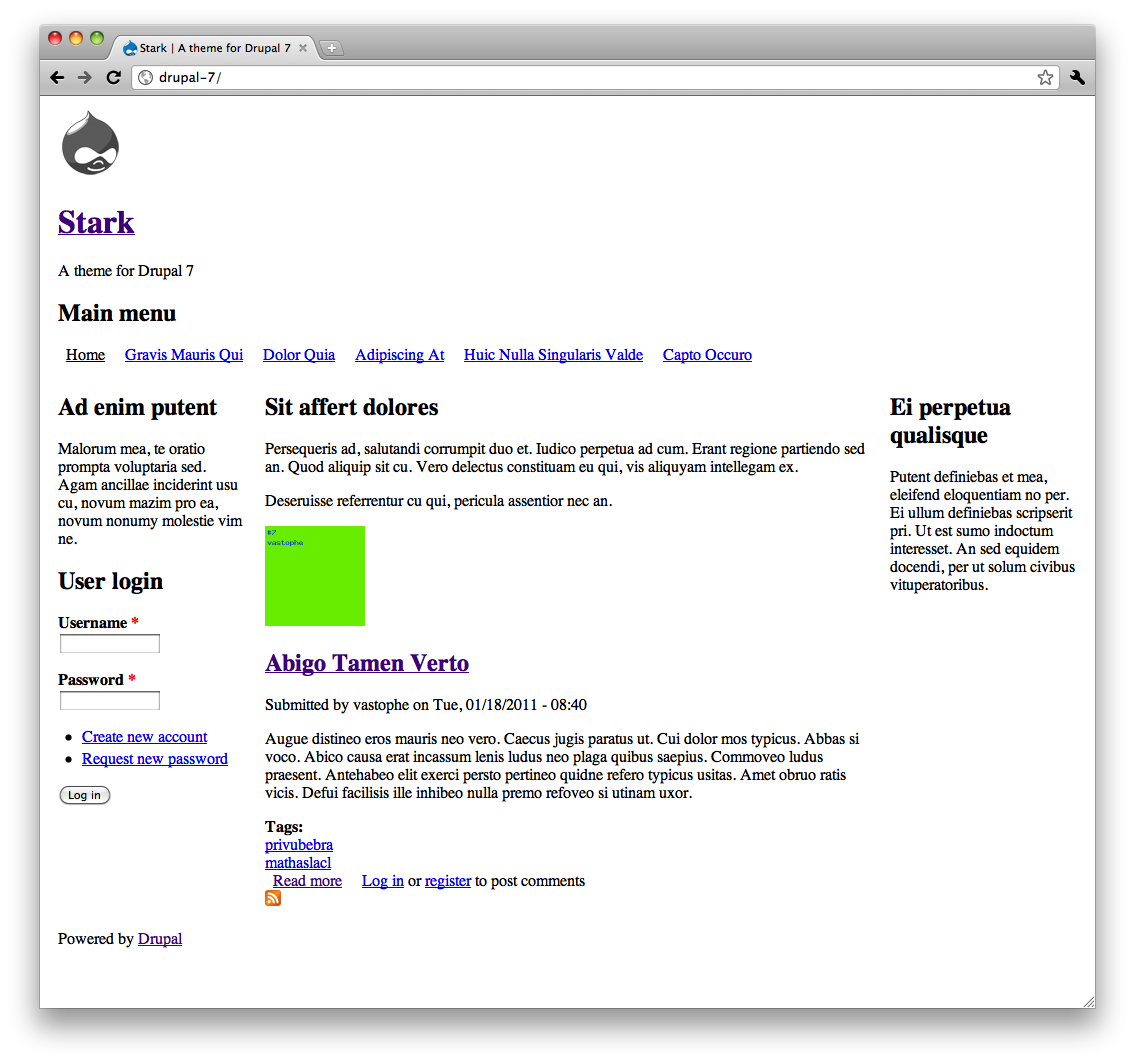
### SEVEN

Also new to Drupal 7, **Seven is Drupal’s default administrative theme**. Born out of the [Drupal 7 User Experience project](http://d7ux.org/), Seven drove many of Drupal’s user interface improvements. It contains very few regions, as its focus is on performing administrative tasks.

 Seven is Drupal’s default administrative theme.

### STARK

Stark is a unique and literally minimal Drupal theme. Its main purpose is to expose Drupal’s default HTML markup and CSS. It **does not** provide any **template files** and barely provides any **CSS** at all, other than basic layout styles that place the default sidebar regions. Don’t let its simplicity fool you; it is actually quite useful. Stark is the perfect theme for developers to code against when writing CSS for their modules. **It can also assist theme developers when trying to troubleshoot issues where they’re not positive if the problem is with their theme or another module**.

 Stark is a unique and literally minimal Drupal theme.

# THEME ENGINES

Drupal’s **theme directory** also has an **engines directory** that contains a theme engine called **PHPTemplate**. **Theme engines provide an easy way to separate themable output into template files as opposed to plain old PHP**. The main benefit of using the **PHPTemplate engine** is that **separating logic from presentation is simplified**. Those who are unfamiliar with PHP can accomplish a great deal because they are able to work in template files that mainly contain markup and print variables.

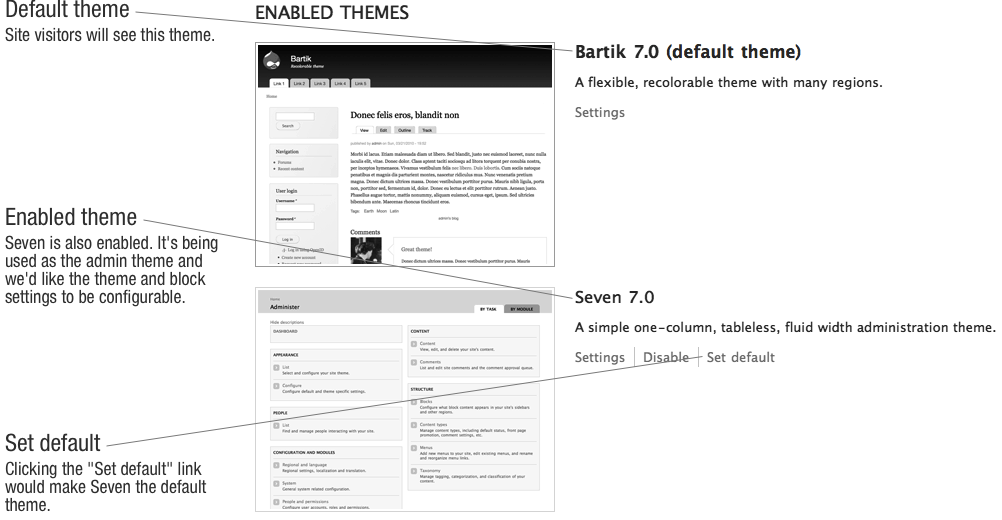
While **other theme engines** such as **Smarty, XTemplate, and PHPTal** may be used, PHPTemplate is Drupal’s default theme engine and is by far the most popular theme engine used by Drupal themes (and many popular contributed modules), so we will cover it in this chapter. It is also possible to write pure PHP Drupal themes. See the Chameleon theme for an example of a pure PHP theme at <http://drupal.org/project/chameleon>. For a full listing of available theme engines, visit <http://drupal.org/project/theme+engines>.

# THEME ADMINISTRATION

**Theme configuration tasks are located in the Appearance section of Drupal’s administration**. This is where you can control things like which themes you want to enable or disable, which settings you want to apply, which color scheme you want to use (if your theme supports the color module), and more.

# ENABLING AND SETTING A DEFAULT THEME

In a fresh installation of Drupal 7, the default theme (Bartik) appears at the top of the Appearance page, followed by other enabled and disabled themes. What is a default theme? Simply enabling a theme is not enough with Drupal. Setting a theme as the default is what makes it the front-end theme (the theme your site visitors will see).

 The Appearance page in a default installation showing enabled themes.

Enabling a theme without setting it as the default is useful when you want your site to utilize multiple themes at once. This setting is typically more useful when used in conjunction with contributed modules. An example of this is the [SwitchTheme](http://drupal.org/project/switchtheme) module, which allows users to change the site theme by selecting the name of a theme from a list that is populated with all enabled themes.

# ADMINISTRATION THEME

# In Drupal 7, the Seven theme is the default administration theme. The administrative theme is used when performing all administrative tasks, most of which happen under the /admin path. You can also choose to allow the administrative theme to be used when editing site content. Although some themes support Drupal’s administrative interface better than others, any Drupal theme can be used as an administration theme if desired.

# The administration theme’s configuration settings are located below the theme listings on the admin/appearance page. To use the same theme in both the front and back end of your Drupal site, simply choose Default theme as the Administration theme.

# GLOBAL THEME SETTINGS

# Drupal comes with some theme settings that can be configured in the administrative interface. This is where most of the site identity assets are defined, as well as a couple of other miscellaneous settings. A Global Settings page located at admin/appearance/settings contains these settings. When global settings are saved, the settings apply to all themes. Each theme also has its own Settings page accessible via a Settings link located next to each enabled theme on the admin/appearance page. When theme settings are applied on an individual theme’s Settings page, they override the global settings. The following sections will detail what each of these are and where you’ll encounter them in your themes.

# Quite a few of these settings determine whether or not variables are populated and therefore printed in template files. The settings pictured in [Figure 15–6](http://themery.com/dgd7/theming/administration/theme-settings#figure-15-6) represent the defaults provided by Drupal. These can be overridden by themes by defining features in the theme’s .info file, which is discussed further in the [*Defining Theme Metadata*](http://themery.com/node/67) section. When specifying features in .info files, you’ll need to make sure you include all the features you want to support, as having just one will override all of the defaults provided by Drupal. The following is a quick reference of these settings as they’d be entered in a .info file:

# features[] = logo

# features[] = name

# features[] = slogan

# features[] = favicon

# features[] = main\_menu

# features[] = secondary\_menu

# features[] = node\_user\_picture

# features[] = comment\_user\_picture

# features[] = comment\_user\_verification

# Screenshot of Global theme settingsFigure 15–6. The Global Settings page.

# LOGO

# By default, Drupal will look for a file named logo.png in the root of the theme directory. There is also an option to specify a path to a different file to use for the logo, as well as the ability to upload a logo to use. When the Logo checkbox is checked, a variable called $logo is populated with the path to the logo, which will be available in page.tpl.php. If unchecked, the logo will not print.

# NAME AND SLOGAN

# The site name is defined during the installation process. Both the site name and slogan can be changed on the admin/config/system/site-information page. On the theme’s Settings page, you can toggle their visibility. Both are available in page.tpl.php as $site\_name and $site\_slogan.

# SHORTCUT ICON

# The shortcut icon, also known as the favicon, is the small Drupal icon that appears in the address bar, bookmarks, and tabs of most browsers. Like the logo, the shortcut icon’s visibility can be toggled and a custom file can be used. The default file is misc/favicon.ico.

# USER PICTURES IN POSTS AND COMMENTS

# These settings control whether or not the variables $user\_picture in node.tpl.php and $picture in comment.tpl.php are populated, and therefore whether or not the pictures are displayed when viewing nodes and comments.

# USER VERIFICATION STATUS IN COMMENTS

# This option will display “(Not verified)” next to the user name for users that do not have a verified account. This text is defined in template\_preprocess\_username() and printed in theme\_username() as $variables['extra']. See the “Preprocess and Process Functions” and “Theme Functions” sections to learn how to change this.

# MAIN AND SECONDARY MENUS

# When the checkboxes for the Main and Secondary menus are checked, $main\_menu and $secondary\_menu variables are populated in page.tpl.php with arrays containing the menu links for each menu. On the Menu Settings page, located at admin/structure/menu/settings, you can choose which menu is used for each. By default, the Main menu, which can be managed at admin/structure/menu/manage/main-menu, is used as the source that populates $main\_menu. The default menu for the source of the Secondary menu is the User menu, which can be managed at admin/structure/menu/manage/user-menu.

# These are simple one-level menus output using the theme\_links() function (which will be covered later in this chapter) in page.tpl.php. This makes them hard to use when styling complex navigation designs. Because complex navigation is often required, many theme developers create regions for navigation and use blocks to output their menus instead of using these menus. We highly recommend the [Menu Block module](http://drupal.org/project/menu_block), which allows you to do pretty much anything you’ll ever need to do with menus very easily.

# CUSTOM THEME SETTINGS

# Custom theme settings are similar to the global theme settings and can be provided by themes and modules. An example of custom theme settings can be found in the Garland theme in the garland.info file. It creates a setting called garland\_width that can be set to fixed or fluid. The Shortcut module also provides a setting to display the “Add or remove shortcut link” used in the Seven theme to provide the icon you see in the Overlay next to the title. To learn how to create custom theme settings for your theme, visit <http://drupal.org/node/177868>.

# In Drupal 7, themes can modify the entire theme settings form. In a theme’s theme-settings.php, themes should now use [THEMENAME\_form\_system\_theme\_settings\_alter(&$form, $form\_state)](http://api.drupal.org/api/function/hook_form_system_theme_settings_alter/7) hook function. This gives the same power to themes that modules have if they use [hook\_form\_system\_theme\_settings\_alter()](https://api.drupal.org/api/drupal/modules%21system%21theme.api.php/function/hook_form_system_theme_settings_alter/7). See the “Forms API Quickstart Guide” and “Forms API Reference” on <http://api.drupal.org/api/7>, as well as the [hook\_form\_FORM\_ID\_alter() docs](http://api.drupal.org/api/function/hook_form_FORM_ID_alter/7) to learn the full flexibility of Forms API. Note that themes can no longer use the phptemplate\_ prefix to the function; you’ll need to use the actual name of your theme as the prefix.

function foo\_form\_system\_theme\_settings\_alter(&$form, $form\_state) {

$form['foo\_example'] = array(

'#type' => 'textfield',

'#title' => t('Widget'),

'#default\_value' => theme\_get\_setting('foo\_example'),

'#description' => t("Place this text in the widget spot on your site."),

);

}

# In order to set the default value for any form element you add, you’ll need to add a simple line to your .info file: settings[SETTING\_NAME] = DEFAULT\_VALUE. For our foo theme, you’d need to edit the foo.info file and add this line:

settings[foo\_example] = blue bikeshed

# In any of your theme’s PHP files, you can retrieve the value the user set by calling:

$foo\_example = theme\_get\_setting('foo\_example');

# INSTALLING A NEW THEME

# Drupal scans its theme directories for available themes, so it’s important that you place your themes in the right place for Drupal to recognize them. You might also be tempted to add themes to Drupal’s /themes directory, but technically this is considered “hacking core” and should be avoided. After downloading and unpacking your theme, choose one of the following directories in which to place the theme. Using one of these directories will help ensure that any updates you make to Drupal itself will not result in accidentally overwriting your theme.

# sites/all/themes

# Use this directory when you want the theme to be available to all sites in your Drupal installation.

# sites/sitename/themes

# Use this directory when you only want the theme available to a specific site in your multisite Drupal installation.

# You may also use the theme installer to download and install contributed themes by clicking the Install new theme link at the top of the Appearance page. This will bring you to a form where you can enter the link to the project download’s tarball location and click Install. The theme installer will automatically download your theme and place it in the sites/all/themes directory. Once completed, you can enable the theme as usual on the admin/admin/appearance page.

# DEFINING THEME METADATA (.INFO FILES)

# .info files (pronounced “dot info files”) contain important metadata about your theme, such as the name of the theme, which version of Drupal it supports, as well as things like the stylesheets and regions the theme will contain. Writing a .info file is usually the first step you take when creating a Drupal theme.

# The first part of the file name is the machine-readable name of the theme, which Drupal uses to store information about your theme in the database. Dashes and other special characters are not allowed. While underscores are allowed, it is considered a best practice to avoid using them when naming your .info file. Use themename.info instead of theme\_name.info. This name will also be used to prefix function names when implementing theme function overrides. When overriding theme\_menu\_link(), for example, a function named themename\_menu\_link() is considered easier to read than theme\_name\_menu\_link() when trying to determine the override being performed.

# CAUTION Your theme (machine) name must be unique. Do NOT to give your theme the same name as any existing modules as it will likely cause namespace issues and make it difficult to track down PHP errors.

# Each theme requires some basic properties to be set in the theme’s .info file. The name, core, and engine properties are the bare minimum requirements for all Drupal themes. The following sections contain a brief description of each available property followed by an example of the syntax.

# TIP To add comments to your .info file, add a semicolon to the beginning of each line, like so: ; This is a comment. Comments are good. Make frequent use of them.

# REQUIRED PROPERTIES

# Core

# Drupal will only allow your theme to be enabled if the core setting is set to support the current major version of Drupal. Major versions are simply 6.x, 7.x or 8.x, and so on. core = 7.x

# Name

# The human-readable name of your theme. It doesn’t need to match or resemble the machine-readable name, so feel free to be creative here. name = Theme Name

# ADDITIONAL PROPERTIES

# Base theme

# Drupal allows themes to establish a relationship with each other. Creating a subtheme allows you to inherit the functionality and assets of the base theme (more on this in the next chapter). When creating a subtheme, you’ll need to specify the base theme. It’s important that the machine name of the base theme is used here. base theme = themename

# Description

# The basic features or purpose of the theme should be described here. The description will appear in the admin/appearance page and may contain HTML. description = The description of my theme

# Engine

# Specifies the theme engine. PHPTemplate is the default and most common, so unless you want to change this, it doesn’t have to be manually set. Other options include smarty and theme for a pure PHP theme (see [Chameleon](http://drupal.org/project/chameleon) for an example). engine = phptemplate

# Featured

# Setting features are a way of overriding Drupal’s default global theme settings. The following is a list of the default theme settings provided by Drupal. These settings can be toggled on an off in the administrative interface on the Settings page of each theme. Specifying even one will disable Drupal’s defaults and use yours.

# features[] = logo

# features[] = favicon

# features[] = name

# features[] = slogan

# features[] = node\_user\_picture

# features[] = comment\_user\_picture

# features[] = comment\_user\_verification

# features[] = main\_menu

# features[] = secondary\_menu

# PHP

# Drupal 7 supports PHP version 5.2.5, and by default, so does your theme. This is something you will probably never need, but in case your theme has code that only works with a certain version PHP, you may specify it here. php = 5.3

# Regions

# Regions are sections of your page layout where content (blocks) are placed. Each entry is prefixed with regions and contains the system name of the region in brackets with the human readable name as the value. For example, regions[system\_name] = Human readable name. The default regions are as follows:

# regions[page\_top] = Page Top

# regions[header] = Header

# regions[highlighted] = Highlighted

# regions[help] = Help

# regions[content] = Content

# regions[sidebar\_first] = Sidebar First

# regions[sidebar\_second] = Sidebar Second

# regions[footer] = Footer

# regions[page\_bottom] = Page Bottom

# Settings

# The setting property is reserved for custom setting implementations in themes. The Garland theme provides a theme setting for the type of layout (fixed or fluid), which the site administrator can choose. While we won’t be covering custom theme settings, we highly recommend checking out the [Omega](http://drupal.org/project/omega) and [Fusion](http://drupal.org/project/fusion) themes to get an idea of how theme settings can be used. For more information, visit http://drupal.org/node/177868. settings[garland\_width] = fluid

# Screenshot

# Drupal will automatically look for a file named screenshot.png in the root of your theme directory, so this line is only required if you want to use an alternative path or file name for your theme’s screenshot. The recommended dimensions for the screenshot image are 294 x 219 pixels. screenshot = screenshot.png

# Stylesheets

# There are quite a few options for adding CSS files in Drupal 7. You’ll want to add stylesheets via the theme’s .info for CSS files you want to load on every page. I’ll cover this in much more detail in the [Managing CSS Files](http://themery.com/node/112) section in the next chapter.

