MOODLE DEVELOPER DOCUMENTATION

BLOCKS
Jon Papaioannou (pj@uom.gr)
$Id: HOWTO.html,v 1.1 2004/11/19 02:32:48 defacer Exp $

A Step-by-step Guide To Creating Blocks

1. Summary

The present document serves as a guide to developers who want to create their own blocks for use in Moodle. It applies to the 1.5 development version of Moodle (and any newer) ONLY, as the blocks subsystem was rewritten and expanded for the 1.5 release. However, you can also find it useful if you want to modify blocks written for Moodle 1.3 and 1.4 to work with the latest versions (look at Appendix B).

The guide is written as an interactive course which aims to develop a configurable, multi-purpose block that displays arbitrary HTML. It's targeted mainly at people with little experience with Moodle or programming in general and aims to show how easy it is to create new blocks for Moodle. A certain small amount of PHP programming knowledge is still required, though. Experienced developers and those who just want a reference text should refer to Appendix A because the main guide has a rather low concentration of pure information in the text.

2. Basic Concepts

Through this guide, we will be following the creation of an "HTML" block from scratch in order to demonstrate most of the block features at our disposal. Our block will be named "SimpleHTML". This does not constrain us regarding the name of the actual directory on the server where the files for our block will be stored, but for consistency we will follow the practice of using the lowercased form "simplehtml" in any case where such a name is required. Whenever we refer to a file or directory name which contains "simplehtml", it's important to remember that ONLY the "simplehtml" part is up to us to change; the rest is standardized and essential for Moodle to work correctly.

Whenever a file's path is mentioned in this guide, it will always start with a slash. This refers to the Moodle home directory; all files and directories will be referred to with respect to that directory.

3. Ready, Set, Go!

To define a "block" in Moodle, in the most basic case we need to provide just one source code file. We start by creating the directory /blocks/simplehtml/ and creating a file named /blocks/simplehtml/block_simplehtml.php which will hold our code. We then begin coding the block:

```
class CourseBlock_simplehtml extends MoodleBlock {
    function init() {
        $this->title = get_string('simplehtml', 'block_simplehtml');
        $this->content_type = BLOCK_TYPE_TEXT;
        $this->version = 2004111200;
    }
```

```
}
```

The first line is our block class definition; it must be named exactly in the manner shown. Again, only the "simplehtml" part can (and indeed must) change; everything else is standardized.

Our class is then given a small method: init(). This is essential for all blocks, and its function is to set the three class member variables listed inside it. But what do these values actually mean? Here's a more detailed description.

$this->title is the title displayed in the header of our block. We can set it to whatever we like; in this case it's set to read the actual title from a language file we are presumably distributing together with the block. I 'll skip ahead a bit here and say that if you want your block to display NO title at all, then you should set this to any descriptive value you want (but NOT make it empty). We will later see in Part 9 how to disable the title's display.

$this->content_type tells Moodle what kind of content to expect from this block. Here we have two simple choices. Either we set content_type to BLOCK_TYPE_TEXT, which tells Moodle to just take our content and display it on screen as-is; or we set it to BLOCK_TYPE_LIST, which tells Moodle that we want our block to display a nicely formatted list of items with optional icons next to each one. We can use BLOCK_TYPE_TEXT to manually create any content we want (do not be fooled by the name; HTML is allowed in the block's content without restriction) or use BLOCK_TYPE_LIST to easily create a simple menu.

$this->version is the version of our block. This actually would only make a difference if your block wanted to keep its own data in special tables in the database (i.e. for very complex blocks). In that case the version number is used exactly as it's used in activities; an upgrade script uses it to incrementally upgrade an "old" version of the block's data to the latest. We will outline this process further ahead, since blocks tend to be relatively simple and not hold their own private data. In our example, this is certainly the case so we just set $this->version to YYYYMMDD00 and forget about it.

UPDATING: Prior to version 1.5, the basic structure of each block class was slightly different. Refer to Appendix B for more information on the changes that old blocks have to make to conform to the new standard.

4. I Just Hear Static

In order to get our block to actually display something on screen, we need to add one more method. All together now:

```
function get_content() {
    if ($this->content !== NULL) {
        return $this->content;
    }

    $this->content = new stdClass;
    $this->content->text = 'The content of our SimpleHTML block!';
    $this->content->footer = 'Footer here...';

    return $this->content;
}
```

It can't get any simpler than that, can it? Let's dissect this method to see what's going on...

First of all, there is a check that returns the current value of $this->content if it's not NULL; otherwise we proceed with "computing" it. Since the computation is potentially a time-consuming operation and it WILL be called several times for each block (Moodle works that way internally), we take a precaution and include this time-saver.

Supposing the content had not been computed before (it was NULL), we then define it from scratch. The code speaks for itself there, so there isn't much to say. Just keep in mind that we can use HTML both in the text AND in the footer, if we want to.

