**Models**

**Setup**

Here's the list of files required for the desired setup:

* app/etc/modules/Training\_Overriding.xml: It's a file used to enable our custom module.
* app/code/local/Training/Overriding/etc/config.xml: It's a module configuration file in which we'll set up model class overriding using certain tags as per the Magento conventions.
* app/code/local/Training/Overriding/Model/Category.php: It's a model class of our custom module which will override the base model class.

**Creating Files and Folders: Custom Module**

First, we need to create a module enabler file. Create a file "app/etc/modules/Training\_Overriding.xml" and paste the following contents in that file. We've used Training as our module namespace and Overriding as our module name. It'll enable our "Overriding" module by default.

|  |  |
| --- | --- |
|  | <?xml version="1.0"?>  <config>    <modules>      <Training\_Overriding>        <active>true</active>        <codePool>local</codePool>      </Training\_Overriding>    </modules>  </config> |

Next, we need to create a module configuration file. Create "app/code/local/Training/Overriding/etc/config.xml" and paste the following contents in that file.

|  |  |
| --- | --- |
|  | <?xml version="1.0"?>  <config>    <modules>      <Training\_Overriding>        <version>1.0</version>      </Training\_Overriding>    </modules>      <global>      <models>        <catalog>          <rewrite>            <category>Training\_Overriding\_Model\_Category</category>          </rewrite>        </catalog>      </models>    </global>  </config> |

First, we've defined a module version number using the <version> tag. After that, the <catalog> and <rewrite> tags are used to inform the Magento overriding system that we're going to override a "model" of the "Catalog" core module.

Next, the <category> tag is used to define a model identity which will be overridden by the Training\_Overriding\_Model\_Category class. It's mapped to a model file "Category.php" under the "Model" directory of the Catalog module. An important thing to notice here is that we're following a directory structure similar to the core module. Although that's not strictly necessary, it's better than using a different directory structure, because it helps to maintain readability.

Finally, the only remaining thing is to define a model class Training\_Overriding\_Model\_Category. Let's create a model file "app/code/local/Training/Overriding/Model/Category.php" and paste the following contents in that file.

|  |  |
| --- | --- |
|  | <?php  /\*\*   \* Catalog category model   \*   \* @category   Training   \* @package    Training\_Overriding   \*/  class Training\_Overriding\_Model\_Category extends Mage\_Catalog\_Model\_Category  {    public function getProductCollection()    {      // Include your custom code here!        $collection = Mage::getResourceModel('catalog/product\_collection')        ->setStoreId($this->getStoreId())        ->addCategoryFilter($this);        return $collection;    }  } |

We've defined an Training\_Overriding\_Model\_Category class which extends the core Mage\_Catalog\_Model\_Category model class of the "Catalog" module. Thus, you can override every method of the base class and create new methods if necessary.

In the above example, the getProductCollection method is overridden so it'll be called instead of the method defined in the core model class! You can modify the model code as per your requirements.

When you override any methods in models, you should make sure that the data type of the return value of that method matches with the data type of the base class method. Since model methods are called from several core modules, we should make sure that it doesn't break other features!

Although this is a very basic example, it serves the purpose of overriding the core model, and you can extend it as per your custom requirements.

**Blocks**

**Setup**

Here's a list of the files required for the desired setup:

* app/code/local/Training/Overriding/Block/Category/View.php: It's our new block file which will replace a core block file.

In "app/code/local/Training/Overriding/etc/config.xml", paste the following contents in that file.

|  |  |
| --- | --- |
|  | <?xml version="1.0"?>  <config>    <modules>      <Training\_Overriding>        <version>1.0</version>      </Training\_Overriding>    </modules>      <global>  <models>        <catalog>          <rewrite>            <category>Training\_Overriding\_Model\_Category</category>          </rewrite>        </catalog>      </models>      <blocks>        <catalog>          <rewrite>            <category\_view>Training\_Overriding\_Block\_Category\_View</category\_view>          </rewrite>        </catalog>      </blocks>    </global>  </config> |

The <category\_view> tag is used to define a block identity which will be overridden by the Training\_Overriding\_Block\_Category\_View class. It's mapped to a block file "Category/View.php" under the "Block" directory of the Catalog module. The important thing to notice here is that we're following a directory structure similar to the core module. Although that's not strictly necessary, it's preferred over the different directory structure to maintain readability.