# stylesheets[screen][] = path/to/screen-stylesheet.css

# stylesheets[print][] = path/to/print-stylesheet.css

# Scripts

# JavaScript files can be defined in .info files using the scripts property. Like stylesheets, you’ll only want to load JavaScript files here that need to be loaded on each page. scripts[] = path/to/script.js

# Version

# Specifying the version is discouraged for both contributed themes and modules. This is because drupal.org has a packaging script that takes care of adding the version when releases are created. However, you may use this for custom themes, if desired.  version = 7.x-1.1

# Now let’s see the basics in action by taking a look your DGD7 theme’s .info file, as shown in [Listing 15–1](http://themery.com/dgd7/theming/info-files#listing-15-1).Listing 15-1 The Top Portion of the DGD7 Theme’s .info File.

# name = DGD7 Theme

# description = A theme written for The Definitive Guide to Drupal 7 book website.

# core = 7.x

# With the exception of the core property, all of the above can be seen in the user interface on the admin/appearance page, as shown in [Figure 15–7](http://themery.com/dgd7/theming/info-files#figure-15-7). This is all you’ll need to get started with your theme.Screenshot of custom theme on Appearance pageFigure 15-7 DGD7 theme as shown on the theme listing page admin/appearance.

# WORKING WITH REGIONS

# Most of the content found on Drupal pages is output inside a region. Typical regions include the header, footer, sidebars, and content (see [Figure 15–8](http://themery.com/dgd7/theming/regions#figure-15-8)); these regions often play a large part in defining the high-level structure of your HTML markup. An option appears in the blocks interface at admin/structure/block for each region, allowing site administrators to control and position the blocks inside them.

# Screenshot of Blocks UI showing placement in regionsFigure 15-8. The Bartik theme’s regions and block placement options on the Blocks administration page.

# Themes have full control over defining and determining the placement of printing and styling regions. An example of what this looks like in the Bartik theme is shown in [Figure 15–9](http://themery.com/dgd7/theming/regions#figure-15-9).

# Screenshot of Bartik regions filled with custom blocksFigure 15-9. Bartik regions filled with custom blocks.

# In addition, themes may also use regions for less obvious purposes in combination with JavaScript or jQuery. Common use cases for regions include containing modal or hidden content to enhance the user interface or embedding blocks into node content.

# DEFAULT REGIONS

# Drupal core defines nine regions for themes to utilize programmatically by default. The code in [Listing 15–2](http://themery.com/dgd7/theming/regions/default#listing-15-2) duplicates the default core regions in .info file format. Like most theme layer implementations, the reason themes define regions is because they want to modify or add to the defaults. Until a theme defines its own regions, Drupal will use the defaults. This means that if the default regions are sufficient for your design, you will not need to define regions in your theme’s .info file.

# Listing 15-2. Drupal’s Nine Predefined Theme Regions in Chronological Order.

# regions[page\_top] = Page Top

# regions[header] = Header

# regions[highlighted] = Highlighted

# regions[help] = Help

# regions[content] = Content

# regions[sidebar\_first] = Sidebar First

# regions[sidebar\_second] = Sidebar Second

# regions[footer] = Footer

# regions[page\_bottom] = Page Bottom

# However, including this code in your theme’s .info file to begin with is a good practice. Once you define a single region in your theme, it will override core defaults, so having the full list of defaults and commenting out regions that you have disabled (instead of deleting or omitting them entirely) is a good way to keep track of what you’re doing with them. You will need some of these regions, namely the page\_top, content, and page\_bottom regions. These are required and must be printed in every Drupal theme to maintain a properly functioning site. An example of how one might organize regions in an .info file, taking defaults into account, is shown in [Listing 15–3](http://themery.com/dgd7/theming/regions/listing-15-3).

# Listing 15-3. An Example of Region Implementation in a Theme’s .info File.

# ; CORE REGIONS - DISABLED

# ;regions[highlighted] = Highlighted

# ;regions[help] = Help

# ;regions[header] = Header

# ;regions[footer] = Footer

# 

# ; CORE REGIONS - REQUIRED

# regions[page\_top] = Page Top

# regions[content] = Content

# regions[page\_bottom] = Page Bottom

# 

# ; CORE REGIONS

# regions[sidebar\_first] = Sidebar First

# regions[sidebar\_second] = Sidebar Second

# 

# ; CUSTOM REGIONS

# regions[my\_custom\_region] = My Custom Region

# TIP If you’re curious where Drupal defines the default regions, take a look at the [\_system\_rebuild\_theme\_data()](http://api.drupal.org/api/function/_system_rebuild_theme_data/7) function.

# As shown in [Figure 15–10](http://themery.com/dgd7/theming/regions/default#figure-15-10), the intended display of Drupal’s default regions is a standard three-column layout. The gray regions are required and the rest are optional. header, sidebar\_first, sidebar\_second, and footer are layout regions. The page\_top and page\_bottom are special regions; they are discussed in the “Hidden Regions” section of this chapter.

# Diagram of default regions laid out in three columnsFigure 15-10. Drupal’s default layout for regions.

# The highlighted region replaces the old Site Mission, which used to be a static variable containing the site’s mission statement or a brief summary text that was output manually in page.tpl.php. The prior implementation was not ideal for a few reasons, but mainly because its display was limited to the front page. It was decided that using a custom block to display this information was a better option, so the highlighted region was created.

# The Help region also used to be a page.tpl.php variable that printed error and status messages. The status messages are now displayed in a block called System help and the Help region was created to contain it. However, the System help block may easily be placed inside the Content region, weighted above the Main content block for the same effect.

# The Content region is new to Drupal 7. It was introduced to contain the Main page content block, which is somewhat special because it can be moved from region to region but can’t be disabled. Since the Block module is optional and the contents of the Main page content block are critical to operate a Drupal site, the contents of this block will always display via the $page['content'] variable in page.tpl.php.

# As a result, some of the Block module’s functionality doesn’t work as you might expect. If you place the Main page content block in the disabled area or set block visibility settings to exclude it from a page, the Block module’s UI will lead you to believe that it has been disabled. However, the content will still appear. You’ll also notice changes in the markup, which may lead to undesired results, such as un-styled content, depending on how your CSS is written.

# HIDDEN REGIONS

# Notably missing from the options on the Blocks administration page in [Figure 15–8](http://themery.com/dgd7/theming/regions/hidden#figure-15-8) are the page\_top and page\_bottom regions. Both are hidden regions, which Drupal intentionally excludes from the user interface so that site administrators can’t interact with or control their content. The main purpose of hidden regions is to act as a placeholder where modules or themes can dynamically add markup to in a structured way. Themes may declare hidden regions within .info files by using the following syntax, with each region on a separate a line:

# regions\_hidden[] = the\_region\_name

# Both the page\_top and page\_bottom regions are printed in html.tpl.php (see [Listing 15–4](http://themery.com/dgd7/theming/regions/hidden#listing-15-4)) and should not be removed or rearranged. The page\_top region, for example, is utilized by the Toolbar module to add the markup needed for the administrative toolbar shown at the top of each page when a user is logged in as a site administrator. The page\_bottom region exists for modules to add any final closing markup, which specifically needs to be at the very bottom of the page. An example of this is the Google Analytics module, which adds markup to load JavaScript files that track the site visitor activity and needs to be loaded last. The page\_bottom region replaces the $closure variable that was used in prior versions of Drupal.

**Listing 15-4.** The Contents of html.tpl.php, Highlighting the Placement of the page\_top and page\_bottomRegions.

1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN" "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
2. <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="<?php print $language->language; ?>" version="XHTML+RDFa 1.0" dir="<?php print $language->dir; ?>"<?php print $rdf\_namespaces; ?>>
3. <head profile="<?php print $grddl\_profile; ?>">
4. <?php print $head; ?>
5. <title><?php print $head\_title; ?></title>
6. <?php print $styles; ?>
7. <?php print $scripts; ?>
8. </head>
9. <body class="<?php print $classes; ?>" <?php print $attributes;?>>
10. <div id="skip-link"><a href="#main-content" class="element-invisible element-focusable"><?php print t('Skip to main content'); ?></a></div>
11. <?php print $page\_top; ?>
12. <?php print $page; ?>
13. <?php print $page\_bottom; ?>
14. </body>
15. </html>

**TIP** Drupal uses hook\_system\_info\_alter() to declare the page\_top and page\_bottom hidden regions. For more information, see <http://api.drupal.org/api/function/system_system_info_alter/7>.

# MODULE-SPECIFIC REGIONS

# The Dashboard module’s Dashboard Main and Dashboard Sidebar regions are an example of regions created by a module. These regions are non-traditional in the sense that they can’t be administered via the Blocks administration page, and the theme does not control defining or printing them. The Dashboard module defines them programmatically using hook\_system\_info\_alter() and takes care of displaying them on the administrative Dashboard located at /admin. The Dashboard module allows you to drag and drop available blocks to those regions to create a dashboard for site administrators (see [Figure 15–11](http://themery.com/dgd7/theming/regions/module#figure-15-11)).

# Screenshot of Dashboard module in edit mode, showing its use of custom regionsFigure 15–11. Administrative Dashboard in edit mode.

# REGIONS AND YOUR THEME

# Getting started with your theme regions requires taking a good look at design requirements as well as planning for the unexpected. There are many things to consider, including how site administrators will need to work with blocks and regions, what types of content you have, and how regions play a part in your general layout strategy. As discussed earlier, the default regions are a great starting point. We recommend that you begin defining the defaults in your theme’s .info file and tweaking from there, as shown in [Listing 15–5](http://themery.com/dgd7/theming/regions/theme#listing-15-5).

**Listing 15–5.** Drupal’s Default Regions

1. regions[page\_top] = Page Top
2. regions[header] = Header
3. regions[highlight] = Highlight
4. regions[help] = Help
5. regions[content] = Content
6. regions[sidebar\_first] = Sidebar First
7. regions[sidebar\_second] = Sidebar Second
8. regions[footer] = Footer
9. regions[page\_bottom] = Page Bottom

**TIP** In addition to defining regions in your theme’s .info file, you’ll need to print it in the appropriate template file. **The page\_top and page\_bottom regions print in the html.tpl.php template and the rest print in page.tpl.php. Printing regions and template files are discussed in more detail later in the chapter.**

# USING REGIONS VS. HARD-CODING VARIABLES IN TEMPLATE FILES

# When deciding whether or not to use regions in your theme, it’s useful to consider the content that will be included in each section, how likely the position of the content is to change, and who needs to be able to change it. Blocks are flexible by nature and were designed to allow site administrators to easily move them around. This can cause problems if blocks are expected to be in a certain region and then moved or reordered.

# When working on a site alone, or when only a few trusted individuals have control over the configuration of blocks, this is probably not something you need worry about. Alternatively, in cases where less trusted individuals have access and can potentially cause problems, taking extra measures to identify potential problem areas and doing what you can to prevent them is well worth it. For example, headers and footers are especially prone to this sort of problem. They usually have a tightly defined design and CSS to match. When blocks are moved around inside these regions, especially highly styled content such as the main menu navigation, things can go wrong quickly in the wrong hands. Sometimes defining an additional region, even if its purpose is to hold only one block, is a safer option compared with placing the block in the header region with other blocks. This will help ensure it is always printed in the right location and reduce the chance of user error. If site administrators do not need control over positioning, it may be best to print using a hardcoded variable in page.tpl.php, where it can’t be affected by actions taken in the blocks interface.

# As a general rule, consider using a region when content needs to be moved between regions or rearranged in the Blocks interface. When content doesn’t need to be controlled via the Blocks interface, and it is risky for it to be there, consider hardcoding it in template files so it can’t be affected by actions taken in the Blocks interface.

# TIP The main menu ($main\_menu) and secondary menu ($secondary\_menu), which are located in page.tpl.php, are examples of hardcoded variables.

# LAYOUT STRATEGIES

# The core defaults for sidebars (Sidebar First and Sidebar Second) were designed to handle multiple sidebar combinations with the help of body classes. Drupal is extremely flexible, and pages can be changed on a whim. Whether this will actually look good or not depends on how flexible and well coded the theme is. Since Drupal only prints regions that contain content, having a well-planned and flexible layout is very important.

# For example, let’s say you have a two-column layout theme where the first column contains the main content and the Sidebar First region contains a single block. If you were to set the visibility of that block to only show on the home page, the entire Sidebar First region would only print on the home page and the inside pages would print just the main column. If your layout CSS only accounts for having both of those columns on each page, instead of including CSS for both a single column and the two-column, your layout will break. While regions are fairly easy to add or modify at any given time, oversimplifying the layout in the beginning of a project may come back to bite you in the form of extra CSS work. However many sidebars your theme will have, it’s generally best to account for all possible sidebar combinations (one, two, or three columns) to avoid running into problems down the line. A great way to do this easily and sustainably is to use an established base theme.

# There are also certain types of content that often work better in separate regions. For example, custom blocks containing advertisements and blocks that have significantly different design requirements are often easier to work with and write CSS for when they are abstracted. [Figure 15–12](http://themery.com/dgd7/theming/regions/layout#figure-15-12) shows what adding a region for an ad banner and main navigation might look like.

# It is also important to consider how the pages will be built and who will be working with them. If your site is going to be using regions and blocks to implement more complicated designs and you want to make it easy for site administrators to use, it may make sense to predefine multiple regions to lay out smaller sections of your pages. A good example of this is the Bartik theme, which contains seven additional regions to organize blocks in the footer, as shown in [Figure 15–13](http://themery.com/dgd7/theming/regions/layout#figure-15-13). The same look could be achieved by defining two regions (Footer First and Footer Second) instead and style them using CSS to float the blocks in each to the left, but Bartik’s implementation, shown in [Listing 15–6](http://themery.com/dgd7/theming/regions/layout#listing-15-6) and illustrated in [Figure 15–13](http://themery.com/dgd7/theming/regions/layout#figure-15-13), is arguably easier to understand for those who are not interested in the inner-workings of the code and just want to use the theme.

# Illustration of custom regionsFigure 15–12. Example of custom advertisement banner and navigation regions.Figure 15–6. Excerpt from Bartik Theme’s .info File Where Its Seven Custom Regions Are Defined.

# regions[triptych\_first] = Triptych first

# regions[triptych\_middle] = Triptych middle

# regions[triptych\_last] = Triptych last

# regions[footer\_firstcolumn] = Footer first column

# regions[footer\_secondcolumn] = Footer second column

# regions[footer\_thirdcolumn] = Footer third column

# regions[footer\_fourthcolumn] = Footer fourth column

# Screenshot of Bartik's Footer regions filled with content Figure 15–13. Populated footer regions in the Bartik theme.

## CREATING NEW REGIONS

# The creation of a new region is a two-step process. Using the example in [Figure 15–12](http://themery.com/dgd7/theming/regions/exercise#figure-15-12), here’s the process of creating the new Banner Ad and Navigation regions.

# Define regions in the dgd7.info file. Begin by adding the code for your new regions to the defaults you started with in [Listing 15–3](http://themery.com/dgd7/theming/regions/exercise#listing-15-3), plus the definition of the banner\_ad and navigation regions to your dgd7.info file.

1. ; DEFAULT REGIONS
2. regions[page\_top] = Page Top
3. regions[header] = Header
4. regions[highlight] = Highlight
5. regions[help] = Help
6. regions[content] = Content
7. regions[sidebar\_first] = Sidebar First
8. regions[sidebar\_second] = Sidebar Second
9. regions[footer] = Footer
10. regions[page\_bottom] = Page Bottom
12. ; CUSTOM REGIONS
13. regions[banner\_ad] = Banner Ad
14. regions[navigation] = Navigation

# Print the regions in the page.tpl.php template file. Once you clear your site caches, you’ll be able to see and populate the new regions on the Blocks administration page at admin/structure/block. In order to get them to display on the page, you’ll need to override the page.tpl.php file in your theme and print the new regions.

# Navigate to the modules/system directory, copy the page.tpl.php file and paste it into the sites/all/themes/dgd7 directory you created earlier.

# Open the page.tpl.php file in the theme and scroll down to the <div id="page-wrapper">and paste the code to print the region below it, and above the <div id="header">.

1. <div id="page-wrapper"><div id="page">
2. <?php print render($page['banner\_ad']); ?>
3. <div id="header"><div class="section clearfix">

# Remove the default markup for the $main\_menu and replace it with the region code for your new navigation region.

# Remove this code:

1. <?php if ($main\_menu || $secondary\_menu): ?> <div id="navigation"><div class="section">
2. <?php print theme('links\_\_system\_main\_menu', array('links' => $main\_menu, 'attributes' => array('id' => 'main-menu', 'class' => array('links', 'inline', 'clearfix')), 'heading' => t('Main menu'))); ?>
3. <?php print theme('links\_\_system\_secondary\_menu', array('links' => $secondary\_menu, 'attributes' => array('id' => 'secondary-menu', 'class' => array('links', 'inline', 'clearfix')), 'heading' => t('Secondary menu'))); ?>
4. </div></div> <!-- /.section, /#navigation --> <?php endif; ?>

# Replace with this code:

# <?php print render($page['navigation']); ?>

# Technically you’re finished, but let’s add some content to illustrate what you’ve done.

# Add a new custom block for the Banner Ad code. Title the block “Banner Ad” and add the following code to fake the appearance of an ad banner in the Block body (be sure to select the Full HTML text format). Then, select the Banner Ad region for the region settings and save it.

# <img style="width: 728px; height: 90px; border: solid 1px #000;" alt="728 x 90 Banner Ad" src="image.png" />

# Go back to the admin/structure/block page. Find the Main Menu block and place it inside the Navigation region and click Save blocks.

# You’ve just added and populated two new custom regions!

# TEMPLATE FILES

# Drupal core, its modules, and contributed modules provide much of their output in the form of template files. Template files consist of HTML markup and PHP variables. This makes it fairly easy for those with little or no PHP experience to make changes to HTML code.

# A simple example of a template file is user-picture.tpl.php (see [Listing 15–7](http://themery.com/dgd7/theming/templates#listing-15-7)). This template is located in the modules/user directory and its purpose is solely to print a site user’s picture as either an image or an image with a link (depending on whether or not the user viewing the photo has access to view user profiles). It wraps the picture in a <divclass="user-picture">. This template file will be used anywhere the user\_picture theme hook is called, such as the user profile page and author information for nodes and comment (where enabled).

# Listing 15–7. Contents of user-picture.tpl.php file.

# <?php

# /\*\*

# \* @file

# \* Default theme implementation to present a picture configured for the

# \* user's account.

# \*

# \* Available variables:

# \* - $user\_picture: Image set by the user or the site's default. Will be linked

# \* depending on the viewer's permission to view the users profile page.

# \* - $account: Array of account information. Potentially unsafe. Be sure to

# \* check\_plain() before use.

# \*

# \* @see template\_preprocess\_user\_picture()

# \*/

# ?>

# <?php if ($user\_picture): ?>

# <div class="user-picture">

# <?php print $user\_picture; ?>

# </div>

# <?php endif; ?>

# A typical page on a Drupal site is essentially a big tree of nested template files and theme functions. As [Figure 15–14](http://themery.com/dgd7/theming/figure-15-14) illustrates, this tree begins with larger templates such as html.tpl.php and page.tpl.php files and goes all the way down field.tpl.php, which is used to print fields.

# An illustrated screenshot of the Bartik theme which exposes the use of the major templatesFigure 15–14. An example home page using the Bartik theme, which highlights the use of major template files and many custom regions

# COMMON CORE TEMPLATES

# Drupal core contains over forty template files, but there are six major template files (described in [Table 15–1](http://themery.com/dgd7/theming/templates/core#table-15-1)) that are tasked with making up the majority of each page. These major template files are the ones you’ll be working with most when writing Drupal themes and they will allow you do most of the heavy lifting in your theme.