At this point our block should be capable of being automatically installed in Moodle and added to courses; visit your administration page to install it and after seeing it in action come back to continue our tutorial.

5. Configure That Out

The current version of our block doesn't really do much; it just displays a fixed message. Not very useful. What we 'd really like to do is allow the teachers to customize what goes into the block. This, in block-speak, is called "instance configuration". So let's give our block instance configuration...

First of all, we need to tell Moodle that we want it to provide instance-specific configuration amenities to our block. That's as simple as adding one more method to our block class:

```
function instance_allow_config() {
    return true;
}
```

This small change is enough to make Moodle display an "Edit..." icon in our block's header when we turn editing mode on in any course. However, if you try to click on that icon you will be presented with a notice that complains about the block's configuration not being implemented correctly. Try it, it's harmless.

Well, that makes sense. We told Moodle that we want to have configuration, but we didn't tell it WHAT kind of configuration. To do that, we need to create one more file: /blocks/simplehtml/config_instance.html which as you see has to be named just so. For the moment, copy paste the following into it and save:

```
<table cellpadding="9" cellspacing="0">
<tr valign="top">
    <td align="right">
        <?php print_string('configcontent', 'block_simplehtml'); ?>:
    </td>
    <td>
        <?php print_textarea(true, 10, 50, 0, 0, 'text', $this->config-
>text); ?>
    </td>
</tr>
<tr>
    <td colspan="2" align="center">
        <input type="submit" value="<?php print_string('savechanges') ?>" />
    </td>
</tr>
</table>
<?php use_html_editor(); ?>
```

It isn't difficult to see that the above code just provides us with a wysiwyg-editor-enabled textarea to write our block's desired content in and a submit button to save. But... what's $this->config->text? Well...

Moodle goes a long way to make things easier for block developers. Did you notice that the textarea is actually named "text"? When the submit button is pressed, Moodle saves each and every field it can find in our config_instance.html file as instance configuration data. We can then access that data as $this->config->variablename, where variablename is the actual name we used for our field; in this case, "text". So in essence, the above form just pre-populates the textarea with the current content of the block (as indeed it should) and then allows us to change it.

You also might be surprised by the presence of a submit button and the absence of any <form> element at the same time. But the truth is, we don't need to worry about that at all; Moodle goes a really long way to make things easier for developers! We just print the configuration options we want, in any format we want; include a submit button, and Moodle will handle all the rest itself. The instance configuration variables are automatically at our disposal to access from any of the class methods EXCEPT init().

In the event where the default behavior is not satisfactory, we can still override it. However, this requires advanced modifications to our block class and will not be covered here; refer to Appendix A for more details.

Having now the ability to refer to this instance configuration data through $this->config, the final twist is to tell our block to actually DISPLAY what is saved in is configuration data. To do that, find this snippet in /blocks/simplehtml/block_simplehtml.php:

```
$this->content = new stdClass;
$this->content->text = 'The content of our SimpleHTML block!';
$this->content->footer = 'Footer here...';
```

and change it to:

```
$this->content = new stdClass;
$this->content->text = $this->config->text;
$this->content->footer = 'Footer here...';
```

Oh, and since the footer isn't really exciting at this point, we remove it from our block because it doesn't contribute anything. We could just as easily have decided to make the footer configurable in the above way, too. So for our latest code, the snippet becomes:

```
$this->content = new stdClass;
$this->content->text = $this->config->text;
$this->content->footer = '';
```

After this discussion, our block is ready for prime time! Indeed, if you now visit any course with a SimpleHTML block, you will see that modifying its contents is now a snap.

6. The Specialists

Implementing instance configuration for the block's contents was good enough to whet our apetite, but who wants to stop there? Why not customize the block's title, too?

Why not, indeed. Well, our first attempt is natural enough: let's add another field to /blocks/simplehtml/config_instance.html. Here goes:

```
<tr valign="top">
    <td align="right"><p><?php print_string('configtitle',
'block_simplehtml'); ?>:</td>
    <td><input type="text" name="title" size="30" value="<?php echo $this-
>config->title; ?>" /></td>
</tr>
```

We save the edited file, go to a course, edit the title of the block and... nothing happens! The instance configuration is saved correctly, all right (editing it once more proves that) but it's not being displayed. All we get is just the simple "SimpleHTML" title.

That's not too wierd, if we think back a bit. Do you remember that init() method, where we set $this->title? We didn't actually change its value from then, and $this->title is definitely not the same as $this->config->title (to Moodle, at least). What we need is a way to update $this->title with the value in the instance configuration. But as we said a bit earlier, you can use $this->config in all methods EXCEPT init()! So what can we do?

Let's pull out another ace from our sleeve, and add this small method to our block class:

```
function specialization() {
    $this->title = $this->config->title;
}
```

Aha, here's what we wanted to do all along! But what's with the specialization() method?

