mod_perl

DEVELOPER’S COOKBOOK
CHAPTER 10
Object-Oriented mod_perl

Introduction

If you are not already convinced of the merits of mod_perl, then this chapter ought to set you straight. On top of the normal bag of goodies mod_perl provides by granting access to the Apache API, it also offers the ability to extend the Apache framework using object-oriented design. In fact, if you are unfamiliar with object-oriented programming and are reading this chapter with some trepidation, fear not—you have already been (silently) exposed to most of the basic concepts of object-oriented Perl techniques in earlier chapters.

Take the all-important variable $r, typically captured at the beginning of a handler. $r is an example of an object—the Apache request object referred to frequently in previous chapters. Objects always belong to a class. Again, in the case of $r the object belongs to the Apache class, which gives it the ability to call a certain set of methods. A subroutine belonging to a class is called a method. Some methods can be called as class methods and be used without an intervening object, such as Apache->request, whereas others must be called as object methods, such as $r->send_http_header().
Much of the following material is, therefore, putting concepts already introduced in the context of Perl's object-oriented model, and then describing how you can use object-oriented techniques within mod_perl in very customizable, dramatic, and powerful ways.

Although recipes in this chapter offer a brief introduction into the fundamentals of object-oriented programming with Perl, they are by no means meant to be an all-encompassing reference. If you are unfamiliar with the syntax and mechanisms of Perl's object-oriented programming model, the recipes here ought to be enough to give you a place to start and whet your appetite somewhat. For those who are ready to take advantage of the Apache framework using object-oriented Perl, the requisite hooks into mod_perl's object-oriented mechanisms are here for your enjoyment.

10.1. Class and Object Creation

You want to create a class or object.

**Technique**
Follow the basic steps outlined here for creating an object-oriented package.

```
# Place this code in a file named Cookbook/Dinghy.pm

package Cookbook::Dinghy;

use strict;

sub new {
    my ($class, %args) = @_;

    my ($class, args) = @_; # Should this be ($class, %args) = @_;?

    return bless { _capacity => 2,
        color => $args{color} || 'navy',
        count => $args{count} || 0,
    }, $class;
}

sub check_load {

    sub check_load {
```
my $self = shift;

die "We sunk" if ($self->{count} > $self->{_capacity});
}
1;

Then use the newly created Cookbook::Dinghy package by creating and executing the following simple perl script.

```perl
#!/usr/bin/perl -w
use Cookbook::Dinghy;
use strict;

my $lifeboat = Cookbook::Dinghy->new(count => 2);

$lifeboat->check_load;

print "We are still floating.\n"
```

**Comments**

Object-oriented techniques offer a number of distinct advantages to writing function-driven code in certain situations—easier maintainability, less duplication for common routines, faster development time, encapsulation of related code, greater scalability, and so on. In this recipe we just describe the basics for creating an object-oriented module, along with certain conventions; the remaining recipes in this chapter discuss how to use these techniques in the mod_perl environment.

We first need to get some jargon out of the way. A *class* in Perl is simply a package, which is just a fancy way of grouping a related set of subroutines and variables into a common namespace. Typically one package identifier exists per Perl module, but this is a convention more than a rule. An *object* is a data structure reference (often a hash reference) associated with a package namespace by using Perl's `bless()` function. A *method* is a Perl subroutine that is called with a special syntax using the `->` operator. The result of this syntax is that the first argument passed to the method is either the name of the calling class (for a class method) or an object (in the case of an object method). A *constructor* is a class method that returns an object so this whole object-oriented process can get started. That wasn't so bad, was it? Now we can discuss in more detail where these concepts appear in the preceding example.
The example Cookbook::Dinghy class is created with the initial package declaration at the beginning of the module. To use the class, an object must first be created; this is done in the example script via

```perl
use Cookbook::Dinghy;

my $lifeboat = Cookbook::Dinghy->new(count => 2);
```

which associates the variable $lifeboat with the object. The constructor used to create the object, conventionally called `new()`, has the following basic form:

```perl
sub new {
    my ($class, %args) = @_;

    return bless {
        _capacity => 2,
        color     => $args{color} || 'navy',
        count     => $args{count} || 0,
    }, $class;
}
```