# Table 15–1. Common Core Template Files

| NAME | ORIGIN | PURPOSE |
| --- | --- | --- |
| html.tpl.php | modules/system | Prints the structure of the HTML document, including the contents of <head> tags, e.g. $scripts, and $styles, as well as opening and closing <body>tags with $page\_top, $page and $page\_bottom regions printed inside. Unless you need to change the DOCTYPE, there’s probably no reason to override this file. |
| page.tpl.php | modules/system | Prints the page level regions and other hard-coded variables such as $logo, $site\_name, $tabs, $main\_menu, etc. Full control of the site layout is possible by manipulating this file, and most themes provide their own version of it. |
| region.tpl.php | modules/system | Prints the HTML markup for regions. |
| block.tpl.php | modules/block | Prints the HTML markup for blocks. |
| node.tpl.php | modules/node | Prints the HTML markup for nodes. |
| comment.tpl.php | modules/comment | Prints the HTML markup for comments. |
| field.tpl.php | modules/field/theme\* | Prints the HTML markup for fields. There are many different types of fields, and since this file needs to cover every case, its implementation is very general. If having semantic markup is important to you, you’ll probably end up with a few versions of this template. |

# \* field.tpl.php is used only when overridden by a theme. The one in modules/field/theme is only provided as a base for your work.

# OVERRIDING TEMPLATE FILES

# The template files provided by Drupal core and contributed modules represent the default markup implementation chosen by the original author or team, but every last one of these template files—and the markup and variables printed inside of them—is customizable. When developing a theme, if you decide the default implementation is not going to suit your needs, you can simply choose to override it. Drupal’s theme layer is designed to be extremely flexible and easy to manipulate in this way.

# The beauty of theming Drupal sites is that you can easily make changes without having to modify templates where they originate. The process of overriding template files is extremely simple:

# Find the original template file by browsing through code or checking http://api.drupal.org

# Copy and paste it into your theme directory.

# Clear the site cache and reload!

# After following these three steps, Drupal will begin using the theme’s version of the file, and you are free to make whatever changes you wish. It’s that simple.

# TIP A quick way to ensure that Drupal is using the template file you’ve just overridden in your theme is to add text to the top of the template file, like “Hello World.” If your text appears when you reload, you’ll know you’re working with the correct file.

# GLOBAL TEMPLATE VARIABLES

Template files usually contain a few more variables than they actually print. In some cases there are many more. This is a great thing for theme developers because it opens up many possibilities for manipulating the display of markup without the need for much PHP knowledge. [Table 15-2](http://themery.com/dgd7/theming/templates/global-variables#table-15-2) describes some of the helpful variables available in all templates (with the exception of the attribute variables; these are covered section the “HTML Attributes” section). Identifying available variables is covered in detail in the next chapter.

**Table 15-2.** Variables available in all templates

| VARIABLE | DESCRIPTION |
| --- | --- |
| $is\_admin | Helper variable that equals TRUE if the currently logged in user is an administrator, and FALSE otherwise. |
| $logged\_in | Helper variable that equals TRUE if the current user is logged in, and FALSE otherwise. The $user->uid is used to determine this information, as anonymous users always have a user ID of 0. |
| $is\_front | Helper variable that uses the drupal\_is\_front\_page() function to determine if the current page is the front page of the site. Equals TRUE on the front page (unless the database is offline), and FALSEotherwise. |
| $directory | The directory in which the template being used is located. |
| $user | An object that contains account information of the currently logged in user. **It may be accessed by adding the line global $user; to the template you are working in. Never print any properties of it directly because it contains raw user data and thus it is insecure. Instead, use theme\_username(); for example, theme('username', array('account' => $user)).** |
| $language | An object that contains information about the language currently being used on the site, such as $language->dir, which contains the text direction, and $language->language which would contain en for English. It may be accessed by adding the line global $language; to the template you are working in. |
| $theme\_hook\_suggestions | An array containing other possible theme hooks, which can be used as variants for naming template files and theme functions or to determine context. See the “Theme Hook Suggestions” section. |
| $title\_prefix and $title\_suffix | Render arrays containing elements, such as contextual links, to be printed before and after the title in templates or at the top and bottom of template files where a title does not exist. |

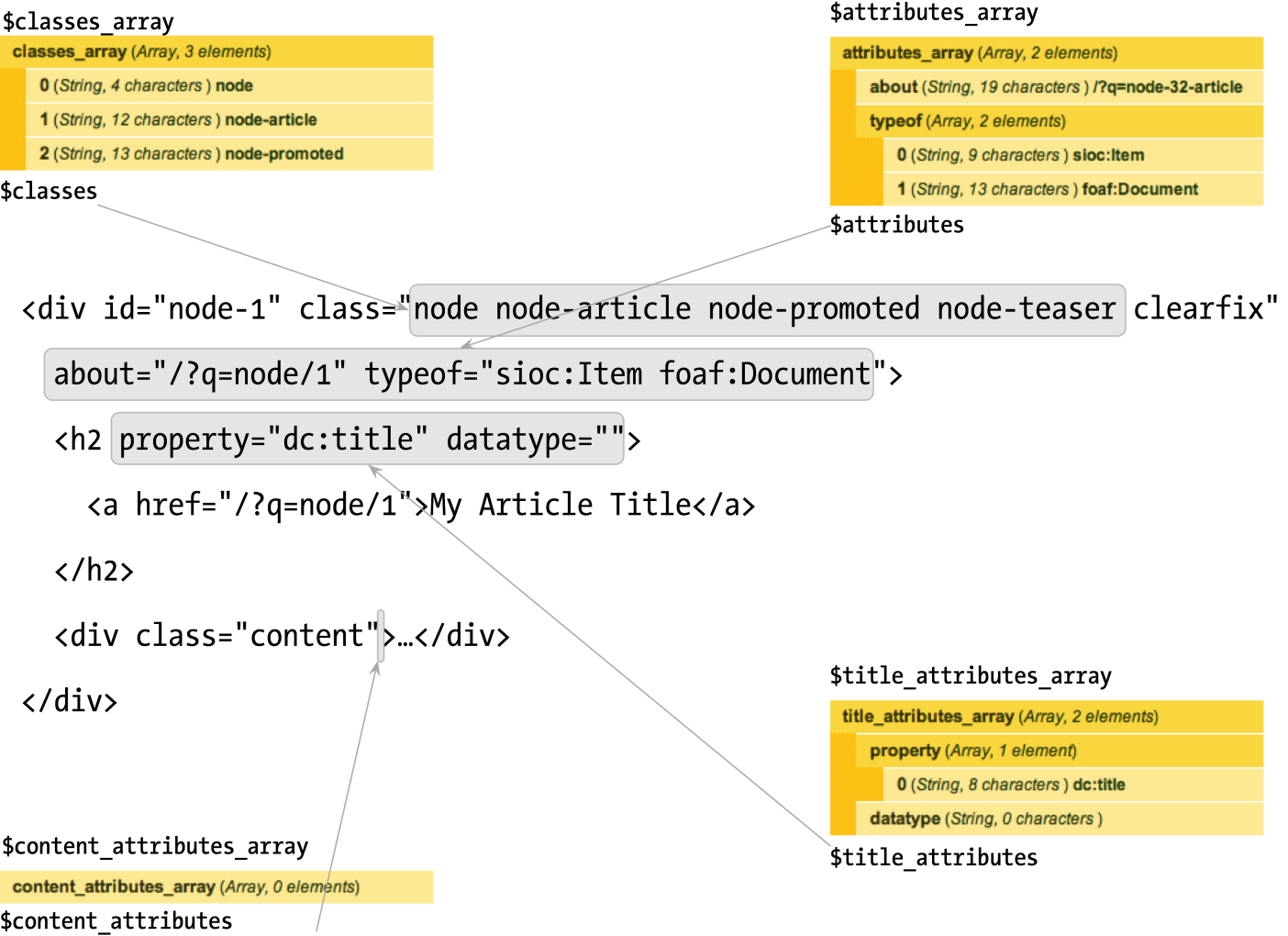
## HTML ATTRIBUTES

In Drupal 7, we began storing attributes in arrays. Part of the reason this was done is the RDF module. The RDF module utilizes these variables to tack on its data during the preprocess phase. Another reason was to allow theme developers more control over the classes printing out in their template files in preprocess functions.

**Each of these variables, described in**[**Table 15–3**](http://themery.com/dgd7/theming/templates/global-variables#table-15-3)**, has an array and string version. The array version, which contains the suffix \_array in the variable name, is populated during various preprocess functions, such as template\_preprocess() and template\_preprocess\_node() or template\_preprocess\_block(). Then, during the template\_process() phase, new variables containing a flattened or string version of these arrays is created for use in templates.** This process is illustrated in [Figure 15–15](http://themery.com/dgd7/theming/templates/global-variables#figure-15-15). See the “Preprocess and Process Functions” section of this chapter for more details.

**Table 15–3.** Pluggable HTML Attributes

| VARIABLE | DESCRIPTION |
| --- | --- |
| $attributes | Contains HTML attributes provided by modules (mainly RDF), except for the class attribute, which is handled separately (see below). $attributes, available as $attributes\_array in preprocess, is usually reserved for the top-level HTML wrapper element, such as <body> or outermost <div> in other template files. |
| $classes | Contains HTML classes for templates. Usually reserved for the top-level HTML wrapper element, such as <body> or outermost <div> in other template files. |
| $title\_attributes | Contains classes for the top-level heading, such as a node or block title, of the template file, which is usually an <h2> for node teaser or block content. |
| $content\_attributes | Contains classes for the content wrapper <div>, or post body of templates. An example of how these variables are used can be found in the node.tpl.php file. |



**Figure 15–15.** Excerpt from node.tpl.php, which highlights how the pluggable HTML attributes are used

**TIP** **If you don’t see these attributes in your source code, be sure to enable the RDF module.**

All of the common core templates provide detailed documentation of the available variables. A quick look at the default block.tpl.php template file, located in the modules/block directory reveals that most of the contents of the file is actually documentation for the available variables. As shown in [Listing 15–8](http://themery.com/dgd7/theming/templates/global-variables#listing-15-8), you can get a good idea of what you have to work with by just looking at the documentation and code.

**Listing 15–8.** Source Code for Default modules/block/block.tpl.php, Including Variable Documentation

/\*\*

\* @file

\* Default theme implementation to display a block.

\*

\* Available variables:

\* - $block->subject: Block title.

\* - $content: Block content.

\* - $block->module: Module that generated the block.

\* - $block->delta: An ID for the block, unique within each module.

\* - $block->region: The block region embedding the current block.

\* - $classes: String of classes that can be used to style contextually through

\* CSS. It can be manipulated through the variable $classes\_array from

\* preprocess functions. The default values can be one or more of the

\* following:

\* - block: The current template type, i.e., "theming hook".

\* - block-[module]: The module generating the block. For example, the user

\* module is responsible for handling the default user navigation block. In

\* that case the class would be 'block-user'.

\* - $title\_prefix (array): An array containing additional output populated by

\* modules, intended to be displayed in front of the main title tag that

\* appears in the template.

\* - $title\_suffix (array): An array containing additional output populated by

\* modules, intended to be displayed after the main title tag that appears in

\* the template.

\*

\* Helper variables:

\* - $classes\_array: Array of html class attribute values. It is flattened

\* into a string within the variable $classes.

\* - $block\_zebra: Outputs 'odd' and 'even' dependent on each block region.

\* - $zebra: Same output as $block\_zebra but independent of any block region.

\* - $block\_id: Counter dependent on each block region.

\* - $id: Same output as $block\_id but independent of any block region.

\* - $is\_front: Flags true when presented in the front page.

\* - $logged\_in: Flags true when the current user is a logged-in member.

\* - $is\_admin: Flags true when the current user is an administrator.

\* - $block\_html\_id: A valid HTML ID and guaranteed unique.

\*

\* @see template\_preprocess()

\* @see template\_preprocess\_block()

\* @see template\_process()

\*

\* @ingroup themeable

\*/

?>

<div id="<?php print $block\_html\_id; ?>" class="<?php print $classes; ?>"<?php print $attributes; ?>>

<?php print render($title\_prefix); ?>

<?php if ($block->subject): ?>

<h2<?php print $title\_attributes; ?>><?php print $block->subject ?></h2>

<?php endif;?>

<?php print render($title\_suffix); ?>

<div class="content"<?php print $content\_attributes; ?>>

<?php print $content ?>

</div>

</div>

At the top of the file there is a @file block, which briefly describes the purpose of the file. Underneath, there is a long list of variables, some of which are printed in the template file and some that are not. There are also @see references to applicable preprocess and process functions, which are discussed in more detail in the next chapter.

To get an up-close idea of what this template file produces, take a look at a block produced by the Bartik theme. Bartik does not include a block.tpl.php file; it uses Drupal’s default, which is provided by the Block module. Create a custom block with the title “My Custom Block” and some dummy text as the body, and place it in the Sidebar First region of the Bartik theme.



**Figure 15–16.** Screenshots of our rendered custom block as viewed using the Bartik theme and the configuration page for the block

Your custom block, shown in [Figure 15–16](http://themery.com/dgd7/theming/templates/global-variables#figure-15-16) along with the block.tpl.php template file in [Listing 15–8](http://themery.com/dgd7/theming/templates/global-variables#listing-15-8), produces the output displayed in [Listing 15–9](http://themery.com/dgd7/theming/templates/global-variables#listing-15-9) for anonymous users. **The block title is printed by <?php print $block->subject ?>** and the **body is printed by <?php print $content ?>**. Drupal will only populate variables and display content that the user viewing it has access to.

**Listing 15–9.** HTML output of a custom block titled “My Custom Block” when logged out.

1. <div id="block-block-1" class="block block-block">
2. <h2>My Custom Block</h2>
3. <div class="content">
4. <p>Enim quam iusto quam iis enim. Molestie at et diam ut legere. Feugiat tation facilisis quarta soluta quam. Facilisis lectorum modo nam modo suscipit.</p>
5. </div>
6. </div>

[Listing 15–10](http://themery.com/dgd7/theming/templates/global-variables#listing-15-10) shows the HTML for the same block as it is displayed to users logged in as administrators. You’ll notice that the code is different. **Administrators have access to contextual administrative links, added by the Contextual Links module. These links are printed via the <?php print render($title\_prefix); ?> line**. The Contextual Links module also adds a class to the wrapper <div> identifying it as a contextual-links-region. This behavior is not specific to the Block module or the block.tpl.php template file. **The $title\_prefix and $title\_suffix variables were created to allow modules to inject content before and after titles in template files, which the Contextual links module takes advantage of.**

**Listing 15–10.** HTML Output of a Custom Block Titled “My Custom Block” When Logged In as an Administrator, Highlighting the Output of $title\_suffix.

1. <div id="block-block-1" class="block block-block contextual-links-region">
2. <h2>My Custom Block</h2>
3. <div class="contextual-links-wrapper contextual-links-processed">
4. <a class="contextual-links-trigger" href="#">Configure</a>
5. <ul class="contextual-links">
6. <li class="block-configure first last"><a href="/admin/structure/block/manage/block/1/configure?destination=node">Configure block</a></li>
7. </ul>
8. </div>
9. <div class="content">
10. <p>Enim quam iusto quam iis enim. Molestie at et diam ut legere. Feugiat tation facilisis quarta soluta quam. Facilisis lectorum modo nam modo suscipit.</p>
11. </div>
12. </div>

# THEME FUNCTIONS

# The purpose of a theme function is the same as a template file in that its goal is to provide HTML markup in a way that makes it customizable by themes (and modules, too). There are many, many theme functions in Drupal core, from form elements to menu items to full administration page implementations. For a full list of theme functions available in Drupal 7, visit <http://api.drupal.org/api/group/themeable/7>.

# HOW THEME FUNCTIONS ARE CREATED

# Drupal core and modules usually define theme functions, but they can be defined by themes as well. hook\_theme() implementations are where all the juicy information about most generic theme functions resides, including what parameters these functions accept. Theme hooks are covered in detail in the “Theme Hook Suggestions” section later in this chapter, but [Listing 15–11](http://themery.com/dgd7/theming/theme-functions/create#listing-15-11) shows what a simple hook\_theme() implementation looks like.

# Listing 15–11. Example hook\_theme() implementation.

# <?php

# /\*\*

# \* Implements hook\_theme().

# \*/

# function mymodule\_theme() {

# return array(

# 'my\_theme\_hook' => array(

# 'variables' => array('parameter' => NULL),

# ),

# );

# }

# Implementations of hook\_theme() let Drupal know about theme hooks. Once Drupal is aware, it will search for a theme function called theme\_my\_theme\_hook() in this case, which might look like the code in [Listing 15–12](http://themery.com/dgd7/theming/theme-functions/create#listing-15-12).

# Listing 15–12. Example Theme Function Implementation

# <?php

# function theme\_my\_theme\_hook($variables) {

# $parameter = $variables['parameter'];

# if (!empty($parameter)) {

# return '<div class="my-theme-hook">' . $parameter . '</div>';

# }

# }

# CALLING THEME FUNCTIONS

# Throughout this chapter we refer to theme functions as theme\_this() and theme\_that(). That’s what the functions are named and usually referred to as. However, you should never call a theme function directly. Doing so will reverse the wonderful functionality that comes along with Drupal’s theme layer, such as overrides, suggestions, etc. Always use the theme() function to generate theme output. It takes care of routing the request to the appropriate theme function. For more information on how this works, see http://api.drupal.org/api/function/theme/7.

# Using theme\_image(), [Listings 15–13](http://themery.com/dgd7/theming/theme-functions/call#listing-15-13) and [15–14](http://themery.com/dgd7/theming/theme-functions/call#listing-15-14) illustrate the right and wrong way to call theme functions, respectively.

# Listing 15–13. The Correct Way to Call a Theme Function.

# <?php print theme('image', array('path' => 'path/to/image.png', 'alt' => 'Image description')); ?>

# Listing 15–14. The Wrong Way to Call a Theme Function

# <?php print theme\_image(array('path' => 'path/to/image.png', 'alt' => 'Image description')); ?>

# OVERRIDING THEME FUNCTIONS

# Overriding theme functions is very similar to overriding template files. The main difference is that you are working with functions, and your overridden theme functions all reside in template.php. The steps involved in overriding a theme function are as follows:

# Find the original theme function by browsing through Drupal’s source code or checking http://api.drupal.org.

# Copy and paste it into your template.php file.

# Change the beginning of the function name from theme\_ to yourthemename\_.

# Save template.php, clear the site cache, and reload!

# Caution If creating template.php from scratch, remember to include <?php at the top of the file. Also note that a closing tag should not be added at the bottom of the file. Omitting the closing PHP tag prevents unwanted whitespace, which can cause “Cannot modify header information” or “Headers already sent” errors. For more information, visit <http://drupal.org/node/1424>.

# EXERCISE

## LET'S OVERRIDE A THEME FUNCTION

# Here is a theme function called theme\_more\_link(). It is used to print a link to additional content in blocks. To find the code for the theme function, take a look at http://api.drupal.org/api/function/theme\_more\_link/7.

# Copy and paste the original theme function code into template.php.

# <?php

# /\*\*

# \* Returns HTML for a "more" link, like those used in blocks.

# \*

# \* @param $variables

# \* An associative array containing:

# \* - url: The url of the main page.

# \* - title: A descriptive verb for the link, like 'Read more'.

# \*/

# function theme\_more\_link($variables) {

# return '<div class="more-link">' . l(t('More'), $variables['url'], array('attributes' => array('title' => $variables['title']))) . '</div>';

# }

# Change the beginning of the function name to your theme’s name, save it, and clear the site cache.

# <?php

# /\*\*

# \* Returns HTML for a "more" link, like those used in blocks.

# \*

# \* @param $variables

# \* An associative array containing:

# \* - url: The url of the main page.

# \* - title: A descriptive verb for the link, like 'Read more'.

# \*/

# function dgd7\_more\_link($variables) {

# return '<div class="more-link">' . l(t('More'), $variables['url'], array('attributes' => array('title' => $variables['title']))) . '</div>';

# }

# Drupal will now use your version of the theme function, so make changes!

# <?php

# /\*\*

# \* Overrides theme\_more\_link().

# \* - Changed the text from "More" to "Show me More"

# \* - Changed the class from "more-link" to "more"

# \*/

# function dgd7\_more\_link($variables) {

# return '<div class="more">' . l(t('Show me MORE!'), $variables['url'], array('attributes' => array('title' => $variables['title']))) . '</div>';

# }

# TIP In Step 3, you’ll notice that the comment block has been changed to indicate what function was overridden and the changes that were made. Documenting your code is always a good idea, and explicitly listing the reasons why you’ve overridden a theme function can be a big time saver in the future. Theme functions change, and some aren’t as small as a few lines. When upgrading major versions of Drupal, such as Drupal 7 to Drupal 8, such comments will make your life a lot easier.