This "magic" method has actually a very nice property: it's GUARANTEED to be automatically called by Moodle as soon as our instance configuration is loaded and available (that is, a bit after init() is called). That means before the block's content is computed for the first time, and indeed before ANYTHING is else done with the block. Thus, providing a specialization() method is the natural choice for any configuration data that needs to be acted upon "as soon as possible", as in this case.

Unrelated to the above discussion, we also remove the footer from our block because it doesn't contribute anything; we could just as easily have decided to make the footer configurable in the above way, too. So for our latest code, the snippet

```
$this->content = new stdClass;
$this->content->text = $this->config->text;
$this->content->footer = 'Footer here...';
```

becomes

```
$this->content = new stdClass;
$this->content->text = $this->config->text;
$this->content->footer = '';
```

7. Now You See Me, Now You Don't

Now would be a good time to mention another nifty technique that can be used in blocks, and which comes in handy quite often. Specifically, it may be the case that our block will have something interesting to display some of the time; but in some other cases, it won't have anything

useful to say. (An example here would be the "Recent Activity" block, in the case where no recent activity in fact exists. However in that case the block chooses to explicitly inform you of the lack of said activity, which is arguably useful). It would be nice, then, to be able to have our block "disappear" if it's not needed to display it.

This is indeed possible, and the way to do it is to make sure that after the get_content() method is called, the block is completely void of content. Specifically:

- If $this->content_type equals BLOCK_TYPE_TEXT, then "void of content" means that both $this->content->text and $this->content->footer are each equal to the empty string (").
- If $this->content_type equals BLOCK_TYPE_LIST, then "void of content" means that $this->content->items is an empty array and $this->content->footer is equal to the empty string (").

Refer to Part 3 which explains the init() function for more information on $this->content_type and its significance. Note that the exact value of the block's title and the presence or absence of a hide_header() method do NOT affect this behavior. A block is considered empty if it has no content, irrespective of anything else.

8. We Are Legion

Right now our block is fully configurable, both in title and content. It's so versatile, in fact, that we could make pretty much anything out of it. It would be really nice to be able to add multiple blocks of this type to a single course. And, as you might have guessed, doing that is as simple as adding another small method to our block class:

```
function instance_allow_multiple() {
    return true;
}
```

This tells Moodle that it should allow any number of instances of the SimpleHTML block in any course. After saving the changes to our file, Moodle immediately allows us to add multiple copies of the block without further ado!

There are a couple more of interesting points to note here. First of all, even if a block itself allows multiple instances in the same page, the administrator still has the option of disallowing such behavior. This setting can be set separately for each block from the Administration / Configuration / Blocks page.

And finally, a nice detail is that as soon as we defined an instance_allow_config() method, the method instance_allow_config() that was already defined became obsolete. Moodle assumes that if a block allows multiple instances of itself, those instances will want to be configured (what is the point of same multiple instances in the same page if they are identical?) and thus automatically provides an "Edit" icon. So, you can also remove the whole instance_allow_config() method now without harm. We had only needed it when multiple instances of the block had not been allowed.

9. The Effects of Globalization

Configuring each block instance with its own personal data is cool enough, but sometimes administrators need some way to "touch" all instances of a specific block at the same time. In the

case of our SimpleHTML block, a few settings that would make sense to apply to all instances aren't that hard to come up with. For example, we might want to limit the contents of each block to only so many characters, or we might have a setting that filters HTML out of the block's contents, only allowing pure text in. Granted, such a feature wouldn't win us any awards for naming our block "SimpleHTML" but some tormented administrator somewhere might actually find it useful.

This kind of configuration is called "global configuration" and applies only to a specific block type (all instances of that block type are affected, however). Implementing such configuration for our block is quite similar to implementing the instance configuration. We will now see how to implement the second example, having a setting that only allows text and not HTML in the block's contents.

First of all, we need to tell Moodle that we want our block to provide global configuration by, what a surprise, adding a small method to our block class:

```
function has_config() {
    return true;
}
```

Then, we need to create a HTML file that actually prints out the configuration screen. In our case, we 'll just print out a checkbox saying "Do not allow HTML in the content" and a "submit" button. Let's create the file /blocks/simplehtml/config_global.html which again must be named just so, and copy paste the following into it:

```
<div style="text-align: center;">
<input type="hidden" name="block_simplehtml_strict" value="0" />
<input type="checkbox" name="block_simplehtml_strict" value="1"
  <?php if(!empty($CFG->block_simplehtml_strict)) echo 'checked="checked"';
?> />
<?php print_string('donotallowhtml', 'block_simplehtml'); ?>
<p><input type="submit" value="<?php print_string('savechanges'); ?>" /></p>
</div>
```

True to our block's name, this looks simple enough. What it does is that it displays a checkbox named "block_simplehtml_strict" and if the Moodle configuration variable with the same name is set and not empty (that means it's not equal to an empty string, to zero, or to boolean false) it displays the box as pre-checked (reflecting the current status). Why does it check the configuration setting with the same name? Because the default implementation of the global configuration saving code takes all the variables we have in our form and saves them as Moodle configuration options with the same name. Thus, it's good practice to use a descriptive name and also one that won't possibly conflict with the name of another setting. "block_simplehtml_strict" clearly satisfies both requirements.