This returns the object—in this case, a hash reference that has been `bless()`ed into the class Cookbook::Dinghy. The named arguments `count` and `color` are stored within the hash. Note that we handle any missing arguments by setting default values. The default `color` is `navy` and the default `count` is `0`. Subroutines, in an object-oriented setting, are called methods and are called with the special syntax `$object->method()`, just as was done for the `new()` method. As with ordinary functional subroutines, arguments can be passed into methods and received through the standard argument list. However, an important difference to remember is that for methods the first argument passed in is the static class name or object. Conventionally this is assigned the variable name `$self`, and can be captured within a method as

```perl
sub check_load {
    my $self = shift;

    die "We sunk" if ($self->{count} > $self->{_capacity});
}
```

Note in the preceding that `$self->{count}` and `$self->{_capacity}` are accessing the attributes of the object; they are not method calls but indirect dereferencing of the
underlying data structure. Also note the initial _ in the _capacity attribute. Although not required, this Perl convention signifies to the rest of the world that this particular attribute is considered to be private data. We wouldn’t want anyone overstuffing our dinghy by overriding the base capacity of the class, now would we?

Now that we have some mechanics of creating object-oriented modules under our belts, we turn to their use in a mod_perl context, which is actually much easier than you might expect.

### 10.2. Method Inheritance

You want to subclass an existing class so that you can inherit its methods.

**Technique**

Add the parent class to the @ISA array for your class, then add or redefine the class methods as appropriate.

```perl
package Cookbook::SSI;

use Apache::SSI;

use HTTP::Request;
use LWP::UserAgent;

use strict;

@Cookbook::SSI::ISA = qw(Apache::SSI);

sub ssi_include {
    # Re-implement the 'include' SSI tag so that its output
    # can be filtered using Apache::Filter.
    # We only handle <!--#include virtual="file"--> tags for now.

    my ($self, $args) = @_;

    return $self->error('Include must be of type 'virtual')
        unless $args->{virtual};

```
# Create a self-referential URI.
my $uri = Apache::URI->parse(Apache->request);

# Now, add the URI path based on the SSI tag.
if ($args->{virtual} =~ m!^/!) {
    # Path is absolute.
    $uri->path($args->{virtual});
} else {
    # Path is relative to current document.
    my ($base) = $uri->path =~ m!(.*/)!;
    $uri->path($base . $args->{virtual});
}

my $request = HTTP::Request->new(GET => $uri->unparse);

my $response = LWP::UserAgent->new->request($request);

return $self->error("Could not Include virtual URL") unless $response->is_success;

# Return the content of the request back to Apache::SSI.
return $response->content;

1;

Comments
One of the nicest features of object-oriented design is that you can subclass existing classes, changing only the methods whose features do not fit your needs. In Perl, inheritance is controlled by adding parent classes to the @ISA array for your package. You have a number of ways to declare this array; you may see code that uses the newer our declaration, or the older use vars construct, as well as using the fully qualified package variable as shown earlier. Any of the three forms will get the job done. The end result is a new class that can act as a drop-in replacement for the original class, with all of its previous functionality intact except for the methods you explicitly choose to add or change. Here, we chose to subclass Apache::SSI, the CPAN module, in accordance with its published interface, and override only one aspect of its functionality.
The Apache::SSI package provides a Perl implementation of mod_include with one important distinction: Apache::SSI can receive output from other content handlers when used in conjunction with the Apache::Filter module. This gives the programmer the ability to dynamically generate content containing SSI tags and still have those tags properly parsed by a Server Side Include engine—a feat currently impossible using mod_cgi and mod_include but actually quite simple using some mod_perl extensions from CPAN. See Recipe 15.4 for a deeper explanation of Apache::Filter and filtered content generation.

Although this feature of Apache::SSI is a huge win for Web programmers, the current implementation suffers from a few limitations. Namely, Apache::SSI must be the last filter in the chain when using the popular exec and include SSI tags. In both cases, this limitation is due to the fact that Apache::SSI uses the Apache subrequest mechanism to process the tags and generate content. As you recall from Recipe 3.15, when the content-generation phase for a subrequest is run the content gets sent directly to the client. Even though using a subrequest is more efficient than a full request to the same server (and relies on no other third-party modules) using a subrequest subverts any attempt to gather the output of Apache::SSI and pass it along to another handler, such as Apache::Compress.