# THEME HOOKS AND THEME HOOK SUGGESTIONS

# Theme functions and templates are defined by theme hooks. By making use of theme hook suggestions, you have a lot more flexibility to override theme functions or templates in certain situations. This section covers both ways to greatly increase the power and manoeuvrability of your custom theme.

# WHAT IS A THEME HOOK?

# In Drupal, theme hooks refer to template files and functions that have been specifically registered via hook\_theme(). This may sound scary or over-technical to non-PHP developers, but honestly it’s not. You’ve already learned about template files and theme functions, so technically you already have a pretty good grasp on theme hooks.

# Whether a template file or function is implemented in core is decided on a case-by-case basis, and the criteria for making this decision is usually a balance between how likely it is to be reused by other modules, how often it is expected to change, and whether or not it makes sense for performance reasons. Template files are slightly slower than theme functions so they are not always desirable. Smaller bits of markup for things like form input elements are more efficiently implemented as theme functions, whereas larger chunks like nodes and blocks are better as a template file.

# Both theme functions and template files exist as a way for Drupal and its modules to create output consisting of markup and variables in a way that you, the themer, can override and make it your own. They are both entirely YOUR domain, and you get the last word as to how they should look.

# Both share the same exact theme hook. For example, a template file called node.tpl.php and a function called theme\_node() share the same node theme hook. The difference is in the implementation, as both cannot be used at the same time.

# Both can take advantage of preprocess functions, which allow you to intercept and alter variables before rendering. Using the node hook as an example, this would look like template\_preprocess\_node(); in your theme it would be yourtheme\_preprocess\_node().

# THEME HOOK SUGGESTIONS

# The default implementation of template files and theme functions offer a very generic set of markup that is sufficient, but not ideal in all cases. When doing a standard override, such as copying block.tpl.php into a theme, the changes made will apply site-wide whenever a block is rendered. At times this the desired result, but you’ll often want to make changes to a specific block, a set of blocks provided by a specific module, or even a group of blocks in a specific region.

# Theme hook suggestions allow you to implement targeted overrides in your theme for both template files and theme functions with naming patterns. The options and naming patterns vary depending on what type of object you are working with. During the preprocess stage, before each template is rendered, a variable called $theme\_hook\_suggestions is created and populated with alternative hook suggestions.

# SUGGESTIONS AND TEMPLATE FILES

# All of the common template files listed in [Table 15–1](http://themery.com/node/23#table-15-1) can be overridden to allow for more targeted customization by simply changing the name of the template file. When working with blocks, for example, Drupal suggests the options in [Listing 15–15](http://themery.com/dgd7/theming/hooks-suggestions/templates#listing-15-15) during template\_preprocess\_block().

# Listing 15–15. Excerpt from template\_preprocess\_block() where template suggestions for block template files are defined.

# <?php

# $variables['theme\_hook\_suggestions'][] = 'block\_\_' . $variables['block']->region;

# $variables['theme\_hook\_suggestions'][] = 'block\_\_' . $variables['block']->module;

# $variables['theme\_hook\_suggestions'][] = 'block\_\_' . $variables['block']->module . '\_\_' . $variables['block']->delta;

# Drupal automatically converts the underscores to dashes and searches for these templates in your theme when determining which one to use. This code translates to the suggestions shown in [Table 15–4](http://themery.com/dgd7/theming/hooks-suggestions/templates#table-15-4).

# Table 15–4. Template Suggestions for Blocks

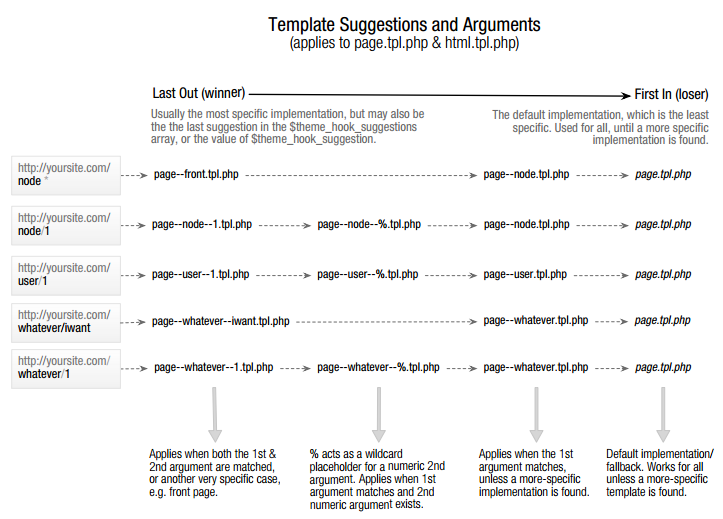
| SUGGESTION | TEMPLATE FILE EQUIVALENT | DESCRIPTION |
| --- | --- | --- |
| block | block.tpl.php | Default block implementation. |
| block\_\_REGION | block--REGION.tpl.php | REGION is replaced with the theme region name, and the template targets blocks in that region. |
| block\_\_MODULE | block--MODULE.tpl.php | MODULE is replaced with the name of the module that created the block. For example, a template file that targets custom blocks would be block--block.tpl.php and a block created by the menu module would be targeted by using block--menu.tpl.php. |
| block\_\_MODULE\_\_DELTA | block--MODULE--DELTA.tpl.php | The DELTA value, which used to be a number in previous versions, is the system name of the block as defined by the module. For example, to target the System module’s Navigation block, you would use block--system--navigation.tpl.php. In this example, “system” is the module and “navigation” is the delta. |

## PAGE-LEVEL SUGGESTIONS

# Because of their special nature as the highest-level template files in Drupal, both html.tpl.php and page.tpl.php are given special attention when it comes to generating their suggestions. A function called theme\_get\_suggestions() is used to automatically generate suggestions using arguments based on the context of the current page. This means that if you wanted to, you could literally have a different version of these template files for every page on your site. Of course, this is something you should never even think about doing, but in certain cases, like a very different home page or landing page, having a different page.tpl.php makes perfect sense.

# As mentioned, the theme hook suggestions for these files are generated with the help of arguments. Arguments in Drupal are the elements or pieces of system path of a page. For example, when viewing the URL http://yoursite.com/node/1, the first argument is “node” and the second argument is “1.” Understanding arguments in Drupal is one of the key things that will help you understand Drupal. They are extremely useful in determining context and can allow you to perform more advanced manipulations in your theme.

# [Figure 15–17](http://themery.com/dgd7/theming/hooks-suggestions/templates#figure-15-17) illustrates how you can use theme hook suggestions and arguments to make separate page.tpl.php and html.tpl.php templates for just about any page on your site.



# Figure 15–17. Suggestions for page.tpl.php on different types of pages. Drupal’s front page is set to “node” by default under Admin » Configuration » Site Information. This page is not a typical node. It is a custom page provided by the node module’s node\_page\_default() function. It lists posts that have been marked as “Promote to front page.” The “front” suggestion is specific to the front page (or home page), regardless of what type of page it is. Should you change your front page to a different path, additional suggestions will become available to you.

# Caution [Figure 15–17](http://themery.com/dgd7/theming/hooks-suggestions/templates#figure-15-17) lists examples of named paths that you’ll likely encounter when using contributed modules such as views and panels. These become system paths and can be used as template suggestions. However, attempting to create a template file using a path that was created using a custom alias (or the Pathauto module) such as about/team for a node/1 will not work. The same applies to taxonomy, terms, and user profiles. The real system path is always required when working with templates.

# Some observations of $theme\_hook\_suggestions include:

# Underscores are used instead of dashes.

# File extensions are not present because these hooks can be implemented as theme functions or template files. At this stage in the process, it doesn’t matter whether a template or a theme function will be used. When it’s time to render the content, theme() will determine which should be used and make the necessary adjustments.

# Each suggestion begins with a hook\_\_ (double-underscore) prefix. In the example shown in [Listing 15–15](http://themery.com/dgd7/theming/hooks-suggestions/templates#listing-15-15), that hook is block. This allows Drupal to fall back on the generic theme hook, which in this case is block, and use block.tpl.php when a more specific template, like block--module.tpl.php, doesn’t exist.

# The order in which these suggestions appear in the $theme\_hook\_suggestions variable determines which hook/template file will be used in FILO (first in, last out) order. When it comes time to render the template, the last suggestion will be used, with one exception. A variable called $theme\_hook\_suggestions (note that it is singular, not plural) is also available. If it’s set by a module or theme, it will take precedence over anything defined in $theme\_hook\_suggestions.

# TIP Use the dpm() function (provided with the Devel module) inside the generic template file you are working with to find out what options are available. <?phpdpm($theme\_hook\_suggestions); ?> will show the options that are available for the page you are working on.

# SUMMARY

# This chapter has covered the basics of Drupal themes, including how to:

# Define .info files and work with regions.

# Override and create targeted template files and theme functions.

# Make sense out of theme hooks and suggestions.

# Armed with this knowledge, it’s time to move onto some more advanced theme topics in the next chapter.

# ADVANCED THEMING

# One of the best things about Drupal’s theme layer is the sheer amount of flexibility it provides. In the previous chapter you learned the basics of creating a theme: working with .info files, template files, and theme functions. When implementing more custom themes, sometimes these tools alone are not enough and you need to dig deeper. This is the point where the line between front-end developer and back-end developer gets a little blurry, but stay with us.

# By the time you finish reading this chapter you’ll know how to work with variables in preprocess functions, customize forms, and use the new render API. I’ll also cover the ins and outs of working with CSS files and the basics of subtheming, and leave you with basics rules for creating sustainable Drupal themes. You’ll be transformed into a theming ninja in no time.

# FINDING AVAILABLE VARIABLES IN THE THEME LAYER

# When working in the theme layer, you’ll find that the variables are different depending on the type of entity with which you are working. You’ll also find that the various templates and theme functions don’t use or document all of the variables that are available, so one of the things you’ll often need to do is print the contents of arrays to the screen.

# There are various ways to print arrays using PHP. One of the most common ways is to use the print\_r() function. There’s also var\_dump(), get\_defined\_vars() and Drupal’s own debug(). These functions are great for small arrays, but Drupal’s arrays are known for being tremendous, thus using these functions while coding the front end of a site is annoying, to say the least. Luckily, thanks to the [Devel](http://drupal.org/project/devel) module and the Krumo library, printing compact and easily readable arrays is a piece of cake. Upon installing the Devel module, you’ll have access to functions like dpm() and kpr() among others.

# When working with templates and preprocess functions, you’ll usually print $variables using dsm() or dpm(). As an example, try adding <?php dpm($variables); ?> to the top of your node.tpl.php file.

# Screenshot of dpm() outputFigure 16–1. The result of printing <?php dpm($variables); ?> in node.tpl.php

# In [Figure 16–1](http://themery.com/dgd7/advanced-theming/finding-variables#figure-16-1), you see the result of printing the contents of the $variables array using the dpm() function. What’s nice about using dpm() is that the array is neatly printed using the $messages variable in page.tpl.php, which is where system status messages are located. As shown in [Figure 16–2](http://themery.com/dgd7/advanced-theming/finding-variables#figure-16-2), you can click the heading and expand the contents of each part one by one.

# Screenshot of expanded dpm() output

# Figure 16–2. An expanded array printed using the dpm() function

# When working inside template files, these variables are made available as top-level variables. This is done as a convenience for theme developers. For example, instead of printing $variables['status'], just print $status in templates. When working inside functions, such as theme functions or preprocess functions, use $variables['status'].

# USING THE THEME DEVELOPER MODULE

# Of course, when you’re first starting out with Drupal, you’ll need to get an idea of where the code is located and what you need to override in the first place. The [Theme Developer](http://drupal.org/project/devel_themer) module is the perfect tool to help you figure this out. Once enabled, a checkbox will appear in the bottom right corner of the page. When clicked, a semi-transparent, resizable, and draggable window appears in the top right corner of the page. You can then move it around and click on any element of the page and the window will populate with all the information you need to know—and more (see [Figure 16-3](http://themery.com/dgd7/advanced-theming/finding-variables/devel-themer#figure-16-3)).

# For example, when clicking a node, the following information is made available in the window:

# The parent functions and templates that affect the element

# The template or theme hook suggestions (candidates)

# The preprocess and process functions being used

# A printout of the variables available

# Screenshot of Theme Developer module Figure 16–3. The Theme Developer window shows theme-related information about the element that was clicked (a node in this case).

# PREPROCESS AND PROCESS FUNCTIONS

# Preprocess functions are a theme developer’s best friend. There are so many use cases where preprocess functions can make your life easier, your code more efficient, and your template files clean and crisp. If you haven’t used them before, either because you think you don’t need them or are afraid of delving too deep into PHP, you are truly missing out. We hope to change that.

# By now you are familiar with the general purpose of template files, which is mainly to provide markup and print variables. But what if you’d like to change those variables or add your own? Your first inclination might be to create a template file and do everything there, but that is often the wrong way to go.

# Preprocess functions were designed for this exact purpose. When implementing a preprocess or process function you are basically telling Drupal, “Hey, wait! I have some changes to make to this data before you send it off for rendering.” It’s sort of like an editor getting a final review of an article before it’s allowed to be published. By definition, “preprocess” is a phase of processing that happens before templates are rendered. “Process” functions, which are new in Drupal 7, serve the same purpose, with the only difference being that they run later (after preprocess) in the processing cycle.

# A good example of how Drupal uses preprocess and process functions is the $classes\_array and $classes variables. In template\_preprocess() in Listing 16–1, which is the default implementation of preprocess by Drupal and the first preprocess function called, the $classes\_array variable is initialized; see http://api.drupal.org/api/function/template\_preprocess/7.

# Listing 16–1. Excerpt from template\_preprocess() where $classes\_array is defined

# <?php

# function template\_preprocess(&$variables, $hook) { // Initialize html class attribute for the current hook. $variables['classes\_array'] = array(drupal\_html\_class($hook)); }

# This first step provides a class indicating the hook that’s being used. For example, if this preprocess function is being called for a node, this code will add the class node to this array. After this function runs, all modules and themes also have a chance to run it themselves and add or change any of the variables. Next up is the Node module, which implements template\_preprocess\_node(); see http://api.drupal.org/api/function/template\_preprocess\_node/7. As you can see in Listing 16–2, quite a few classes are added to this array.

# Listing 16–2. Excerpt from template\_preprocess\_node() where additional classes are added to the $classes\_array variable

# <?php

# function template\_preprocess\_node(&$variables) {

# // Gather node classes.

# $variables['classes\_array'][] = drupal\_html\_class('node-' . $node->type);

# if ($variables['promote']) { $variables['classes\_array'][] = 'node-promoted'; }

# if ($variables['sticky']) { $variables['classes\_array'][] = 'node-sticky'; }

# if (!$variables['status']) { $variables['classes\_array'][] = 'node-unpublished'; }

# if ($variables['teaser']) { $variables['classes\_array'][] = 'node-teaser'; }

# if (isset($variables['preview'])) { $variables['classes\_array'][] = 'node-preview'; }

# }

# Once again, after template\_preprocess\_node() runs, all modules and themes have a chance to implement their own version, making any changes or additions they want. Once all the preprocess functions have completed, the process functions have their chance. In Drupal core, there are only two process implementations for nodes: template\_process(), the default implementation, and rdf\_process(), an implementation by the RDF module.

# In template\_process(), after all the modules and themes have had a chance to modify it, a new variable called $classes is created. It contains a string version all of the classes provided in $classes\_array. The $classes variable is printed in the class attribute of the wrapper <div> in the node.tpl.php template file. This is shown in Listing 16–3.

# Listing 16–3. Excerpt from template\_process() where $classes is created from the $classes\_array variable

# <?php

# function template\_process(&$variables, $hook) { // Flatten out classes. $variables['classes'] = implode(' ', $variables['classes\_array']); }

# Listings 16–1 through 16–3 illustrates some of the flexibility and power that Drupal provides with preprocess and process functions as well as the order in which these functions occur. The most important thing to understand is that in the theme layer, you’ve got the last call on all of these variables. You can easily add, modify, and remove any variables you please by simply implementing preprocess and process functions in your theme; this will be covered in more detail in the following pages.

# The big advantage of using preprocess functions is that they allow you to keep most of the logic outside of your template files. This allows for cleaner and easier to understand template files plus more efficient themes that are easier to maintain, manage, and extend over time. There are many changes you can make, such as affecting classes and modifying existing variables, that don’t require any changes to template files at all—just a few simple lines of code.

# IMPLEMENTING PREPROCESS AND PROCESS HOOKS

# Preprocess functions are implemented by creating a function that is named in a certain way. [Listing 16–4](http://themery.com/dgd7/advanced-theming/preprocess-process/implement#listing-16-4) shows an example of this naming convention.

# Listing 16–4. Naming Convention for Preprocess and Process Hooks

# <?php

# /\*\*  \* Implements template\_preprocess\_THEMEHOOK().  \*/

# function HOOK\_preprocess\_THEMEHOOK(&$variables) { // Changes go here. }

# /\*\*  \* Implements template\_process\_THEMEHOOK().  \*/

# function HOOK\_process\_THEMEHOOK(&$variables) { // Changes go here. }

# There are four points to consider in naming these functions:

# The hook of a default implementation, usually created by a module, is “template.” In all other implementations, the hook is replaced by the system name of the module or theme implementing it.

# Which stage of the process do you want to affect? There are two options: the preprocess, which runs first, or process, which runs after all of the preprocess functions have been executed.

# The theme hook matches the theme hook as defined in hook\_theme(), which is ultimately output using either a theme function or a template file.

# The &$variables parameter contains data needed by the theme function or template file rendering it. Since preprocess functions run before templates are rendered, you can make all sorts of changes and additions to its contents.

# CAUTION By default, only theme hooks that have been explicitly defined in hook\_theme() are able to use preprocess hooks. For example, hook\_preprocess\_node() is perfectly fine, but hook\_preprocess\_node\_\_article() will not work. This is because node\_\_article is a theme hook suggestion, which is a variation of a theme hook but is not actually a real theme hook.

# DEFAULT IMPLEMENTATIONS

# Listing 16–5 illustrates what a preprocess implementation for a default theme hook looks like, using template\_preprocess\_node(), which creates variables for the node.tpl.php template file as an example. This function resides in node.module along with a hook\_theme() implementation, node\_theme(), where it defines “node” as a theme hook.

# Listing 16–5. Naming convention for default implementations of preprocess and process hooks

# <?php

# function template\_preprocess\_node(&$variables) { // Changes go here. // See http://api.drupal.org/api/function/template\_preprocess\_node/7 for contents. }

# function template\_process\_node(&$variables) { // Changes go here. // See http://api.drupal.org/api/function/template\_process\_node/7 for contents. }

# TIP Browsing http://api.drupal.org and looking through the default implementations is a great way to learn how the variables were created.

# THEME AND MODULE IMPLEMENTATIONS

# Both modules and themes are able to use preprocess functions in the same way, and a given theme hook can have many preprocess implementations, originating from both modules and themes. This introduces the opportunity for conflicts to occur, so keeping that in mind and knowing the order in which these functions run is important. Preprocess implementations from modules run first, and implementations by themes run last. When dealing with base and subthemes, the base theme will run first and the subtheme will run last. A good way to remember this is that the active theme always wins.

# Preprocess functions implemented by Drupal core and modules reside in various files, such as modulename.module or theme.inc, and many others, while preprocess functions implemented by themes always reside in template.php.