The astute reader may have noticed that we actually have TWO input fields named "block_simplehtml_strict" in our configuration file. One is hidden and its value is always 0; the other is the checkbox and its value is 1. What gives? Why have them both there?

Actually, this is a small trick we use to make our job as simple as possible. HTML forms work this way: if a checkbox in a form is not checked, its name does not appear at all in the variables passed to PHP when the form is submitted. That effectively means that, when we uncheck the box and click submit, the variable is not passed to PHP at all. Thus, PHP does not know to update its value to "0", and our "strict" setting cannot be turned off at all once we turn it on for the first time. Not the behavior we want, surely.

However, PHP handles received variables from a form this way: variables are processed in the order in which they appear in the form. If a variable comes up having the same name with an already-processed variable, the new value overwrites the old one. Taking advantage of this, our logic runs as follows: the variable "block_simplehtml_strict" is first unconditionally set to "0". Then, IF the box is checked, it is set to "1", overwriting the previous value as discussed. The net result is that our configuration setting behaves as it should.

To round our bag of tricks up, notice that the use of if(!empty($CFG->block_simplehtml_strict)) in the test for "should the box be checked by default?" is quite deliberate. The first time this script runs, the variable $CFG->block_simplehtml_strict will not exist at all. After it's set for the first time, its value can be either "0" or "1". Given that both "not set" and the string "0" evaluate as empty while the sting "1" does not, we manage to avoid any warnings from PHP regarding the variable not being set at all, AND have a nice human-readable representation for its two possible values ("0" and "1").

Now that we have managed to cram a respectable amount of tricks into a few lines of HTML, we might as well discuss the alternative in case that tricks are not enough for a specific configuration setup we have in mind. Saving the data is done in the method config_save(), the default implementation of which is as follows:

```
function config_save($data) {
    // Default behavior: save all variables as $CFG properties
    foreach ($data as $name => $value) {
        set_config($name, $value);
    }
    return true;
}
```

As can be clearly seen, Moodle passes this method an associative array $data which contains all the variables coming in from our configuration screen. If we wanted to do the job without the "hidden variable with the same name" trick we used above, one way to do it would be by overriding this method with the following:

```
function config_save($data) {
    if(isset($data['block_simplehtml_strict'])) {
        set_config('block_simplehtml_strict', '1');
    }
    else {
        set_config('block_simplehtml_strict', '0');
    }
    return true;
}
```

Quite straightfoward: if the variable "block_simplehtml_strict" is passed to us, then it can only mean that the user has checked it, so set the configuration variable with the same name to "1". Otherwise, set it to "0". Of course, this version would need to be updated if we add more configuration options because it doesn't respond to them as the default implementation does. Still, it's useful to know how we can override the default implementation if it does not fit our needs (for example, we might not want to save the variable as part of the Moodle configuration but do something else with it).

So, we are now at the point where we know if the block should allow HTML tags in its content or not. How do we get the block to actually respect that setting?

We could decide to do one of two things: either have the block "clean" HTML out from the input before saving it in the instance configuration and then display it as-is (the "eager" approach); or have it save the data "as is" and then clean it up each time just before displaying it (the "lazy" approach). The eager approach involves doing work once when saving the configuration; the lazy approach means doing work each time the block is displayed and thus it promises to be worse performance-wise. We shall hence go with the eager approach.

Much as we did just before with overriding config_save(), what is needed here is overriding the method instance_config_save() which handles the instance configuration. The default implementation is as follows:

```
function instance_config_save($data) {
    $data = stripslashes_recursive($data);
    $this->config = $data;
    return set_field('block_instance', 'configdata',
base64_encode(serialize($data)),
                     'id', $this->instance->id);
}
```

This may look intimidating at first (what's all this stripslashes_recursive() and base64_encode() and serialize() stuff?) but do not despair; we won't have to touch any of it. We will only add some extra validation code in the beginning and then instruct Moodle to additionally call this default implementation to do the actual storing of the data. Specifically, we will add a method to our class which will go like this:

```
function instance_config_save($data) {
    // Clean the data if we have to
    global $CFG;
    if(!empty($CFG->block_simplehtml_strict)) {
        $data = strip_tags($data);
    }

    // And now forward to the default implementation defined in the parent
class
    return parent::instance_config_save($data);
}
```

At last! Now the administrator has absolute power of life and death over what type of content is allowed in our "SimpleHTML" block! Absolute? Well... not exactly. In fact, if we think about it for a while, it will become apparent that if at some point in time HTML is allowed and some blocks have saved their content with HTML in, and then the administrator changes the setting to "off", this will only prevent subsequent content changes from having HTML included. Blocks which already have HTML in their content won't lose it!