Fortunately, Apache::SSI has yet another advantage over mod_include: It presents an object-oriented interface that is designed to be subclassed, so we can override only the methods that require tinkering. Instead of creating a subrequest, like in the standard Apache::SSI implementation, we use a combination of Apache::URI and the LWP suite as described in Recipe 5.7. First, we generate a self-referential URI and use this to issue a request to the file specified through the include SSI tag. After capturing the content from the file, we return the content back to Apache::SSI, where it can be operated on by filters farther down the chain.

For our Cookbook::SSI class, just part of the include tag is handled for the sake of simplicity; implementing exec, the file argument to include, and variable substitution is left as an exercise for the reader. The end result, however, is that the new Cookbook::SSI class can be used as a drop-in replacement for Apache::SSI wherever the new functionality is desired.

Actually, there are many advantages to using the inheritance model we have described here. Probably the most important is that a class that allows itself to be properly subclassed will pass along all its methods to the subclass. For instance, the following four lines of code (known as the empty subclass test) are sufficient for us to use Cookbook::SSI in place of Apache::SSI throughout our httpd.conf and still maintain native Apache::SSI functionality:
package Cookbook::SSI;

use Apache::SSI;

@Cookbook::SSI::ISA = qw(Apache::SSI);

1;

The implications of this are pretty important when it comes to building scalable applications that are maintainable: By inheriting the methods from an existing class the overall maintenance of the application is kept to a minimum, because you only have to alter code in a single place for it to take effect everywhere. This is especially beneficial when using third-party software such as the modules found on CPAN, which is another reason Apache::SSI makes for a good example. By inheriting from Apache::SSI, instead of modifying the core code to meet our needs, we can allow the module author (and other open-source participants) to maintain the SSI engine and other features we are not interested in altering—a bug fix in Apache::SSI does not require applying a patch to a local CVS repository to integrate the change with existing production code. The result is a much more robust application framework.

10.3. Creating Method Handlers

You want to be able to create mod_perl handlers that take advantage of object-oriented design.

Technique

Follow a few basic steps to make your handler into a method handler.

First, modify your handler by adding ($$) as a function prototype, and adjusting the code that reads the input variables to accept a class, like this:

```perl
sub handler ($$) {
    my ($self, $r) = @_;

    # Continue along...
}
```
Then preload your new method handler by adding a `PerlModule` directive to your `httpd.conf`:

```plaintext
PerlModule Cookbook::Dinghy
```

Finally, change your `httpd.conf` to specifically invoke the handler using an object-oriented syntax:

```plaintext
<Location /pleasure-craft>
    SetHandler perl-script
    PerlHandler Cookbook::Dinghy->handler
</Location>
```

**Comments**

Yet another great benefit of becoming familiar with the mod_perl handler API is the ability to leverage the power of object-oriented programming techniques within your handlers. Although scripts that run under `Apache::Registry` force you into a functional programming model, using the handler API allows you the freedom to choose to take advantage of Perl's object-oriented programming features, should you ever find the need.

Handlers that are programmed around the object-oriented model are called *method handlers*. In addition to the steps outlined earlier, you will have to build mod_perl with `PERL_METHOD_HANDLERS=1` or `EVERYTHING=1` to take advantage of method handlers.

Actually, if you flip back to any handler in this book you will see that it is practically begging to be put under Perl's object-oriented control—handlers themselves are already subroutines contained within a package. So, if you have created a mod_perl handler, you have also created a class with at least one method without even knowing it! The only thing missing is to make sure that mod_perl knows that your handler is a method handler, expecting a class name as its first argument instead of the Apache request object.

Every time mod_perl calls a handler subroutine it checks to see whether the subroutine wants to be called as an ordinary subroutine or as a class method. The trigger that mod_perl uses to recognize that a handler is really a method handler is the subroutine's use of the `$$` prototype. When mod_perl sees that the `handler()` subroutine is prototyped to accept two scalar arguments it will invoke the subroutine using Perl's object-oriented syntax, and your handler will be passed both the `handler()` method's calling class and the Apache request object. Because your `handler()`
subroutine is already a method residing in the class defined by the package identifier for your module, you now have all the components you need to start using Perl's object-oriented features.

As with ordinary handlers, mod_perl assumes the handler() method if the configured handler is a method handler but only the class is specified. So, the example configuration could also be written just like a normal handler:

```xml
<Location /pleasure-craft>
  SetHandler perl-script
  PerlHandler Cookbook::Dinghy
</Location>
```

and as long as `Cookbook::Dinghy::handler()` was prototyped properly it would be called as a method handler. This is a convenient way of using object-oriented functionality without burdening end users with the details.