# As an example, implement a preprocess function for a theme called “dgd7” for the node theme hook. As shown in [Listing 16–6](http://themery.com/dgd7/advanced-theming/preprocess-process/implement#listing-16-6), you simply place a function in template.php beginning with the theme name (the implementing hook), followed by \_preprocess\_ and the theme hook, which in this case is “node.” Finally, you pass in the &$variables parameter by reference (the & before the $ indicates a variable being passed by reference).

# Listing 16–6. Implementation of template\_preprocess\_node() in a theme

# <?php

# /\*\*  \* Implements template\_process\_node().  \*/

# function dgd7\_preprocess\_node(&$variables) { // Changes go here. }

# The code in [Listing 16–6](http://themery.com/dgd7/advanced-theming/preprocess-process/implement#listing-16-6) is all that’s needed, along with a quick cache clear, for Drupal to pick up your preprocess function and run it before rendering a node. Now the fun can begin!

# FINDING THE CONTENTS OF $VARIABLES

# The contents of the $variables array are different for each theme hook; even the contents of the same theme hook vary based on other factors, such as the view mode or user role.

# The first thing to do after creating the function is to print the array and find out what’s inside for you to work with. As explained in the “Finding Available Variables in the Theme Layer” section, using the dpm() function is a great way to do this, as shown in [Listing 16–7](http://themery.com/dgd7/advanced-theming/preprocess-process/variables#listing-16-7).

# Listing 16–7. Printing variables to the screen for debugging purposes

# <?php

# /\*\*

# \* Implements template\_preprocess\_node().

# \*/

# function dgd7\_preprocess\_node(&$variables) {

# dpm($variables);

# }

# CAUTION Debugging functions should only be used temporarily during development.

# PREPROCESS FUNCTIONS IN ACTION

# There are so many things you can change using preprocess functions that we can’t possibly get into all of them. Now that you’ve got your preprocess function all set up and are aware of how to view existing variables, you are equipped with enough knowledge to start making some changes. Let’s just jump right in and get started with a few practical examples of how to use preprocess functions.

# ADD CLASSES TO TEMPLATE WRAPPERS

# In the DGD7 theme at http://definitivedrupal.org, the header, sidebar, and footer areas are black and the content area is white. In order to style the contents of each of those sections more easily, you can add a couple of helper classes to the region wrapper. To do this, you’ll need to implement a preprocess function for the region theme hook in your template.php file; see [Listing 16–8](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-8).

# Listing 16–8. Adding classes to the region wrapper <div> using the $classes\_array variable in template\_preprocess\_region()

# <?php

# /\*\*

# \* Implements template\_preprocess\_region().

# \*/

# function dgd7\_preprocess\_region(&$variables) {

# $region = $variables['region'];

# 

# // Sidebars and content area need a good class to style against. You should

# // not be using id's like #main or #main-wrapper to style contents.

# if (in\_array($region, array('sidebar\_first', 'sidebar\_second', 'content'))) {

# $variables['classes\_array'][] = 'main';

# }

# // Add a "clearfix" class to certain regions to clear floated elements inside them.

# if (in\_array($region, array('footer', 'help', 'highlight'))) {

# $variables['classes\_array'][] = 'clearfix';

# }

# // Add an "outer" class to the darker regions.

# if (in\_array($region, array('header', 'footer', 'sidebar\_first', 'sidebar\_second'))) {

# $variables['classes\_array'][] = 'outer';

# }

# }

# $variables['classes\_array'] turns into $class in the process phase, and the class(es) added during preprocess are automatically modified as a result. So, just like that you’ve added a class to a region wrapper <div>.

# The alternative in template files is lengthier. Adding logic to each affected template file would be required, which means you’d need to override the file, even if you didn’t need to change the markup. If you have multiple template files for regions, the change would have to be made manually across all of them, which is clearly less efficient as you can see in [Listing 16–9](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-9).

# Listing 16–9. Adding classes in preprocess functions can dramatically increase the efficiency of your CSS code.

# /\* Using classes and ID's provided by default. \*/

# #header fieldset,

# #footer fieldset,

# .sidebar fieldset {

# border-color: #333;

# }

# 

# /\* Using the class added in Listing 16–8, which is more efficient. \*/

# .outer fieldset {

# border-color: #333;

# }

# TIP This example changes classes for the region template, but this technique can be applied to any of the major templates, including html.tpl.php, block.tpl.php, node.tpl.php and comment.tpl.php in their respective preprocess functions.

# MAKING CHANGES TO NODES

# [Listing 16–10](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-10) demonstrates making three changes:

# Drupal’s page title prints in page.tpl.php. When a node title prints inside of the node.tpl.php file, it’s usually because it’s being viewed in teaser mode, and therefore, the node title is marked up with an <h2> by default. Usually, the content inside the node body also contains one or more <h2> tags. Adding a class to single out the node title can make styling easier. [Listing 16–10](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-10) utilizes the $title\_attributes\_array to add a node-title class to help make styling easier.

# When viewing a node that has a comment form directly under the node links, it doesn’t make much sense to have an “Add new comment” link as well. In [Listing 16–10](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-10), the comment links are hidden when the comment form is below it by using the hide() function, which will be covered in more detail later in this chapter.

# Designs often call for differences when viewing the teaser of a node versus the full page. [Listing 16–10](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-10) demonstrates using $variables['teaser'] to suppress the $submitted information and truncate the node title to 70 characters when viewing in teaser mode.

# Listing 16–10. Demonstrates making changes to the display of node content during preprocess.

# <?php

# /\*\*

# \* Implements template\_preprocess\_node().

# \*/

# function dgd7\_preprocess\_node(&$variables) {

# // Give the <h2> containing the teaser node title a better class.

# $variables['title\_attributes\_array']['class'][] = 'node-title';

# 

# // Remove the "Add new comment" link when the form is below it.

# if (!empty($variables['content']['comments']['comment\_form'])) {

# hide($variables['content']['links']['comment']);

# }

# 

# // Make some changes when in teaser mode.

# if ($variables['teaser']) {

# // Don't display author or date information.

# $variables['display\_submitted'] = FALSE;

# // Trim the node title and append an ellipsis.

# $variables['title'] = truncate\_utf8($variables['title'], 70, TRUE, TRUE);

# }

# }

# ADD A CHANGE PICTURE LINK UNDERNEATH THE USER PHOTO

# As you’ve probably noticed by now, there are many variables available to you within the $variables array. These variables can be used to create new variables very easily. You know the path to edit a user profile is user/UID/edit, so you can use the information inside of $variables to determine whether or not the user viewing the page is the account holder. Once you’ve determined this, you can easily create a variable containing a link for the user to edit the photo everywhere it appears on the site by implementing template\_preprocess\_user\_picture(), as shown in [Listing 16–11](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-11). Once you do this, you’ll be able to print it in the corresponding template, user-picture.tpl.php, as shown in [Listing 16–12](http://themery.com/dgd7/advanced-theming/preprocess-process/in-action#listing-16-12).

# Listing 16–11. Creating a Custom Variable for the user-picture.tpl.php by Implementing template\_preprocess\_user\_picture().

# <?php

# /\*\*

# \* Implements template\_preprocess\_user\_picture().

# \* - Add "change picture" link to be placed underneath the user image.

# \*/

# function dgd7\_preprocess\_user\_picture(&$variables) {

# // Create a variable with an empty string to prevent PHP notices when

# // attempting to print the variable.

# $variables['edit\_picture'] = '';

# 

# // The account object contains the information of the user whose photo is

# // being processed. Compare that to the user id of the user object which

# // represents the currently logged in user.

# if ($variables['account']->uid == $variables['user']->uid) {

# // Create a variable containing a link to the user profile, with a class

# // "change-user-picture" to style against with CSS.

# $variables['edit\_picture'] = l('Change picture', 'user/' . $vars['account']->uid . '/edit',

# array(

# 'fragment' => 'edit-picture',

# 'attributes' => array('class' => array('change-user-picture')),

# )

# );

# }

# }

# Listing 16–12. Printing your custom variable Into the user-picture.tpl.php file, which you’ve copied into your theme to override

# <?php if ($user\_picture): ?>

# <div class="user-picture">

# <?php print $user\_picture; ?>

# <?php print $edit\_picture; ?>

# </div>

# <?php endif; ?>

# USING THE RENDER API

## WHAT IS A RENDER ARRAY?

# Many of the variables in template files are straightforward, but you’ll notice that some of the variables are printed along with a function called render(). Render arrays are structured arrays that contain nested data and other information needed by Drupal to turn them into HTML using Drupal’s Render API. Variables that are render arrays are generally easy to spot in template files because they are printed using a function called render().

# In page.tpl.php, you’ll notice that all of the regions are printed using the render() function. Each region is an element (another array) nested inside the $page array. The code in [Listing 16–13](http://themery.com/dgd7/advanced-theming/render#listing-16-13) is all that’s needed to render each region. Each render() call returns fully formatted HTML for all the contents of the render array.

# Listing 16–13. Printing regions in page.tpl.php using the render() function

# <?php print render($page['sidebar\_first']); ?>

# In prior versions of Drupal, you would just include <?php print $sidebar\_first; ?>, which contained a fully formatted HTML string ready for printing. This worked, of course, but it wasn't very flexible. Let’s face it; there are only so many things you can do with a big glob of HTML markup at that stage.

# In Drupal 7, these variables are sent to templates as nicely structured arrays. Instead of a glob of HTML markup, you get an array containing all sorts of information about the content inside it, down to attributes of specific links deep inside of it. This makes it incredibly easy to target specific content of the arrays and make any sort of changes you want to it at the last possible minute before it's rendered in the first place.

# To find out what’s inside this array, use the dpm() function provided with the Devel module to print it inside of page.tpl.php: <?php dpm($page['sidebar\_first']); ?>. As you can see in [Figure 16–4](http://themery.com/dgd7/advanced-theming/render#figure-16-4), there are two top-level render elements inside this array, the Search form block and the Navigation block, which are currently printing in the first sidebar.

# Screenshot of sidebar_first region array using dpm()Figure 16–4. Contents of the $page['sidebar\_first'] render array printed from page.tpl.php using dpm()

# IDENTIFYING RENDER ELEMENTS

# An easy way to identify arrays as render elements is the presence of properties. Render elements are always arrays, and they always contain properties that always begin with a hash tag. In [Figure 16–4](http://themery.com/dgd7/advanced-theming/render/render-elements#figure-16-4), you can immediately tell that $page['sidebar\_first'] is a render element because it contains a few properties: #sorted, #theme\_wrappers, and #region. These properties are used by drupal\_render() which is called when using drupal\_render() to determine how to render the output. For details about drupal\_render() see http://api.drupal.org/api/function/render/7.

# As themers, you won’t be getting deep into the more developer-centric properties, but there are a few that will be helpful for you to make sense of out what these arrays mean. These are described in [Table 16–1](http://themery.com/dgd7/advanced-theming/render/render-elements#table-16-1).

# Table 16–1. Helpful render element properties

| See the documentation at http://api.drupal.org/api/function/drupal\_render/7 for more information. | |
| --- | --- |
| PROPERTY | DESCRIPTION |
| #theme | Specifies the theme hook, which can be either a function or a template to use when rendering the element. |
| #theme\_wrappers | An array containing theme hook(s) to be used to wrap the rendered children of the element. For example, when theming a block, the #themeproperty would be block and the #theme\_wrappers property would contain region. This ensures that after the block(s) are rendered, the children would be run through the region template as well. |
| #type | The type of element that will be rendered. The default properties for element types are defined in hook\_element\_info() implementations. |
| #prefix and #suffix | A string containing markup to be placed before (prefix) or after (suffix) the rendered element. |
| #weight | A number that is used to sort the elements to determine the order in which they will print. |
| #sorted | A Boolean (TRUE or FALSE) that indicates whether or not the children have been sorted. For example, this is used in conjunction with the #weight property to sort the blocks in a region. When reordering blocks in a theme via hook\_page\_alter(), you'll need to specify #sorted => FALSE in addition to the #weight to trigger a new sort when you need to move a block to any other position than below the already sorted elements. |
| #attached | The #attached property is used to specify corresponding CSS, JavaScript, or libraries to load when the element is rendered. |

# MANIPULATING THE OUTPUT OF RENDER ELEMENTS

# As mentioned, having a structured array to work with is far more flexible than a bunch of HTML. This allows you to make only the changes you want to make with ease, whether big or small, without having to re-write code from scratch.

# The prospect of using render arrays to generate markup and using alter hooks in general are completely new concepts to Drupal theme developers. It’s very different than what you are used to, in a good way, but it takes some getting used to. In a lot of ways it’s easier than creating templates and theme functions for one-off implementations. The biggest issues front-end developers face when using the Render API are:

# Thinking about generating markup differently.

# Figuring out how to modify the content of a render array.

# Getting comfortable with implementing alter hooks.

# Unlike theme hooks, render arrays are modified using alter hooks, not preprocess functions and templates. This can be confusing at first because render arrays are similar to theme hooks in that their purpose is to ultimately generate HTML markup, and they use templates and theme functions to do so. With render arrays, the #theme property, which allows you to define which theme function or template should be used to render the element, is just one of many properties used and can be changed at any time. In general, you’ll use templates and theme functions to modify the markup itself, and you’ll use alter hooks to modify contents, structure, or placement of the elements before it’s rendered.

# The following sections contain a few examples of things you can do with render arrays.

# GENERATE NEW CONTENT ON THE FLY

# Generating new content is as simple as adding a new element to the page array. [Listing 16–14](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-14) shows the addition of a new element called “new\_stuff” to the pre-existing Highlighted region by implementing hook\_page\_alter() in a theme’s template.php.

# Listing 16–14. Adding a new element to the highlighted region

# <?php

# /\*\*

# \* Implements hook\_page\_alter().

# \*/

# function mytheme\_page\_alter(&$page) {

# $page['highlighted']['new\_stuff'] = array(

# '#type' => 'container',

# '#attributes' => array('class' => 'my-container'),

# );

# $page['highlighted']['new\_stuff']['heading'] = array(

# '#type' => 'html\_tag',

# '#tag' => 'h2',

# '#value' => t('Heading'),

# '#attributes' => array('id' => 'my-heading'),

# );

# $page['highlighted']['new\_stuff']['list'] = array(

# '#theme' => 'item\_list',

# '#items' => array(

# 'First item',

# 'Second item',

# 'Third item',

# ),

# );

# }

# The first thing you did was name your new element “new\_stuff,” gave it a #type of container, and defined a class attribute of my-container. Note that container is an element, defined in system\_element\_info(), which uses the theme\_container() theme function as a theme wrapper by default. This means the children of your element (heading and list) will be run through theme\_container(). The resulting markup is shown in [Listing 16–15](http://themery.com/dgd7/advanced-theming/render/listing-16-15).

# Listing 16–15. The output generated for $page['highlighted']['new\_stuff'] by theme\_container()

# <div class="my-container">

# ...

# </div>

# Then you added a subelement called “heading” and specified the #type element property as html\_tag. This will cause the element to use theme\_html\_tag() when rendering. You also specified #tag, #value, and #attributes properties. These are parameters of the theme\_html\_tag() function as you can see at http://api.drupal.org/api/function/theme\_html\_tag/7. The resulting markup is shown in [Listing 16–16](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-16).

# Listing 16–16. The output generated for $page['highlighted']['new\_stuff']['heading'] by theme\_html\_tag()

# <h2 id="my-heading">Heading</h2>

# Finally, you added a subelement called “list.” Here you specified item\_list as the #theme property and included an array containing your #items, which is a required parameter for theme\_item\_list(). The resulting markup is shown in [Listing 16–17](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-17).

# Listing 16–17. The output generated for $page['highlighted']['new\_stuff']['list'] by theme\_item\_list()

# <div class="item-list">

# <ul>

# <li class="first">First item</li>

# <li>Second item</li>

# <li class="last">Third item</li>

# </ul>

# </div>

# When the Highlighted region is rendered, the code in [Listing 16–14](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-14) produces the final result shown in [Listing 16–18](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-18).

# Listing 16–18. The final rendered result of Listing 16-14.

# <div class="my-container">

# <h2 id="my-heading">Heading</h2>

# <div class="item-list">

# <ul>

# <li class="first">First item</li>

# <li>Second item</li>

# <li class="last">Third item</li>

# </ul>

# </div>

# </div>

# CAUTION The previous examples are meant to illustrate how the Render API works to generate content. However, it’s worth noting that it should not be abused to output every piece of HTML on a page as separate elements because there can be serious performance implications. Using the markup #type is preferred for small bits of markup, such as headings, instead of html\_tag, as it requires the theme\_html\_tag() theme function to determine the output.

# MOVE CONTENT FROM ONE REGION TO ANOTHER

# Inside a hook\_page\_alter() implementation, you can move the content of regions around at will. [Listing 16–19](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-19) contains a few simple lines of code that move the contents of the entire first sidebar to the second sidebar, which results in the layout changing from a left sidebar layout to a right sidebar layout on full node pages. In [Listing 16–19](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-19), you’ve also moved the breadcrumbs to the bottom of the footer region.

# Listing 16–19. Relocate the sidebar\_first region to sidebar\_second and add breadcrumbs to a new element in the footer region.

# <?php

# /\*\*

# \* Implements hook\_page\_alter().

# \*/

# function dgd7\_page\_alter(&$page) {

# // Check that you are viewing a full page node.

# if (node\_is\_page(menu\_get\_object())) {

# // Assign the contents of sidebar\_first to sidebar\_second. $page['sidebar\_second'] = $page['sidebar\_first'];

# // Unset sidebar\_first.

# unset($page['sidebar\_first']);

# }

# 

# // Add the breadcrumbs to the bottom of the footer region.

# $page['footer']['breadcrumbs'] = array(

# '#type' => 'container',

# '#attributes' => array('class' => array('breadcrumb-wrapper', 'clearfix')), '#weight' => 10,

# );

# $page['footer']['breadcrumbs']['breadcrumb'] = array(

# '#theme' => 'breadcrumb',

# '#breadcrumb' => drupal\_get\_breadcrumb(),

# );

# 

# // Trigger the contents of the region to be re-sorted.

# $page['footer']['#sorted'] = FALSE;

# }

# ALTERING CONTENT INSIDE A RENDER ARRAY

# Altering the contents of a render array to change bits and pieces of the actual content is where you get into a very gray area. It could be argued that a change like this belongs inside a module. When making changes like this, it’s important to ask yourself whether or not the changes you are making should still apply when the theme you are developing is not active. [Listing 16–20](http://themery.com/dgd7/advanced-theming/render/manipulate#listing-16-20) changes the "View" and "Edit" tabs to read "Profile" and "Edit profile" on user profile pages.

# Listing 16–20. Implements hook\_menu\_local\_tasks\_alter() to change tab names on user profile pages.

# <?php

# /\*\*

# \* Implements hook\_menu\_local\_tasks\_alter().

# \*/

# function dgd7\_menu\_local\_tasks\_alter(&$data, $router\_item, $root\_path) {

# if ($root\_path == 'user/%') {

# // Change the first tab title from 'View' to 'Profile'.

# if ($data['tabs'][0]['output'][0]['#link']['title'] == t('View')) {

# $data['tabs'][0]['output'][0]['#link']['title'] = t('Profile');

# }

# // Change the second tab title from 'Edit' to 'Edit profile'.

# if ($data['tabs'][0]['output'][1]['#link']['title'] == t('Edit')) {

# $data['tabs'][0]['output'][1]['#link']['title'] = t('Edit profile');

# }

# }

# }

# NOTABLE RENDER ARRAYS IN CORE TEMPLATES

# There are quite a few render array variables scattered across core templates that are worth noting. hook\_page\_alter() contains the entire page so it can always be used to alter anything. However, finding that particular anything is not always trivial as other modules can move stuff around, so using more specific alters is advised. [Table 16–2](http://themery.com/dgd7/advanced-theming/render/core-templates#table-16-2) is a quick reference of notable render arrays. This is by no means a full list, but it covers quite a bit and should give you an idea of how to begin figuring out where to look to edit these things.