Following that train of thought, the next stop is realizing that we wouldn't have this problem if we had chosen the lazy approach a while back, because in that case we would "sanitize" each block's content just before it was displayed. The only thing we can do with the eager approach is strip all the tags from the content of all SimpleHTML instances as soon as the admin setting is "HTML off"; but even then, turning the setting back to "HTML on" won't bring back the tags we stripped away. On the other hand, the lazy approach might be slower, but it's more versatile; we can choose whether to strip or keep the HTML before displaying the content, and we won't lose them at all if the admin toggles the setting off and on again. Isn't the life of a developer simple and wonderful?

We will let this part of the tutorial come to a close with the obligatory excercise for the reader: in order to have the SimpleHTML block work "correctly", find out how to strengthen the eager approach to strip out all tags from the existing configuration of all instances of our block, OR go back and implement the lazy approach instead. (Hint: do that in the get_content() method)

UPDATING: Prior to version 1.5, the file "config_global.html" was named simply "config.hmtl". Also, the methods config_save() and config_print() were named handle_config() and print_config() respectively. Upgrading a block to work with Moodle 1.5 involves updating these aspects; refer to Appendix B for more information.

10. Eye Candy

Our block is just about complete functionally, so now let's take a look at some of the tricks we can use to make its behavior customized in a few more useful ways.

First of all, there are a couple of ways we can adjust the visual aspects of our block. First of all, it might be useful to create a block that doesn't display a header (title) at all. You can see this effect in action in the Course Description block that comes with Moodle. This behavior is achieved by, you guessed it, adding one more method to our block class:

```
function hide_header() {
    return true;
}
```

As you see, hiding the title is quite straightforward. One more note here: you cannot just set an empty title inside the block's init() method; it's necessary for each block to have a unique, non-empty title after init() is called so that Moodle can use those titles to differentiate between all of the installed blocks.

Another adjustment we might want to do is instruct our block to take up a certain amount of width on screen. Moodle handles this as a two-part process: first, it queries each block about its preferred width and takes the maximum number as the desired value. Then, the page that's being displayed can choose to use this value or, more probably, bring it within some specific range of values if it isn't already. That means that the width setting is a best-effort settlement; your block can REQUEST a certain width and Moodle will TRY to provide it, but there's no guarantee whatsoever about the end result. As a concrete example, all standard Moodle course formats will deliver any requested width between 180 and 210 pixels, inclusive.

To instruct Moodle about our block's preferred width, we add one more method to the block class:

```
function preferred_width() {
    // The preferred value is in pixels
    return 200;
}
```

This will make our block (and all the other blocks displayed at the same side of the page) a bit wider than standard.

Finally, we can also affect some properties of the actual HTML that will be used to print our block. Each block is fully contained within a TABLE element, inside which all the HTML for that block is printed. We can instruct Moodle to add HTML attributes with specific values to that container. This would be done to either a) directly affect the end result (if we say, assign

bgcolor="black"), or b) give us freedom to customize the end result using CSS (this is in fact done by default as we 'll see below).

The default behavior of this feature in our case will assign to our block's container the id HTML attribute with the value "block_simplehtml" (the prefix "block_" followed by the name of our block, lowercased). We can then use that id to make CSS selectors in our theme to alter this block's visual style (for example, "#block_simplehtml { border: 1px black solid}").

To change the default behavior, we will need to define a method which returns an associative array of attribute names and values. For example, the version

```
function html_attributes() {
    return array(
        'id'          => 'block_'. $this->name(),
        'onmouseover' => 'alert("Mouseover on our block!");'
    );
}
```

will result in a mouseover event being added to our block using JavaScript, just as if we had written the onmouseover="alert(...)" part ourselves in HTML. Note that we actually duplicate the part which sets the id attribute (we want to keep that, and since we override the default behavior it's our responsibility to emulate it if required). And the final elegant touch is that we don't set the id to the hard-coded value "block_simplehtml" but instead use the name() method to make it dynamically match our block's name.

11. Authorized Personnel Only

It's not difficult to imagine a block which is very useful in some circumstances but it simply cannot be made meaningful in others. An example of this would be the "Social Activities" block which is indeed useful in a course with the social format, but doesn't do anything useful in a course with the weeks format. There should be some way of allowing the use of such blocks only where they are indeed meaningful, and not letting them confuse users if they are not.