Finally, the only remaining thing is to define a block class Training\_Overriding\_Block\_Category\_View. Let's create a block file "app/code/local/Training/Overriding/Block/Category/View.php" and fill it up using the following contents.

|  |  |
| --- | --- |
|  | <?php  /\*\*   \* Category View block   \*   \* @category Training   \* @package Training\_Overriding   \*/  class Training\_Overriding\_Block\_Category\_View extends Mage\_Catalog\_Block\_Category\_View  {    public function getProductListHtml()    {      // Include your custom code here      return $this­>getChildHtml('product\_list');    }  } |

We've defined the Training\_Overriding\_Block\_Category\_View class which extends the core Mage\_Catalog\_Block\_Category\_View block class. Thus, you can override every method of the base class and create new methods if necessary.

In the above example, the getProductListHtml method is overridden so it'll be called on the category page view! You can alter the code as per your requirements.

Although it's a very basic example, it serves the purpose of overriding the core block, and you can tweak it to make things more complex and create something more useful!

**Controllers**

**Setup**

Here's the list of files required for the desired setup:

* app/code/local/Training/Overriding/controllers/ProductController.php: It's a custom controller class file which we'll use to override the core "Product" controller class.

In "app/code/local/Training/Overriding/etc/config.xml", paste the following contents in that file.

|  |  |
| --- | --- |
|  | <?xml version="1.0"?>  <config>    <modules>      <Training\_Overriding>        <version>1.0</version>      </Training\_Overriding>    </modules>  <frontend>      <routers>        <catalog>          <args>            <modules>              <Training\_Overriding before="Mage\_Catalog">Training\_Overriding</Training\_Overriding>            </modules>          </args>        </catalog>      </routers>    </frontend>      <global>  <models>        <catalog>          <rewrite>            <category>Training\_Overriding\_Model\_Category</category>          </rewrite>        </catalog>      </models>      <blocks>        <catalog>          <rewrite>            <category\_view>Training\_Overriding\_Block\_Category\_View</category\_view>          </rewrite>        </catalog>      </blocks>    </global>  </config> |

The <catalog> tag is the front controller router name of the "Catalog" core module. It informs the routing system that we want to override the controllers of the "Catalog" module.

Finally, the <Training\_Overriding before="Mage\_Catalog">Training\_Overriding</Training\_Overriding> tag is used to inform the Magento routing system that it should load your custom module's controller files if available. Of course, it'll load the core files of the "Catalog" module if it isn't able to find them under your module.

Finally, the only remaining thing is to define a controller class Training\_Overriding\_ProductController. Let's create a controller file "app/code/local/Training/Overriding/controllers/ProductController.php" and paste the following contents in that file.

|  |  |
| --- | --- |
|  | <?php  /\*\*   \* Product controller   \*   \* @category   Training   \* @package    Training\_Overriding   \*/  require\_once 'Mage/Catalog/controllers/ProductController.php';  class Training\_Overriding\_ProductController extends Mage\_Catalog\_ProductController  {    /\*\*     \* Product view action     \*/    public function viewAction()    {      // Include your custom code here!        // Get initial data from request      $categoryId = (int) $this->getRequest()->getParam('category', false);      $productId  = (int) $this->getRequest()->getParam('id');      $specifyOptions = $this->getRequest()->getParam('options');        // Prepare helper and params      $viewHelper = Mage::helper('catalog/product\_view');        $params = new Varien\_Object();      $params->setCategoryId($categoryId);      $params->setSpecifyOptions($specifyOptions);        // Render page      try {        $viewHelper->prepareAndRender($productId, $this, $params);      } catch (Exception $e) {        if ($e->getCode() == $viewHelper->ERR\_NO\_PRODUCT\_LOADED) {          if (isset($\_GET['store'])  && !$this->getResponse()->isRedirect()) {            $this->\_redirect('');          } elseif (!$this->getResponse()->isRedirect()) {            $this->\_forward('noRoute');          }        } else {          Mage::logException($e);          $this->\_forward('noRoute');        }      }    }  } |

We've defined an Training\_Overriding\_ProductController class which extends the core Mage\_Catalog\_ProductController controller class of the Catalog module. The important thing to note here is that we've included the core controller file "Mage/Catalog/controllers/ProductController.php" before the class definition as Magento won't include it automatically.

In the above example, we've overridden the viewAction method of the Mage\_Catalog\_ProductController class. It means that whenever a product detail page is viewed in the front-­end, it'll call the viewAction method defined in your custom controller class. So you can modify the code defined in that method as per your requirements.