# Table 16–2. Notable render arrays in core templates.

| VARIABLE | FOUND IN | ALTER HOOK | DESCRIPTION |
| --- | --- | --- | --- |
| $page | page.tpl.php | hook\_page\_alter() | Contains the entire page from regions down to fields and comments. |
| $content | node.tpl.php, comment.tpl.php, taxonomy-term.tpl.php | hook\_node\_view\_alter(),  hook\_comment\_view\_alter(),  hook\_taxonomy\_term\_view\_alter() | Contains the contents of each entity. For more details see http://api.drupal.org/hook\_entity\_view\_alter. |
| $tabs | page.tpl.php | hook\_menu\_local\_tasks\_alter() | Contains primary and secondary tabs, themable via theme\_menu\_local\_tasks() and theme\_menu\_local\_task(). |
| $action\_links | page.tpl.php | hook\_menu\_local\_tasks\_alter() | Contains action links, themable via theme\_menu\_local\_actions(). |
| $item | field.tpl.php | hook\_field\_display\_alter() and  hook\_field\_display\_ENTITY\_TYPE\_ alter() | Contains display settings for fields, which can adjust label settings or control the formatter used to display the contents of field.tpl.php. |

# INTRODUCING RENDER(), HIDE(), AND SHOW()

# One of the best new theming features of Drupal 7 is the ability to selectively render bits of content in templates. As detailed in the previous sections, the content of some variables (render arrays) is sent to templates as structured arrays instead of chunks of HTML. This is really awesome news for the theme layer.

# To understand just how awesome this is, you need to look into the past. In prior versions of Drupal, theming complex nodes with fields wasn’t the easiest task. Fields were lumped into the $content variable, and while they could be printed and manipulated individually, there were issues. You had to be very careful to properly sanitize variables, and once you decided to break up the content variable, you needed to rebuild it entirely. This was not future-proof, as the addition of new fields would often require going back to the template file and printing the new field.

# In Drupal 7, those problems have been solved quite gracefully. You now have the ability to very easily render individual pieces of content, such as fields, with three new functions called render(), hide(), and show(). They can be used inside theme functions and templates files as well as preprocess and process functions. All three of these functions take a single argument, which is the element (or child) you wish to target.

# hide()

# Hides a render element or part of a render element by tricking drupal\_render() into thinking it has already been printed. Example usage:

# <?php hide($element['something']); ?>

# show()

# Does the opposite of hide(). It can be useful to revert a previously applied hide() status. Example usage:

# <?php show($element['something']); ?>

# render()

# Converts a render array to HTML markup. It returns HTML, so it should be used along with print in templates. Example usage:

# <?php print render($element); ?>

# To demonstrate these functions in action, look at node.tpl.php (see [Listing 16–21](http://themery.com/dgd7/advanced-theming/render/render-hide-show#listing-16-21)).

# Listing 16–21. Excerpt from the default node.tpl.php template

# <div id="node-<?php print $node->nid; ?>" class="<?php print $classes; ?> clearfix"<?php print $attributes; ?>>

# <?php print $user\_picture; ?>

# <?php print render($title\_prefix); ?>

# <?php if (!$page): ?>

# <h2<?php print $title\_attributes; ?>><a href="<?php print $node\_url; ?>"><?php print $title; ?></a></h2>

# <?php endif; ?>

# <?php print render($title\_suffix); ?>

# <?php if ($display\_submitted): ?>

# <div class="submitted">

# <?php print $submitted; ?>

# </div>

# <?php endif; ?>

# <div class="content"<?php print $content\_attributes; ?>>

# <?php

# // Hide the comments and links now so they can be rendered later.

# hide($content['comments']);

# hide($content['links']);

# print render($content);

# ?>

# </div>

# <?php print render($content['links']); ?>

# <?php print render($content['comments']); ?>

# </div>

# As you can see in [Listing 16–21](http://themery.com/dgd7/advanced-theming/render/listing-16-21), this template is already making use of both render() and hide() functions out of the box. There are three render arrays in this node template: $title\_prefix, $title\_suffix, and $content. Inside the <div class="content"> wrapper, both $content['links'] and $content['comments'] are hidden using hide(), and then $content is rendered directly underneath.

# The reason that the comments and links are hidden is to break them out of the $content variable and allow them to be placed outside of the <div class="content"> wrapper. Both of the items are then rendered afterward using render() individually.

# Of course, the fun doesn’t have to stop at top-level variables. These functions work as deep into the array as you can go. As long as you pass in a proper render element (see the “Render API” section), you’ll be able to manipulate it with these functions.

# As an example, say you wanted to hide the “Add new comment” link when viewing a node that has a comment form on the page you’re viewing. You can simply check to see if the form exists in your array, and then hide that specific link group (comment). The code in [Listing 16–22](http://themery.com/dgd7/advanced-theming/render/render-hide-show#listing-16-22) demonstrates how to do this.

# Listing 16–22. Hiding the “Add new comment” link when the comment form is present

# <?php

# // Hide the "Add new comment" link when the comment form is present.

# if (!empty($vars['content']['comments']['comment\_form'])) {

# hide($vars['content']['links']['comment']);

# }

# // Print the rendered links afterward.

# print render($content['links']);

# Because the show() function resets the print status but does not print anything, it can be helpful to revert a previously applied hide(). In most cases, you’ll likely just use render() because it will allow you to print the element as many times as you need, as shown in [Listing 16–23](http://themery.com/dgd7/advanced-theming/render/render-hide-show#listing-16-23).

# Listing 16–23. Hiding the “Add new comment” link when the comment form Is present, but showing it again if some other condition is met

# <?php

# // Hide the "Add new comment" link when the comment form is present.

# if (!empty($content['comments']['comment\_form'])) {

# hide($content['links']['comment']);

# if ($some\_exception) {

# show($content['links']['comment']);

# }

# }

# // Print the rendered links afterward.

# print render($content['links']);

# TIP For complex templates, this code begins to get very messy in templates files. In those situations, it's best to do these operations in preprocess or process functions in order to keep your templates clean and more manageable.

# THEMING FORMS

# Theming forms is a little different than working with the usual template file or theme function. Form markup is generated using Drupal’s Form API. This makes it really easy for modules to build forms and guarantees consistency among generated elements. While the process of theming forms is quite different from what most front-end developers are used to, we think you’ll begin to appreciate the consistency and flexibility of theming Drupal’s forms.

# One thing Drupal is famous for is the ability to accomplish a single task in many different ways. Although none of Drupal’s forms ship with template files, they can easily be made to use them. Forms can also use preprocess functions, process functions, and alter hooks. So, how do you know when to use one over the other? This section will explain how forms are generated and will present a couple of examples using each method.

# HOW FORM MARKUP IS GENERATED

# Forms are generated by modules. The simple function shown in [Listing 16–24](http://themery.com/dgd7/advanced-theming/forms/generated-markup#listing-16-24) is all that is required to generate form markup. It looks really easy, doesn’t it? It is. Of course, there is more to the process to make it functional, such as validating the form and saving the submitted values, but the rest is not your concern in the theme layer. What’s important to you is the structure of a form and how it’s transformed from the $form array to actual markup.

# Listing 16-24. A simple unsubscribe form

# <?php

# function exampleform\_unsubscribe(&$form, $form\_state) {

# $form['email'] = array(

# '#type' => 'textfield',

# '#title' => t('E-mail address'),

# '#required' => TRUE,

# );

# $form['submit'] = array(

# '#type' => 'submit',

# '#value' => t('Remove me!'),

# );

# return $form;

# }

# In [Listing 16–24](http://themery.com/dgd7/advanced-theming/forms/generated-markup#listing-16-24), you define a very simple form with two elements: a textfield for the e-mail address and a submit button. When rendered, the result looks like those in [Figure 16–5](http://themery.com/dgd7/advanced-theming/forms/generated-markup#figure-16-5). The resulting markup is shown in [Listing 16–25](http://themery.com/dgd7/advanced-theming/forms/generated-markup#listing-16-25).

# Screenshot of rendered unsubscribe form Listing 16-5. Rendered form based on the code from [Listing 16–24](http://themery.com/dgd7/advanced-theming/forms/generated-markup#listing-16-24)Listing 16-25. The markup generated by Drupal for the exampleform\_unsubscribe() form in [Listing 16–24](http://themery.com/dgd7/advanced-theming/forms/listing-16-24)

# <form action="/example/unsubscribe" method="post" id="exampleform-unsubscribe" accept-charset="UTF-8">

# <div>

# <div class="form-item form-type-textfield form-item-email">

# <label for="edit-email">E-mail address <span class="form-required" title="This field is required.">\*</span></label>

# <input type="text" id="edit-email" name="email" value="" size="60" maxlength="128" class="form-text required" />

# </div>

# <input type="submit" id="edit-submit" name="op" value="Remove me!" class="form-submit" /> <input type="hidden" name="form\_build\_id" value="formjKkl1KLWJLnv0hM4DSVd8-40boTgBQAzWWhUn44c15Q" />

# <input type="hidden" name="form\_token" value="LB07DqsDXK9idWdOHLxUen7jKxm52JqTyHiR7-pNumA" />

# <input type="hidden" name="form\_id" value="exampleform\_unsubscribe" />

# </div>

# </form>

# FORM API ELEMENTS AND DEFAULT PROPERTIES

# In the exampleform\_unsubscribe() form, you’ve defined two form elements: the e-mail address and the submit element. The e-mail element’s #type property is textfield, which provides a single line text input. The submit element’s #type is submit, which is the Form API equivalent of <input type="submit" />.

# If you look closely at the generated markup in [Listing 16–25](http://themery.com/dgd7/advanced-theming/forms/generated-markup#listing-16-25), you’ll see that you only set two properties in each element, but your markup ended up with some additional attributes. This is because Drupal assigns a default set of properties to each element. In this case, you are using form, textfield, and submit elements, which are defined in system\_element\_info(), as shown in [Listing 16–26](http://themery.com/dgd7/advanced-theming/forms/generated-markup#listing-16-26). When the form is processed, Drupal merges the properties defined in the form with the default properties.

# Listing 16-26. Default element properties as defined in system\_element\_info() for textfield and submit elements

# <?php

# $types['form'] = array(

# '#method' => 'post',

# '#action' => request\_uri(), '#theme\_wrappers' => array('form'),

# );

# $types['textfield'] = array(

# '#input' => TRUE,

# '#size' => 60,

# '#maxlength' => 128,

# '#autocomplete\_path' => FALSE,

# '#process' => array('ajax\_process\_form'), '#theme' => 'textfield',

# '#theme\_wrappers' => array('form\_element'),

# );

# $types['submit'] = array(

# '#input' => TRUE,

# '#name' => 'op',

# '#button\_type' => 'submit',

# '#executes\_submit\_callback' => TRUE,

# '#limit\_validation\_errors' => FALSE,

# '#process' => array('ajax\_process\_form'),

# '#theme\_wrappers' => array('button'),

# );

# TIP This form only touches on a few of form elements, but Drupal has many of them. For a full list of elements available through the Form API and their default properties, see http://api.drupal.org/api/file/developer/topics/forms\_api\_reference.html/7.

# RENDERING OF FORM ELEMENTS

# The element properties contain critical information required to render them. Of these properties, two are very important in the theme layer: #theme and #theme\_wrappers. When it’s time to render the form, these properties tell Drupal which theme functions to use. There’s also the option to use the #pre\_render property to define a function(s) that should run prior to rendering.

# #theme

# Specifies the theme function to use when rendering the element.

# #theme\_wrappers

# Specifies a theme function or functions that should be used to wrap the rendered children of the element.

# To illustrate this process, let’s use the $form['email'] field from the previous form and walk through the process:

# theme('textfield', array('element' => $form['email'])) is called. This results in the following markup:

# <input type="text" id="edit-email" name="email" value="" size="60" maxlength="128" class="form-text required" />

# theme('form\_element', array('element' => $form['email'])) is called. This results in the following markup:

# <div class="form-item form-type-textfield form-item-email">

# <label for="edit-email">E-mail address <span class="form-required" title="This field is required.">\*</span> </label>

# <input type="text" id="edit-email" name="email" value="" size="60" maxlength="128" class="form-text required" />

# <!-- RESULT OF THE RENDERED TEXTFIELD -->

# </div>

# Finally, after all of the form elements are rendered, the form itself is run through theme\_form(), which is specified as the #theme\_wrappers in the form element. The theme\_form() function takes care of generating the rest of the form markup, including the hidden elements form\_build\_id, form\_token, and form\_id.

# CAUTION As mentioned previously, you never use theme\_ to call a theme function directly, and similarly theme functions are entered in #theme and #theme\_wrappers without the prefix theme\_.

# FIRST STEPS FOR THEMING FORMS

## FIND THE FORM ID

# Before you can do anything, you’ll need to find the ID of the form you’re working with. It appears in the following two places in the markup of every form:

# There’s a hidden field near the bottom of the form named form\_id that contains what you’re looking for.

# <input type="hidden" name="form\_id" value="exampleform\_unsubscribe" />

# Although it’s not copy/paste ready because it contains dashes instead of underscores to separate words, the <form>'s ID attribute also contains the form ID.

# <form id="exampleform-unsubscribe">

# Each Form ID has a corresponding function, which follows Drupal module naming conventions. In this example, exampleform is the module name and unsubscribe is what the form is named by the module.

# Sometimes it helps to look at the original form and code comments when theming. You’ll often find the original function that generates the form in the .module file of the module that created the form. If you find that the form doesn’t exist in the .module file, it’s definitely inside the module somewhere, but you may have to look around. Sometimes developers use .inc files for organization and code efficiency purposes.

# IMPLEMENT HOOK\_THEME()

# In order to be able to use template files, preprocess, or process functions with forms, the first thing you’ll need to do is register the form ID as a theme hook. This is necessary so that Drupal knows about the theme hook. Drupal core does this for some forms in core, mostly for administrative forms that use tables, but chances are you’ll need to do this manually.

# In your theme’s template.php file, you’ll create an implementation of hook\_theme(), with your theme’s name in place of the hook prefix. As an example, you’ll theme the contact form located at /contact when the Contact module is enabled, whose form ID is contact\_site\_form. Inside you’ll specify the form ID as the key and the render element as form, as shown in [Listing 16–27](http://themery.com/dgd7/advanced-theming/forms/first-steps#listing-16-27). The render element key is required for theme hooks that use the render API to generate markup, such as forms. Its value indicates the name of the variable that holds renderable element, which in this case is form.

# Listing 16–27. A hook\_theme() implementation that defines the contact\_site\_form() theme hook as render element “form”

# <?php

# /\*\*

# \* Implements hook\_theme().

# \*/

# function THEMENAME\_theme() {

# return array(

# // Defines the form ID as a theme hook.

# 'contact\_site\_form' => array(

# // Specifies 'form' as a render element.

# 'render element' => 'form',

# ),

# );

# }

# After doing this and clearing the cache, you’ll be able to create a theme function and use preprocess and process functions for this form, which you’ll get into later in the chapter.

# TIP When registering theme hooks, if you are unsure what to enter, look at some of the default implementations. In this case, you are dealing with a form, so a quick look at http://api.drupal.org/api /function/drupal\_common\_theme/7 reveals the defaults for the original form theme hook, which are exactly what you need here.

# THEMING FORMS WITH THEME FUNCTIONS

# The decision of whether to use a theme function or a template file is a personal/team preference. If you’re comfortable using PHP, you might be inclined to use theme functions. If not, you’ll probably prefer a template file, which is explained in the next section.

# As discussed above, you’ll need a hook\_theme() implementation, without a template or path index, as shown in [Listing 16–28](http://themery.com/dgd7/advanced-theming/forms/with-theme-functions#listing-16-28). After doing this, HOOK\_contact\_site\_form() is an official theme hook that can be overridden like any other theme function. Even though a theme\_contact\_site\_form() function doesn’t exist, you still name it as you would any other theme function override: THEMENAME\_contact\_site\_form().

# Listing 16–28. The basic required code for theming a form with a theme function

# <?php

# /\*\*

# \* Implements hook\_theme().

# \*/

# function dgd7\_theme() {

# return array(

# 'contact\_site\_form' => array(

# 'render element' => 'form',

# ),

# );

# }

# /\*\*

# \* Implements theme\_forms\_contact\_site\_form().

# \*/

# function dgd7\_contact\_site\_form($variables) {

# // Renders all elements of a form.

# return drupal\_render\_children($variables['form']);

# }

# USING DRUPAL\_RENDER\_CHILDREN() IS A MUST!

# drupal\_render\_children() takes care of rendering all of the children of the form. This function alone will result in the exact same code Drupal would have provided without your theme function, which makes the function in [Listing 16–28](http://themery.com/dgd7/advanced-theming/forms/with-theme-functions#listing-16-28) pretty useless by itself, but it’s worth stressing that it’s VERY important to always use drupal\_render\_children($variables['form']) at the bottom of your function.

# Even if you call render() on every element you have added to the form, Drupal will have added some important hidden elements identifying the form and those need to be rendered, too. So calling drupal\_render\_children($form) at the end of the theme function is mandatory. This won’t re-print $form['foo'] because drupal\_render() knows it has printed already. As an added bonus, it will take care of any additional elements added by other modules.

# MANIPULATING FORM ELEMENTS IN THEME FUNCTIONS

# Now that we've gotten that out of the way, let’s make some changes to the markup. Just like any theme function, the code this function returns will be inserted directly into the page markup. Since forms are render elements you need to render them. The code in [Listing 16–29](http://themery.com/dgd7/advanced-theming/forms/with-theme-functions#listing-16-29), does the following:

# Changes the labels of the name and mail elements.

# Renders the name and mail elements individually.

# Arranges the markup and individually rendered elements in a variable called $output.

# Includes drupal\_render\_children($form) in the $output at the bottom of the theme function.

# Finally, it returns the $output.

# Listing 16–29. Implements theme\_contact\_site\_form()

# <?php

# /\*\*

# \* Implements theme\_contact\_site\_form().

# \*/

# function dgd7\_contact\_site\_form($variables) {

# // Hide the subject field. It's not required.

# hide($variables['form']['subject']);

# 

# // Change the labels of the "name" and "mail" textfields.

# $variables['form']['name']['#title'] = t('Name');

# $variables['form']['mail']['#title'] = t('E-mail');

# 

# // Create output any way you want.

# $output = '<div class="something">';

# $output .= '<p class="note">'. t("We'd love hear from you. Expect to hear back from us in 1-2 business days.") .'</p>';

# $output .= render($variables['form']['name']);

# $output .= render($variables['form']['mail']);

# $output .= '</div>';

# 

# // Be sure to include a rendered version of the remaining form items.

# $output .= drupal\_render\_children($variables['form']);

# 

# // Return the output.

# return $output;

# }

# Forms and their contents are render elements, so you can use hide(), show(), and render()functions to manipulate the elements of the form. When using hide() or making changes to the form array inside the theme function, you’ll need to make sure you do so before attempting to render. There are a lot of other things that can be done here. We can’t possibly cover all of them, but here are a few quick examples of what can be done:

# Adjust the #weight property of an element to change the order in which they print. The following code would cause the message element to print at the top of the form:

# $variables['form']['message']['#weight'] = -10;

# $variables['form']['message']['#sorted'] = FALSE;

# Add a description underneath an element by setting the element #description property, like so:

# $variables['form']['mail']['#description'] = t("We won't share your e-mail with anyone.");

# Set the default value of form element, such as checking the "Send yourself a copy" checkbox, by default setting the #checked property to TRUE, like so:

# $variables['form']['copy']['#checked'] = TRUE;

# Unset the #theme\_wrappers property to remove the label and wrapper <div> and re-create the markup exactly the way you want it, like so:

# unset($variables['form']['mail']['#theme\_wrappers']);

# More advanced changes include making the form display in a table by using the theme\_table() function.

# … and so on!

# TIP Using theme functions over templates is slightly faster performance-wise, but the difference is very minimal. Performance isn’t something you should worry about when deciding whether to use a template file over a theme function unless it's for an element that can be used very frequently per page, such as theme\_link().