Moodle allows us to declare which course formats each block is allowed to be displayed in, and enforces these restrictions as set by the block developers at all times. The information is given to Moodle as a standard associative array, with each key corresponding to a course format and defining a boolean value (true/false) that declares whether the block should be allowed to appear in that course format.

The format names we can use are: "weeks", "topics", and "social", which correspond to the three standard course format in Moodle; "site", which refers specifically to the front page of our Moodle installation; and "all", which is a fallback case and applies to everything. We can mix and match between these to create the desired effect. If a name is not present at all as a key in the array, it is assumed to be "false".

For example, to have our block appear ONLY in the site front page, we would use:

```
function applicable_formats() {
    return array('site' => true);
}
```

Since "all" is missing, the block is disallowed from appearing in ANY course format; but then "site" is set to true, so it's explicitly allowed to appear in the site front page. For another example,

if we wanted to allow the block to appear in all course formats EXCEPT social, and also to NOT be allowed at the front page, we would use:

```
function applicable_formats() {
    return array('all' => true, 'social' => false, 'site' => false);
}
```

This time we allow the block to appear in "all" course formats by default, and then explicitly disallow those we want to leave outside. Remember that "all" is used as a fallback case if a specific course format isn't explicitly allowed or disallowed; if the specific course format appears in the array, it overrides "all". The exact order of the array items doesn't make any difference at all, but it is good practice to write them in an order easily understood by a human maintaining the code.

12. Lists and Icons

In this final part of the guide, we will briefly discuss what the differences between blocks of BLOCK_TYPE_TEXT and those of BLOCK_TYPE_LIST are. The vast majority of the techniques and options discussed before apply to any block; the only difference is in the handling of the $this->content variable.

As we have seen, blocks of BLOCK_TYPE_TEXT use two properties of $this->content: "text" and "footer". The text is displayed as-is as the block content, and the footer goes below in smaller font size. Blocks of BLOCK_TYPE_LIST use $this->content->footer in the exact same way, but they ignore $this->content->text.

Instead, Moodle expects such blocks to set two other properties when the get_content() method is called. $this->content->items should be a numerically indexed array containing elements that represent the HTML for each item in the list that is going to be displayed. Usually these items will be links, so being free to write HTML allows us to provide anchor tags as list items. $this->content->icons should also be a numerically indexed array, with exactly as many items as $this->content->items has. Each of these items should be a fully qualified HTML <img> tag, with "src", "height", "width" and "alt" attributes. Obviously, it makes sense to keep the images small and of a uniform size.

Thus, if we were to create a list block, our get_content() method might read:

```
function get_content() {
    if ($this->content !== NULL) {
        return $this->content;
    }

    $this->content = new stdClass;
    $this->content->items = array();
    $this->content->icons = array();
    $this->content->footer = 'Footer here...';

    $this->content->items[] = '<a href="some_file.php">Menu Option 1</a>';
    $this->content->icons[] = '<img src="images/icons/1.gif" width="16"
height="16" alt="" />';

    // Add more list items here

    return $this->content;
}
```

The end result when such a block is printed will be a list of items with the corresponding icons before each one, and an optional footer below. For a real world block of this type, there is no better example than the "admin" block which illustrates all of the above.

Appendix A: Reference

This Appendix will discuss the base class MoodleBlock from which all other block classes derive, and present each and every method that can be overridden by block developers in detail. Methods that should NOT be overridden are explicitly referred to as such. After reading this Appendix, you will have a clear understanding of every method which you should or could override to implement functionality for your block.

The methods are divided into three categories: those you may use and override in your block, those that you may NOT override but might want to use, and those that should NEITHER be used NOR overridden. In each category, methods are presented in alphabetical order.

- Methods you can freely use and override:
    - applicable_formats

```
o    function applicable_formats() {
o        // Default case: the block can be used in all course types
o        return array('all' => true);
}
```

      This method allows you to control which formats your block can be added to. Currently "formats" refers to course formats only, but in the future it may also include other kinds of pages than courses as Moodle is developed. You should return an array with keys being course format names and values being boolean true or false, denoting if your block is to be allowed in each specific format.

      Valid options for format names are: "social", "topics", "weeks" (referring to the three standard course formats), "site" (referring to the front page) and "all" (this will be used for those formats you have not explicitly allowed or disallowed).

    - config_print

```
o    function config_print() {
o        // Default behavior: print the config_global.html file
o        // You don't need to override this if you're satisfied with the
above
o        if (!$this->has_config()) {
o            return false;
o        }
o        global $CFG, $THEME;
o        print_simple_box_start('center', '', $THEME->cellheading);
o        include($CFG->dirroot.'/blocks/'. $this->name()
.'/config_global.html');
o        print_simple_box_end();
o        return true;
}
```

      This method allows you to choose how to display the global configuration screen for your block. Override it if you need something much more complex than the default implementation allows you to do. However, keep these points in mind:

        1. If you save your configuration options in $CFG, you will probably need to use global $CFG; before including any HTML configuration screens.
        2. Whatever you do output from config_print(), it will be enclosed in a HTML form automatically. You only need to provide a way to submit that form.

You should return a boolean value denoting the success or failure of your method's actions.

```
o  config_save
o      function config_save($data) {
o          // Default behavior: save all variables as $CFG properties
o          // You don't need to override this if you 're satisfied with the
   above
o          foreach ($data as $name => $value) {
o              set_config($name, $value);
o          }
o          return true;
   }
```

This method allows you to override the storage mechanism for your global configuration data. The received argument is an associative array, with the keys being setting names and the values being setting values. The default implementation saves everything as Moodle $CFG variables.

Note that what you receive as an argument will NOT be all of the POST data; Moodle will automatically strip out some bits it added itself (such as "sesskey"), so it's quite safe to save everything you receive.

You should return a boolean value denoting the success or failure of your method's actions.

```
o  get_content
o      function get_content() {
o          // This should be implemented by the derived class.
o          return NULL;
   }
```

This method should, when called, populate the $this->content variable of your block. Populating the variable means:

EITHER
defining $this->content->text and $this->content->footer if your block is of type BLOCK_TYPE_TEXT. Both of these should be strings which can contain arbitrary HTML.

OR
defining $this->content->items, $this->content->icons and $this->content->footer if your block is of type BLOCK_TYPE_LIST. The first two should be numerically indexed arrays with the same number of elements. $this->content->items can contain arbitrary HTML while $this->content->icons should only contain fully-qualified HTML img tags. $this->content->footer is a string, as above.

If you set ALL of these variables to their default "empty" values (empty arrays for the arrays and empty strings for the strings), the block will NOT be displayed at all except to editing users. This is a good way of having your block NOT displayed if there is no reason for it.

Your function should return the fully constructed $this->content variable. You should also include a check that, if this variable is not exactly equal to NULL, returns the

existing value instead of calculating it once more. If you fail to do this, Moodle will suffer a performance hit.

- has_config

```
    function has_config() {
        return false;
}
```

This method should return a boolean value taht denotes whether your block wants to present a configuration interface to site admins or not. The configuration that this interface offers will impact all instances of the block equally. To fully implement this configuration you will need to take some additional steps apart from overriding this method; refer to the full guide for more information.

- hide_header

```
    function hide_header() {
        //Default, false--> the header is shown
        return false;
}
```

This method should return a boolean value that denotes whether your block wants to hide its title part. Thus, if you override it to return true, your block will not display a title unless the current user is in editing mode.

- html_attributes

```
    function html_attributes() {
        // Default case: just an id for the block, with our name in it
        return array('id' => 'block_'. $this->name());
}
```

This method should return an associative array of HTML attributes that will be given to your block's container element when Moodle sends the page to the user's browser. No sanitization will be performed in these elements at all. You should always include those attributes that the default implementation sets, but this is not strictly obligatory and you can choose to ignore it if there is reason to (is there?).

- instance_allow_config

```
    function instance_allow_config() {
        return false;
}
```

This method should return a boolean value. True indicates that your block wants to have per-instance configuration, while false means it does not. If you do want to implement instance configuration, you will need to take some additional steps apart from overriding this method; refer to the full guide for more information.

This method's return value is irrelevant if instance_allow_multiple() returns true; it is assumed that if you want multiple instances then each instance needs its own configuration.

- instance_allow_multiple

```
    function instance_allow_multiple() {
        // Are you going to allow multiple instances of each block?
        // If yes, then it is assumed that the block WILL USE per-
instance configuration
```

```
o         return false;
   }
```

This method should return a boolean value, indicating whether you want to allow multiple instances of this block in a single course or not. If you do allow multiple instances, it is assumed that you will also be providing per-instance configuration for the block. Thus, you will need to take some additional steps apart from overriding this method; refer to the full guide for more information.

```
o  instance_config_print
o     function instance_config_print() {
o         // Default behavior: print the config_instance.html file
o         // You don't need to override this if you're satisfied with the
   above
o         if (!$this->instance_allow_multiple() && !$this-
   >instance_allow_config()) {
o             return false;
o         }
o         global $CFG, $THEME;
o
o         if (is_file($CFG->dirroot .'/blocks/'. $this->name()
   .'/config_instance.html')) {
o             print_simple_box_start('center', '', $THEME->cellheading);
o             include($CFG->dirroot .'/blocks/'. $this->name()
   .'/config_instance.html');
o             print_simple_box_end();
o         } else {
o             notice(get_string('blockconfigbad'),
   str_replace('blockaction=', 'dummy=', qualified_me()));
o         }
o
o         return true;
   }
```

This method allows you to choose how to display the instance configuration screen for your block. Override it if you need something much more complex than the default implementation allows you to do. Keep in mind that whatever you do output from config_print(), it will be enclosed in a HTML form automatically. You only need to provide a way to submit that form.