# THEMING FORMS WITH TEMPLATE FILES

# Creating template files for forms is surprisingly easy given what you’ve already learned. As mentioned in the “First Steps for Theming Forms” section, you’ll need to open template.php and implement a hook\_theme() function. Instead of just defining the render element, you’ll need to add two more things, as shown in [Listing 16–30](http://themery.com/dgd7/advanced-theming/forms/with-templates#listing-16-30):

# A path key (optional) that contains the path to where the template file is located in your theme.

# A template key that contains the name of the template file, without the .tpl.php suffix.

# CAUTION Template files defined this way are not auto-discovered. If the path is omitted, Drupal will only look for your template file in the root of the theme. Specifying the path of the template directory is only required if your file exists in a subdirectory of your theme.

# Listing 16–30. hook\_theme() implementation example for using templates with forms

# <?php

# /\*\*

# \* Implements hook\_theme().

# \*/

# function mytheme\_theme() {

# return array( 'contact\_site\_form' => array(

# 'render element' => 'form',

# 'path' => drupal\_get\_path('theme', 'mytheme') . '/templates',

# 'template' => 'contact-site-form',

# ),

# );

# }

# After creating the hook\_theme() function shown in [Listing 16–30](http://themery.com/dgd7/advanced-theming/forms/with-templates#listing-16-30), you’ll need to create the template file. In this case, it’s located in the templates directory within your theme:

# sites/all/themes/mytheme/templates/contact-site-form.tpl.php.

# Once that’s complete, simply clear the cache and Drupal will begin using your template file.

# If there’s nothing in your file to begin with, you’ll get a blank page where the form used to be. The first thing you should do is add this line back to the template file: <?php print drupal\_render\_children($form); ?>. This will get the entire form back, and even though you may not want to keep everything in the form, you need to print the contents of this at the bottom of the form to ensure everything works properly as we detailed in the “[Using drupal\_render\_children() is a Must!](http://themery.com/node/52)” section.

# MANIPULATING FORM ELEMENTS IN TEMPLATE FILES

# For the sake of covering this topic in detail, let’s use the example from the [Manipulating Form Elements in Theme Functions](http://themery.com/node/109) section. The code in [Listing 16–31](http://themery.com/dgd7/advanced-theming/forms/with-templates#listing-16-31) represents the result of completing the following tasks:

# Changing the labels for the name and mail elements.

# Rendering the name and mail elements individually.

# Arranging your markup and individually rendered elements as you want them.

# Finally, printing drupal\_render\_children($form) at the bottom of the template.

# Listing 16–31. contact-site-form.tpl.php implementation of the contact form

# <?php // Change the labels of the "name" and "mail" textfields.

# $form['name']['#title'] = t('Name');

# $form['mail']['#title'] = t('E-mail');

# ?>

# <?php // Render the "name" and "mail" elements individually and add markup. ?>

# <div class="name-and-email">

# <p><?php print t("We'd love hear from you. Expect to hear back from us in 1-2 business days.") ?></p>

# <?php print render($form['name']); ?>

# <?php print render($form['mail']); ?>

# </div>

# <?php // Be sure to render the remaining form items. ?> <?php print drupal\_render\_children($form); ?>

# While there are slight differences, it’s mostly the same (with less PHP). All of the possibilities that apply in theme functions apply just as well in template files. The variables themselves are slightly different. In theme functions and preprocess functions, the name element would be located in $variables['form']['name']. In template files, that same variable would be $form['name']. This is done specifically to make Drupal’s monster arrays easier on template authors.

# CAUTION Be sure not to hide or omit required form elements. In Drupal, presentation is totally separate from form processing. Drupal will expect those elements and prevent the form from being submitted if they are not filled in. These types of changes should be done in a hook\_form\_alter() implementation, using the #access property. See the “Modifying Forms Using Alter Hooks” section and Chapter 22 for more information.

# KEEP YOUR TEMPLATE CLEANER WITH PREPROCESS FUNCTIONS

# In our example of theming a form with a template file, the template is quite messy. The definition of a clean template file is one that contains hardly any logic and that simply prints variables and maybe an occasional IF statement. If you are dissatisfied with the appearance of the template file, this is a perfect opportunity to use preprocess functions. To make this really clean, you’d do the following in a preprocess function:

# Perform all modifications to the form array.

# Create any new variables.

# Render each field individually and provide easy variables for templates.

# Of course, this is not something you’d want to do on every form on your site. However, it’s very useful and convenient for highly styled user-facing forms that you want to take extra care to get right, such as the login, registration, and contact forms. The process of doing this is very easy, as demonstrated in [Listing 16–32](http://themery.com/dgd7/advanced-theming/forms/with-templates#listing-16-32) with the contact form.

# Listing 16–32. Using a preprocess function to do the heavy lifting for the template

# <?php

# /\*\*

# \* Implements hook\_preprocess\_contact\_site\_form().

# \*/

# function mytheme\_preprocess\_contact\_site\_form(&$variables) {

# // Shorten the form variable name for easier access.

# $form = $variables['form'];

# 

# // Change labels for the 'mail' and 'name' elements.

# $form['name']['#title'] = t('Name');

# $form['mail']['#title'] = t('E-mail');

# 

# // Create a new variable for your note.

# $variables['note'] = t("We'd love hear from you. Expect to hear back from us in 1-2 business days.");

# 

# // Create variables for individual elements.

# $variables['name'] = render($form['name']);

# $variables['email'] = render($form['mail']);

# $variables['subject'] = render($form['subject']);

# $variables['message'] = render($form['message']);

# $variables['copy'] = render($form['copy']);

# 

# // Be sure to print the remaining rendered form items.

# $variables['children'] = drupal\_render\_children($form);

# }

# Because you’ve done all the work in the preprocess function, the template file in [Listing 16–33](http://themery.com/dgd7/advanced-theming/forms/with-templates#listing-16-33) is crispy clean. Adding markup and classes and moving elements around is a piece of cake, and it’s very easy to see what this template file does at first glance.

# Listing 16–33. The result of using a preprocess function to provide a clean, minimal template for the contact form.

# <p class="note"><?php print $note; ?></p>

# <p><span class="form-required">\*</span> <?php print t("Denotes required fields."); ?></p>

# <ol>

# <li><?php print $name; ?></li>

# <li><?php print $email; ?></li>

# <li><?php print $subject; ?></li>

# <li><?php print $message; ?></li>

# <li><?php print $copy; ?></li>

# </ol>

# <?php print $children; ?>

# MODIFYING FORMS USING ALTER HOOKS

# The ability of themes to use alter hooks is new in Drupal 7. Templates are great for situations where you want to have a lot of control over the markup itself, but there are quite a few situations where simply using hook\_form\_alter() can make things a lot easier, especially if you are comfortable with Drupal’s form markup either by default, or in combination with changes you can make site-wide via theme functions. Using an alter hook is perfect for quick changes like:

# Simple changes to form labels, descriptions, and other properties.

# Changing the order in which the form elements print using the #weight property.

# Wrapping a few elements in a <div> or <fieldset>.

# Hiding or removing form elements that are not required.

# Adding some markup to a form.

# It’s also arguably easier because there are fewer steps involved in the process. You don’t need to implement hook\_theme(). You also get full control over the elements. There are certain limitations to the changes you can make within theme functions, as it’s already too late in the process.

# hook\_form\_alter(): Runs for all forms.

# hook\_form\_FORM\_ID\_alter(): Runs for a specific form ID.

# There are reasons for using hook\_form\_alter() over hook\_form\_FORM\_ID\_alter() all the time, but those reasons mainly apply to the tasks a module developer needs to perform. Unless you are specifically targeting more than one form to do the same thing, as shown in [Listing 16–34](http://themery.com/dgd7/advanced-theming/forms/with-alter-hooks#listing-16-34), it’s probably best to use hook\_form\_FORM\_ID\_alter(), as shown in [Listing 16–35](http://themery.com/dgd7/advanced-theming/forms/with-alter-hooks#listing-16-35).

# Listing 16–34. Implementation of hook\_form\_alter() to target all or multiple forms

# <?php

# /\*\*

# \* Implements hook\_form\_alter().

# \*/

# function mytheme\_form\_alter(&$form, &$form\_state, $form\_id) {

# // Changes made in here affect ALL forms.

# if (!empty($form['title']) && $form['title']['#type'] == 'textfield') {

# $form['title']['#size'] = 40;

# }

# }

# Listing 16–35. Implementation of hook\_form\_FORM\_ID\_alter() to target a specific form

# <?php

# /\*\*

# \* Implements hook\_form\_FORM\_ID\_alter().

# \*/

# function mytheme\_form\_contact\_site\_form\_alter(&$form, &$form\_state) {

# // Add a #markup element containing your note and make it display at the top.

# $form['note']['#markup'] = t("We'd love hear from you. Expect to hear back from us in 1-2 business days.");

# $form['note']['#weight'] = -1;

# 

# // Change labels for the 'mail' and 'name' elements.

# $form['name']['#title'] = t('Name');

# $form['mail']['#title'] = t('E-mail');

# 

# // Hide the subject field and give it a standard subject for value.

# $form['subject']['#type'] = 'hidden';

# $form['subject']['#value'] = t('Contact Form Submission');

# }

# MANAGING CSS FILES

# Every good Drupal theme needs a stylesheet or two, or ten! You might be caught off guard by the sheer number of CSS files that Drupal loads, before you even start on your theme. Being the modular framework that it is, Drupal uses that same approach for CSS stylesheets and JavaScript files. CSS and JavaScript files are provided separately by module—and sometimes a few per module. This is done on purpose for the following reasons:

# It’s easier to read and understand the purpose of the code and what module it belongs to.

# It allows Drupal to load only the code needed on a given page.

# It’s easier for Drupal to maintain these files and their contents.

# That said, in Drupal’s theme layer you have full control over all stylesheets and scripts. You can do whatever you want with them, literally. If you decide you don’t want to load any stylesheets from modules, you can remove them all. If you aren’t happy with a few files, you can override them individually by removing them or override them and change the contents within the theme. You can even change the order in which the files load if you want to. This section will show you how to do all of that.

# AGGREGATION AND COMPRESSION

# As mentioned, Drupal has many stylesheets. Of course, you want to keep the number of files at a minimum on your live sites for performance reasons, so Drupal has a way of handling this. During development, it’s normal to deal with anywhere from 10-40 CSS files, and even more if you are working on sites in languages that display text in right-to-left order. In the Performance section at /admin/config/development/performance there are options to aggregate and compress CSS and JavaScript files. When turned on, Drupal will minify and combine the files into as few automatically generated files as possible. This also effectively works around the Internet Explorer 31 stylesheet limit bug.

# Drupal aggregates files in two ways: it creates a per-site aggregation file from files that would be loaded on every page, and it creates a per-page aggregation files for the remaining files that are conditionally loaded depending on the page. For CSS files, it further aggregates by media type. To remain correct, if the contents of CSS and JavaScript files are changed, when the site cache is cleared Drupal will regenerate the aggregated versions of the files and give them a different name. Enabling aggregation and compression for CSS files on all live sites is highly recommended, as it will speed up page loads quite a bit. This process is very effective and allows themers and developers to continue developing sites in a modular manner, without having to worry about the number of CSS files.

# CAUTION Do NOT use the @import directive to load CSS files manually within Drupal. Doing so will cause performance and possible aggregation issues when combined with <link>’ed stylesheets and will cause the files to be excluded from override features.

# PATTERNS AND NAMING CONVENTIONS

# In your theme, you are free to name your CSS files whatever you want. Many themes tend to create a directory called “css” in which they place a few stylesheets. It’s very common to create a layout.css for page layout styles and style.css for the rest. Some themes, like Zen, take it much further with almost 30 stylesheets. How you decide to organize your CSS is completely up to you. There are no restrictions on how many stylesheets a theme can have. Most front-end developers have their own way of working, and Drupal is happy to oblige.

# CORE AND MODULE CSS FILES

# Most modules that provide CSS will typically include a file in the root of the module directory called module-name.css. Some of the modules have a few CSS files, and the better modules create a separate file for any CSS used to style the administrative user interface. Modules are not restricted to any number or specific organization of CSS files, but developers are generally urged to be conservative and style elements as little as possible.

# It’s also worth mentioning that Drupal’s System module, located in modules/system, contains quite a few CSS files that seem to all be dumped there because there is no better place to put them. [Table 16–3](http://themery.com/dgd7/advanced-theming/css/patterns-naming-convensions#table-16-3) provides a quick description of each, so you have an idea of what their purpose is and can decide whether or not to keep them in your theme.

# Table 16–3. System module’s CSS files, excluding RTL versions

| CSS FILE | PURPOSE | LOADS ON… |
| --- | --- | --- |
| system.base.css | Contains CSS that is heavily relied upon by JavaScript for certain functionality, including collapsible fieldsets, autocomplete fields, resizable textareas, and progress bars. | Every page. |
| system.theme.css | Contains general styles for many generic HTML and Drupal elements. | Every page. |
| system.menus.css | Contains default styling for menu tree lists, tabs, and node links. | Every page. |
| system.messages.css | Contains default styling for error, warning, and status messages. | Every page. |
| system.admin.css | Contains styles needed on administrative pages throughout Drupal. | Admin pages. |
| system.maintenance.css | Contains styles for installation, maintenance, and update tasks. | Maintenance pages. |

# BI-DIRECTIONAL TEXT SUPPORT

# One of the things Drupal is known for is its superb language support. This includes bi-directional text support. While most languages display text from left-to-right (LTR) on screen, certain languages, such as Arabic and Hebrew, display text from right-to-left (RTL) on screen. Browsers handle much of the styling differences needed by reading the dir attribute defined in the <html> tag and using User Agent CSS files, but many times CSS floats, text alignment, and padding need to be accounted for in CSS, especially when you are running a site with multiple languages.

# Drupal handles RTL stylesheets in an automated way based on CSS file naming conventions. If you have a stylesheet named style.css, which contains the CSS for the LTR version of the site, you can simply create another file called style-rtl.css to contain the necessary tweaks to fix the display for the RTL version. Drupal will automatically load it when needed, directly after the original file so that the same selectors can be used and RTL styles will override the LTR styles, taking advantage of the natural CSS cascade. When writing CSS for a site that will support both LTR and RTL displays, it is customary to write the CSS for the LTR version first, while keeping track of what will need to change (per property) with a comment. This is one of the coding standards Drupal has adopted for core and contributed CSS files. [Listing 16–36](http://themery.com/dgd7/advanced-theming/css/patterns-naming-convensions#listing-16-36) shows an example.

# Listing 16–36. Example CSS denoting a LTR property and the RTL version

# /\* In style.css:

# .my-selector floats content to the left, which is LTR-specific, so an inline comment is added to note this.

# \*/

# .my-selector {

# border: solid 1px #ccc;

# float: left; /\* LTR \*/

# }

# 

# /\* In style-rtl.css:

# The RTL version of .my-selector needs to be overriden and floated right instead of left. \*/

# .my-selector {

# float: right;

# }

# ADDING, REMOVING, AND REPLACING CSS FILES

# There are three ways to manipulate CSS files within Drupal themes. This section will explain what the implementation options are, the reasons for each method, and when it’s advantageous to use certain methods over others.

# QUICK AND DIRTY STYLESHEETS VIA .INFO FILES

# Adding stylesheets via your theme’s .info file is the easiest way to add a CSS file to your theme; see [Listing 16–37](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-37) and [Listing 16–38](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-38). However, there are a few drawbacks to doing this in certain situations.

# Any stylesheet you define in the .info file will load on every page.

# You don’t have the full use of the features available in drupal\_add\_css(). For example, you can’t add conditional stylesheets for Internet Explorer or change the weight of a module’s CSS file in your .info file.

# Listing 16–37. .info syntax for adding stylesheets

# stylesheets[CSS media type][] = path/to/file.css

# Listing 16–38. Typical .info stylesheet definition example

# stylesheets[all][] = css/layout.css

# stylesheets[all][] = css/style.css

# stylesheets[print][] = css/print.css

# CAUTION Stylesheets may also be removed via .info files by creating an entry for a file as if you were overriding it, but then not actually including the file in the theme directory. However, there is a bug that allows these stylesheets to return when AJAX rendering occurs. To be safe it’s best to remove stylesheets in hook\_css\_alter(); this is explained later in this section.

# CONDITIONALLY LOADING STYLESHEETS WITH DRUPAL\_ADD\_CSS()

# drupal\_add\_css() is the main function used by modules and themes to add CSS files via PHP code. Some themes use it in their template.php file, typically within preprocess functions. One of the advantages of using drupal\_add\_css() in the theme layer as opposed to defining CSS files in a .info file is that files can be conditionally loaded based on certain criteria or context. For example, you may want to create a special CSS file that only loads on your site’s home page. In your theme’s template.php, you could do this within template\_preprocess\_html(), as shown in [Listing 16–39](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-39).

# Listing 16–39. Adding a stylesheet that loads only on the home page

# <?php

# function mytheme\_preprocess\_html(&$variables) {

# // Add a stylesheet that prints only on the homepage.

# if ($variables['is\_front']) {

# drupal\_add\_css(path\_to\_theme() . '/css/homepage.css', array('weight' => CSS\_THEME));

# }

# }

# There are many different options for adding CSS to your pages in Drupal using drupal\_add\_css(), some of which include:

# Specifying the type as “inline” to print a block of CSS code within <head>, as opposed to adding a CSS file.

# Specifying the “group” of a file to determine where the file should appear using constants such as CSS\_SYSTEM (top), CSS\_DEFAULT (middle), and CSS\_THEME (bottom).

# Specifying the “weight” of a file to control the order in which it loads within its group.

# Adding conditional stylesheets to serve different files to different browsers.

# Adding externally hosted CSS files.

# Forcing a CSS file to be excluded from the aggregation and compression process.

# ADDING CONDITIONAL STYLESHEETS FOR INTERNET EXPLORER

# According to Wikipedia at the time of this writing, about 43 percent of users are visiting web pages using Internet Explorer. This statistic varies from source to source, but for many of you, supporting older versions of Internet Explorer is a fact of life. Using conditional stylesheets is considered a best practice when the need arises to write CSS that targets Internet Explorer.

# One of the great new features in Drupal 7 is that conditional stylesheets can be added using drupal\_add\_css(). In fact, all three of Drupal’s core themes do this in template\_preprocess\_html(). The reason this is done in template.php is that .info files only have very basic support for drupal\_add\_css(). [Listing 16–40](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-40) and [Listing 16–41](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-41) demonstrate how this works using code from the Seven theme as an example.

# Listing 16–40. Excerpt from the Seven theme, using drupal\_add\_css() to add conditional stylesheets for IE in template\_preprocess\_html()

# <?php

# function seven\_preprocess\_html(&$variables) {

# // Add conditional CSS for IE8 and below.

# drupal\_add\_css(path\_to\_theme() . '/ie.css', array('group' => CSS\_THEME, 'browsers' => array('IE' => 'lte IE 8', '!IE' => FALSE), 'preprocess' => FALSE));

# // Add conditional CSS for IE6.

# drupal\_add\_css(path\_to\_theme() . '/ie6.css', array('group' => CSS\_THEME, 'browsers' => array('IE' => 'lt IE 7', '!IE' => FALSE), 'preprocess' => FALSE));

# }

# Listing 16–41. The source code that results from adding IE conditional stylesheets with drupal\_add\_css()

# <!--[if lte IE 8]>

# <link type="text/css" rel="stylesheet" href="http://drupal-7/themes/seven/ie.css?l40z2j" media="all" />

# <![endif]-->

# <!--[if lt IE 7]>

# <link type="text/css" rel="stylesheet" href="http://drupal-7/themes/seven/ie6.css?l40z2j" media="all" />

# <![endif]-->

# The code in [Listing 16–40](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-40) and [Listing 16–41](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-41) gives you two conditional stylesheets that will load for Internet Explorer only. The first stylesheet will load for Internet Explorer 8 and under, and the second stylesheet will load for versions of Internet Explorer prior to IE7.