You should return a boolean value denoting the success or failure of your method's actions.

```
o  instance_config_save
o     function instance_config_save($data) {
o         $data = stripslashes_recursive($data);
o         $this->config = $data;
o         return set_field('block_instance', 'configdata',
   base64_encode(serialize($data)),
o                          'id', $this->instance->id);
   }
```

This method allows you to override the storage mechanism for your instance configuration data. The received argument is an associative array, with the keys being setting names and the values being setting values.

The configuration must be stored in the "configdata" field of your instance record in the database so that Moodle can auto-load it when your block is constructed. However, you may still want to override this method if you need to take some additional action apart from saving the data. In that case, you really should do what data processing you want and then call parent::instance_config_save($data) with your new $data array. This will keep your block from becoming broken if the default implementation of instance_config_save changes in the future.

Note that what you receive as an argument will NOT be all of the POST data; Moodle will automatically strip out some bits it added itself (such as "sesskey"), so it's quite safe to save everything you receive.

You should return a boolean value denoting the success or failure of your method's actions.

- preferred_width

```
    function preferred_width() {
        // Default case: the block wants to be 180 pixels wide
        return 180;
}
```

This method should return an integer value, which is the number of pixels of width your block wants to take up when displayed. Moodle will try to honor your request, but as this is actually up to the implementation of the format of the page your block is being displayed in, there is no guarantee. You might actually get exactly what you want or any other width.

Most display logic at this point allocates the maximum width requested by the blocks that are going to be displayed, bounding it both downwards and upwards to avoid having a bad-behaving block break the format.

- refresh_content

```
    function refresh_content() {
        // Nothing special here, depends on content()
        $this->content = NULL;
        return $this->get_content();
}
```

This method should cause your block to recalculate its content immediately. If you follow the guidelines for get_content, which say to respect the current content value unless it is NULL, then the default implementation will do the job just fine.

You should return the new value of $this->content after refreshing it.

- specialization

```
    function specialization() {
        // Just to make sure that this method exists.
}
```

This method is automatically called by the framework immediately after your instance data (which includes the page type and id and all instance configuration data) is loaded from the database. If there is some action that you need to take as soon as this data becomes available and which cannot be taken earlier, you should override this method.

This method should not return anything at all.

- Methods which you should NOT override but may want to use:
  - get_content_type
  - get_title
  - get_version
  - name
- Methods which you should NOT override and NOT use at all:
  - _self_test
  - add_edit_controls
  - load_instance
  - print_block
  - print_shadow

Appendix B: Differences in the Blocks API for Moodle versions prior to 1.5

This Appendix will discuss what changes in the Blocks API were introduced by Moodle 1.5 and what steps developers need to take to update their blocks to be fully compatible with Moodle 1.5. Unfortunately, with these changes backward compatibility is broken; this means that blocks from Moodle 1.4 will never work with 1.5 and vice versa.

1. Constructor versus init()

In Moodle 1.4, in each block class it was mandatory to define a constructor which accepted a course data record as an argument (the example is from the actual Online Users block):

```
function CourseBlock_online_users ($course) {
    $this->title = get_string('blockname','block_online_users');
    $this->content_type = BLOCK_TYPE_TEXT;
    $this->course = $course;
    $this->version = 2004052700;
}
```

In contrast, Moodle 1.5 does away with the constructor and instead requires you to define an init() method that takes no arguments:

```
function init() {
    $this->title = get_string('blockname','block_online_users');
    $this->content_type = BLOCK_TYPE_TEXT;
    $this->version = 2004111600;
}
```

Of course, this leaves you without access to the $course object, which you might actually need. Since that's probably going to be needed inside get_content(), the way to retrieve it is by using this code:

```
$course = get_record('course', 'id', $this->instance->pageid);
```

If you are going to need access to $course from inside other methods in addition to get_content(), you might fetch the $course object inside the specialization() method and save it as a class variable for later use, in order to avoid executing the same query multiple times:

```
function specialization() {
    $this->course = get_record('course', 'id', $this->instance->pageid);
}
```

2. Blocks with configuration

In Moodle 1.4, blocks could only have what are now (in Moodle 1.5) called "global configuration" options, to differentiate from the new "instance configuration" options. If your block has support for configuration, you will need to take these steps:

1. Rename your config.html file to config_global.html
2. Edit the newly renamed file and REMOVE the <form> tag (Moodle now wraps your configuration in a form automatically)
3. If you are using any HTML <input> tags other than those that directly affect your configuration (for example, "sesskey"), REMOVE those too (Moodle will add them automatically as required)
4. If you have overridden print_config(), rename your method to config_print()
5. If you have overridden handle_config(), rename your method to config_save()

That's everything; your block will now be ready for use in Moodle 1.5!