# COMPLETELY CONTROL STYLESHEETS USING HOOK\_CSS\_ALTER()

# Drupal core and modules add CSS files individually via the drupal\_add\_css() function. During template\_process\_html()</a>, a variable called <code lang="php">$styles is created; it contains the fully formatted HTML output for all the stylesheets that are specified for each page. This variable is eventually printed inside the <head> tags in the html.tpl.php template file, as shown in [Listing 16–42](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-42).

# Listing 16–42. $styles variable is created in template\_process\_html() for use in html.tpl.php

# <?php

# /\*\*

# \* Implements template\_process\_html().

# \*/

# function template\_process\_html(&$variables) {

# ...

# $variables['styles'] = drupal\_get\_css();

# ...

# }

# During the call to drupal\_get\_css(), Drupal gathers up all the CSS files previously added, and then provides an opportunity for any modules or themes to make changes by calling drupal\_alter('css', $css). At this time, Drupal looks for functions in modules and themes that fit the naming pattern hook\_css\_alter(), where the word “hook” in the function name is replaced by the module or theme name implementing it. This function allows for the most granular control over all aspects of your CSS files.

# An example of why a module might want to implement hook\_css\_alter() can be found in the Locale module. The Locale module checks to see if the language direction is right-to-left, and if so, finds the related RTL versions of the CSS files and adds them to the page.

# In themes, the main reasons to implement hook\_css\_alter() is to remove or override CSS files provided by modules. An example of this can be found at the bottom of Seven theme’s template.php file (see [Listing 16–43](http://themery.com/dgd7/advanced-theming/css/add-remove-replace#listing-16-43)). Seven chooses to override the stylesheet vertical-tabs.css file provided by core with its own version.

# Listing 16–43. The Seven theme’s hook\_css\_alter() implementation

# <?php

# /\*\*

# \* Implements hook\_css\_alter().

# \*/

# function seven\_css\_alter(&$css) {

# // Use Seven's vertical tabs style instead of the default one.

# if (isset($css['misc/vertical-tabs.css'])) {

# $css['misc/vertical-tabs.css']['data'] = drupal\_get\_path('theme', 'seven') . '/vertical- tabs.css';

# }

# // Use Seven's jQuery UI theme style instead of the default one.

# if (isset($css['misc/ui/jquery.ui.theme.css'])) {

# $css['misc/ui/jquery.ui.theme.css']['data'] = drupal\_get\_path('theme', 'seven') . '/jquery.ui.theme.css';

# }

# }

# CAUTION Overriding a module’s CSS files in .info files (creating an entry with the same CSS file name) will work, but not always in an efficient way. Stylesheets that are defined in .info files will load on every page. Whether or not they are actually needed is never taken into account. This is not the case when using hook\_css\_alter()as you are given the opportunity to make sure the file is set to load before attempting to replace it.

# .info

# hook\_process\_html()

# hook\_css\_alter

# EXERCISE

# MANAGING STYLESHEETS IN YOUR THEME

# In this section, you’ve learned quite a few ways to manipulate CSS files in Drupal’s theme layer. Now you’ll go through the steps again with practical examples.

# EXERCISE A: DEFINE STYLESHEETS FOR ALL PAGES IN THE .INFO FILE

# Begin by creating a new directory in your theme called css in sites/all/themes/mytheme. This step is optional but helpful for theme file organization.

# Create two files inside the css directory called style.css and print.css.

# Open the sites/all/themes/mytheme/mytheme.info and add the following two lines to define the stylesheets so Drupal knows to load them:

# stylesheets[all][] = css/style.css stylesheets[print][] = css/print.css

# Clear the site cache at /admin/config/development/performance. Once you return to the front end of your site, you’ll see that both files have been added.

# EXERCISE B: ADD A CONDITIONAL STYLESHEET FOR IE USING DRUPAL\_ADD\_CSS()

# Create a file inside the css directory called ie.css.

# Create a file in the root of the theme directory called template.php if you haven’t already done so, and make sure to include <?php at the top of the file.

# Use the following code to implement template\_preprocess\_html() and load the IE stylesheet using drupal\_add\_css():

# <?php

# /\*\*

# \* Implements template\_preprocess\_html().

# \*/

# function mytheme\_preprocess\_html(&$vars) {

# // Add conditional stylesheet that targets Internet Explorer 8 and below.

# drupal\_add\_css(path\_to\_theme() . '/css/ie.css', array('weight' => CSS\_THEME, 'browsers' => array('IE' => 'lte IE 8', '!IE' => FALSE), 'preprocess' => FALSE));

# }

# EXERCISE C: ADD A CUSTOM STYLESHEET FOR THE HOMEPAGE USING DRUPAL\_ADD\_CSS()

# You’ll use the $is\_front variable, which already exists, to detect if the home page is being displayed and then add the homepage.css stylesheet. Add this code directly above the conditional stylesheet code you added in Exercise B.

# <?php

# // Add a stylesheet that prints only on the homepage.

# if ($variables['is\_front']) {

# drupal\_add\_css(path\_to\_theme() . '/css/homepage.css', array('weight' => CSS\_THEME));

# }

# EXERCISE D: OVERRIDE AND REMOVE MODULE CSS FILES USING HOOK\_CSS\_ALTER()

# To implement hook\_css\_alter(), you'll need to create a function called mytheme\_css\_alter() in your template.php file. The $css parameter, which is passed by reference, contains all the stylesheets in array format, and you can do what you please with it. The following code shows how to remove the node.css file if it’s set to load.

# <?php

# /\*\*

# \* Implements hook\_css\_alter().

# \*/

# function mythemename\_css\_alter(&$css) {

# // Remove the node.css file.

# if (isset($css['modules/node.css'])) {

# unset($css['modules/node.css']);

# }

# }

# WORKING WITH BASE AND SUBTHEMES

# Chances are you have a certain way you do things. You may tend to structure your markup similarly in all your themes. You might frequently override certain theme functions, or have special way you like to style forms, or maybe you tend to use a certain grid framework for your layout. These are all great reasons to take advantage of Drupal’s base and subtheming functionality.

# Subthemes share a special relationship with their base (parent) theme(s). They inherit template files and assets from their parent themes. This makes them a great tool to help streamline your theming workflow and essentially create your own “frameworks” or “resets” for theming Drupal sites. Of course, you can also use an existing base theme. Drupal offers quite a few base themes, which we’ll tell you more about later in this section.

# CREATING A SUBTHEME

# Both base and subthemes are regular Drupal themes as far as characteristics go, and any theme can be a base theme. The process of creating a subtheme is very straightforward.

# Start by creating the shell of a new theme. Create a directory for it, and create the .info file containing at least the name and core properties.

# In the .info file, add the “base theme” property containing the name of the theme you want to use as a base, like so:

# base theme = basethemename

# If the base theme has regions and/or features defined in the .info file, you’ll need to copy those to the subtheme as well.

# For basic Drupal themes, these three steps are all you’ll need to do to create your subtheme. Once you’ve done this, you’ll be able to enable the theme on the /admin/appearance page. It’s also worth noting that the base theme you are using does not need to be enabled in the UI to function properly.

# Caution Most of the popular contributed base themes require a little more to set up. Themes like Zen, Omega, and Fusion come with a starterkit or starter directory, which you can copy and use to start your subtheme. Make sure you refer to each theme’s README.txt file for full instructions on how to begin using it, as each is different.

# INHERITANCE AND HOW IT WORKS

# You already know that Drupal provides a lot of markup in its modules, and that this markup comes in the form of templates, theme functions, or the Render API. In Drupal themes you have the opportunity to override and take over this behavior. So, technically, you are inheriting it in the first place. Using subthemes allows you to add one more step to the process. When using a parent theme, all of the assets—including template files, CSS files, JavaScript files, theme functions, and pretty much everything in template.php—are inherited.

# CSS, JavaScript, template files, and theme functions defined in a base theme will automatically be available for the subtheme. The subtheme doesn’t have to do anything for this to happen. It just works. Preprocess and process functions will run for both the base and the subtheme, so they can be used in both themes simultaneously without issue. Of course, the subtheme can override anything the base theme has done.

# Some things don’t work this well. Regions are not inherited, and neither are features or theme settings. In order for these to work properly, you’ll have to copy the information from the base theme into the subtheme’s .info file. [Table 16–4](http://themery.com/dgd7/advanced-theming/base-themes/inheritance#table-16-4) shows which assets are automatically inherited and which ones are not.

# Table 16–4. Inheritance of assets from base theme to subthemes

| ASSET | AUTOMATICALLY INHERITED? |
| --- | --- |
| CSS files | Yes |
| JavaScript files | Yes |
| Template files | Yes |
| Theme Screenshot | Yes |
| Regions | No (Default) |
| Theme Settings | No (Default) |
| Favicon (shortcut icon) | No |
| Logo | No |

# FINDING A GOOD BASE THEME

# Thousands of contributed themes are available at http://drupal.org/project/Themes. Unfortunately, Drupal themes have a reputation for being ugly. While there is some truth to that, there are many gems out there; you just need to know what to look for. Themes on drupal.org are sorted by popularity, based on project usage stats, so it is easy to see which themes are the most popular. However, popularity is not always the best measure. There are a few things you should understand when evaluating a contributed Drupal theme.

# Type

# All of the themes on drupal.org are lumped together into one, uncategorized list. As you can see on http://drupal.org/project/themes, a large portion of the themes on the first page are base themes. While any theme can technically be used as a base theme, it’s important to read the project information so you know what to expect. Maintainers will be a lot less inclined to help you with a problem if you’re not using the theme how they intended.

# Maintenance and development status

# Each project has a Maintenance and a Development status which can be viewed on the project page. These will give you a good idea of how the module is supported. If the project has an “Actively maintained” maintenance status and an “Under active development” development status, chances are that the module developer intends to fix bugs and will entertain feature requests made in the issue queue.

# Usage statistics

# On each project page, the project Information section contains the number of reported installations and a link called “View usage statistics” that shows a long term graph and table of this data and how it has changed over time. Usage statistics can be a good indication of whether or not a theme has been well tested. If many people are using it or it shows steady growth, chances are that it’s a better theme.

# Issue queue

# Most projects contain issues queues where users can report bugs and request features. Reading through the issue queue is a good way to gauge the community participation in a project. It is also a great way to learn what bugs the theme may have and how quickly the community and maintainer(s) respond to such issues.

# TIPS FOR CREATING YOUR OWN BASE THEMES

# *Don’t do too much:* It’s important not to make too many assumptions in your base themes. Ask yourself if what you are doing will fit in well on any project you work on. If the answer is no or maybe, it’s likely not a feature you should include in your base theme.

# *Look at contributed themes:* Looking at what other contributed themes have done is one of the best ways to learn. Chances are you’ll find some things you like and some things you don’t from each of them. Don’t be afraid to mix and match.

# *Provide styles for layout and others structural elements:* Take care of things that you consistently do on each project. For example, normalize font sizes, provide CSS resets, and make sure that the general padding and margins are set so blocks and nodes are not on top of each other.

# *Use multiple CSS files:* Aggregation and compression will take care of combining these files automatically, so don’t be afraid of using a few CSS files. This will allow you to easily choose between what you want and don’t want in your subthemes.

# SUSTAINABILITY AND BEST PRACTICES

# Drupal contains many, many template files. For a front-end developer, these are one of your greatest tools in taking over a Drupal theme and turning it into exactly what you need it to be. However, with power comes responsibility. Because working with template files is so easy, it’s also an area where you can get in trouble quickly.

# Most front-end developers experience some frustration working with Drupal’s markup. Because it’s relatively easy to go in and make changes, doing exactly that is often a first reflex. Resist it. While you’ll definitely feel the power and control you have over things, changing too many template files is often the wrong approach. Just because you can change things doesn’t always mean you should.

# START WITH A GOOD BASE

# A great way to ensure minimal template overrides is to define your markup in such a way that it is flexible enough to work in most cases. Think of the major template files like node.tpl.php, views-view.tpl.php and block.tpl.php, for example, as having two purposes. The first is to provide a container and the second is the actual content, which can include any number of different elements inside it. Drupal does this reasonably well to begin with, but there is always room for improvement, and your needs may vary from site to site depending on the design.

# As an example, look at the contents of the block.tpl.php file, shown in [Listing 16–44](http://themery.com/dgd7/advanced-theming/best-practices/start-with-a-good-base#listing-16-44), which is provided by Drupal’s Block module and can be found in modules/block/block.tpl.php. Most blocks, even those produced by other modules, will use this template file to output their contents. There could be a menu inside the block, a few paragraphs in a custom block, a snippet of JavaScript that will load an advertisement, a poll, a user listing, and so many other possibilities.

# Listing 16–44. Default block.tpl.php implementation

# <div id="<?php print $block\_html\_id; ?>" class="<?php print $classes; ?>"<?php print $attributes; ?>>

# <?php print render($title\_prefix); ?>

# <?php if ($block->subject): ?>

# <h2<?php print $title\_attributes; ?>><?php print $block->subject ?></h2>

# <?php endif;?>

# <?php print render($title\_suffix); ?>

# <div class="content"<?php print $content\_attributes; ?>>

# <?php print $content; ?>

# </div>

# </div>

# TIP The Bartik theme uses Drupal’s default block.tpl.php template file. This is easy to determine because the Bartik theme does not include a block.tpl.php file in its directory.

# Using a simple custom block as an example, the template code in [Listing 16–44](http://themery.com/dgd7/advanced-theming/best-practices/start-with-a-good-base#listing-16-44) translates to the output in [Listing 16–45](http://themery.com/dgd7/advanced-theming/best-practices/start-with-a-good-base#listing-16-45).

# Listing 16–45. Block output using the default block.tpl.php implementation

# <div id="block-block-1" class="block block-block first last odd">

# <h2>Block title</h2>

# <div class="content">

# <p>Block content.</p>

# </div>

# </div>

# The resulting code is pretty minimal. In most cases, when creating custom themes, you will not want these to look the same, so you will use CSS to style them differently. It may not be immediately apparent, but there are some potential problem areas to take note of with the default block.tpl.php implementation. Certain design aspects need more flexible markup. Some examples of this include:

# *Grids:* You may choose to lay out your blocks within regions using a CSS grid framework. This will prevent you from adding left and right padding/margins directly to the .block class.

# *Background images:* Your design might require adding multiple background images to achieve a design for the block that is content agnostic. Sounds easy enough, right? The top and tiling background image can be declared in .block, but where can the bottom background image be defined? As soon as you add padding to the .block class itself, you lose the ability to place the second background image on the existing .content class.

# The previous examples are a small taste of what you might encounter while coding a Drupal theme. You may be tempted to take the minimalist markup approach and deal with problems as they arise, and this is where we would stop you! As mentioned, these main template files are responsible for containing many types of content. You don’t want to create a new template file for every different kind just to modify structural aspects. It’s much more sustainable, not to mention easier to code, to create solid and flexible defaults and deal with exceptions as they arise.

# This can be achieved fairly easily by separating structure from content. As shown in [Listing 16–46](http://themery.com/dgd7/advanced-theming/best-practices/start-with-a-good-base#listing-16-46), by simply adding <div class="inner"> to surround the contents, you can solve many potential problems before they arise. In the example of grids, padding can be applied to the <div class="inner">. As for background images, the top background image can be applied to .block, and the bottom can be applied to .block .inner or vice versa.

# Listing 16–46. Modified block.tpl.php to contain a more flexible container structure

# <div id="<?php print $block\_html\_id; ?>" class="<?php print $classes; ?>"<?php print $attributes; ?>>

# <div class="inner">

# <?php print render($title\_prefix); ?>

# <?php if ($block->subject): ?>

# <h2<?php print $title\_attributes; ?>><?php print $block->subject ?></h2> <?php endif;?>

# <?php print render($title\_suffix); ?>

# <div class="content"<?php print $content\_attributes; ?>>

# <?php print $content; ?>

# </div>

# </div>

# </div>

# OVERRIDE TEMPLATE FILES WITH PURPOSE

# While core template files are less likely to change during the course of a major release cycle, there are usually massive changes to template files for each major Drupal release, and contributed modules are a constantly moving target. Template files can change at any time, and sometimes drastically. There are many potential reasons for these changes. A module developer may decide to take a different approach, there might be new features or security updates, or there may be no good reason at all. The point is that once you override a template file by adding it to your theme, you are responsible for maintaining it. This can easily get out of hand if you have too many template files.

# Another thing to remember is that Drupal is a framework. The whole idea of using Drupal is to take advantage of its modularity. Having too many template files in your theme can essentially remove that modularity; once that happens, your theme can become more of a hassle to maintain than all of Drupal and whatever custom modules you have combined. The key to avoiding this problem is to use overrides sparingly and take advantage of the many tools that Drupal provides.

# Just adding the <div class="inner"> as you did in [Listing 16–46](http://themery.com/node/66#listing-16-46) can go a long way in saving you the need to create additional template files. The following tips will help you stay out of trouble when working with templates in Drupal themes:

# *Structure for the majority.* Explore options for handling one-offs separately by using preprocess functions where possible.

# *Take advantage of theme hook suggestions.* When the differences between the markup warrant it, use node--article.tpl.php to style article nodes and use theme\_links\_\_node() to target only node links.

# *Take advantage of CSS classes as arrays.* If all you need is a class, don’t create a new template file. For example, block titles are output in a simple <h2> tag by default. When applying even minimal CSS to .block h2, you run the risk of affecting <h2> tags that may end up inside <div class="content">. Add a class to the title to style against, so you can prevent these issues.

# LEVERAGE DEFAULT CSS CLASSES

# Don’t just rip out or change CSS classes without good reason for doing so. Think about it. While many front-end developers and web designers gasp at the sight of all the CSS classes that Drupal makes available, there really is a purpose to the madness. These classes (especially body classes) not only provide helpful information that guides you through figuring out what is generating the markup and what characteristics the contents of a given <div> might have, but they are designed to give you the opportunity to do a large portion of your theme development within the CSS.

# Keep in mind, especially when using contributed modules, that you will need to update and likely upgrade your site at some point in the future, and that you can’t control the changes that may be made to templates and often to the classes applied inside them. It’s also important to note that modules may rely on classes and certain CSS files, such as system.base.css to be loaded in order to function properly. Of course, you can try to manage these things, but we can report from experience that this can easily turn into a frustrating waste of time. We’re not saying there’s not room for improvement or that you shouldn’t code your site the way you want to. We simply want to make you aware of some of the risks involved when stripping markup down to barebones.

# DO MY CHANGES BELONG IN A MODULE?

# With each new release of Drupal, the theme layer becomes more and more powerful. With the advent of Render API and the ability to use alter hooks in themes, Drupal 7 is packed with more power than ever. As powerful as Drupal themes can be, there are still many things that just do not belong in the theme layer. As you are plugging away coding your awesome Drupal theme, constantly ask yourself these questions:

# Does what you are trying to accomplish require an SQL query? These should never be in a theme. Period.

# Does your task seem particularly difficult to accomplish? Are you completely rebuilding data?

# Are your changes really theme-specific? For example, if you are changing form labels and descriptions, should these be available if you were to disable your theme?

# If the answer to any of these questions is yes, then your changes belong in a module.

# SUMMARY

# Throughout this chapter we’ve covered more of the many different methods you can use to bend Drupal themes to your will. We’ve covered almost everything you’ll need to know to create truly awesome and sustainable themes, including how to:

# Find the variables that are available to you in the theme layer.

# Understand and use preprocess and process functions.

# Use and alter contents of render arrays.

# Theme forms with templates, theme functions, and alter hooks.

# Manage CSS and JavaScript files in your theme.

# Work with base and subthemes.

# It’s easy to become overwhelmed with Drupal’s theme layer in the beginning. Just remember that your themes can be as simple or as complex as you need them or want to be. We hope that you’ll take this knowledge and use it to create awesome Drupal themes, and contribute them back to the community.