The only caveat to allowing mod_perl to imply the handler() method is that you must preload your module using either the PerlModule directive or from within your startup.pl if you use the arrow syntax from within your httpd.conf. This is true whether you specify the default handler() method or some other method of your choosing.

Another pitfall to be aware of with method handlers is when using the `push_handlers()` or `set_handlers()` methods. As discussed in Recipe 8.8, both methods accept either a subroutine reference or a string representation of the handler. For manipulating method handlers only the string format will work as expected. This is because it is difficult to tell whether a code reference is meant to be called as a simple subroutine, which gets $r as the first argument, or as a method, which gets $class and $r.

```perl
package Cookbook::Dinghy;

sub handler ($$) {

  my ($self,$r) = @_;

  $r->push_handlers(PerlLogHandler => 'Cookbook::Logger->handler');

  # Continue along...
}
```

1;
Again, you must ensure that the module containing the method you are adding to the handler stack is preloaded; otherwise you will get a runtime exception.

These pitfalls are relatively minor. There are many advantages to using method handlers over their customary but not object-oriented counterparts, as the next recipe illustrates.

### 10.4. Using Method Handlers

You want to take advantage of the features of method handlers.

**Technique**

Subclass the module and provide the desired method(s).

```perl
package Cookbook::Authenticate;

use Apache::Constants qw(REDIRECT);
use Apache::AuthCookie;

use Cookbook::Utils qw(authenticate_user authenticate_session);

use strict;

@Cookbook::Authenticate::ISA = qw(Apache::AuthCookie);

sub authen_cred {
    # Do what is needed to authenticate the supplied credentials
    # and return a session key, or undef on failure.
    my ($self, $r, $user, $password) = @_;  
    my $session = authenticate_user($user, $password);
    return $session;
}
```
sub authen_ses_key {
  # Do what is needed to authenticate the session key,
  # and return the user name if it checks, or undef.

  my ($self, $r, $session) = @_;

  my $user = authenticate_session($session);

  return $user;
}

sub logout ($$) {
  # Call Apache::AuthCookie::logout() to make sure that we get
  # rid of all the credentials, then redirect to a friendly page.

  my ($self, $r) = @_;

  $self->SUPER::logout($r);

  $r->headers_out->set(Location => '/logged-out.html');

  return REDIRECT;
}

Comments
Overriding or providing an additional method for a mod_perl method handler follows
the same basic rules for inheritance as any other object-oriented Perl module. As
illustrated in Recipe 10.2, the key concept here is to define the @ISA array so that it
contains the name of the parent modules from which you want to inherit. After this,
you can define additional methods to supplement the parent class, or override specific
methods with your own implementations. An example of where this technique really
proves its worth is with the Apache::AuthCookie module, available from CPAN, which
provides user authentication and authorization via cookies. This allows you to design a
site that uses an HTML form to gather the end-user username and password instead
of using the standard browser pop-up box. A more detailed discussion of Apache’s
authentication model, and some specifics around using a form-based authentication
mechanism is forthcoming in Chapter 13.

To implement authentication using Apache::AuthCookie you are only required to
implement two methods: authen_cred(), which checks a user-supplied credential and
returns a session key, and `authen_ses_key()`, which verifies the session key returned by
`authen_cred()`. After you define both `authen_cred()` and `authen_ses_key()`, and declare your module to be a subclass of `Apache::AuthCookie`, all that is required are a
few configuration additions. Here is a sample configuration.

```
# First, set up a few things required of all protected directories
# see the Apache::AuthCookie manpage for a more detailed explanation.

PerlModule Cookbook::Authenticate

PerlSetVar protectedPath /
PerlSetVar protectedScript /login.html

<Location /login>
  AuthType Cookbook::Authenticate
  AuthName protected
  SetHandler perl-script
  PerlHandler Cookbook::Authenticate->login
</Location>

<Location /logout>
  AuthType Cookbook::Authenticate
  AuthName protected
  SetHandler perl-script
  PerlHandler Cookbook::Authenticate->logout
</Location>

# Now, any directory that requires authentication
# just needs to follow this model.

<Directory /usr/local/apache/htdocs/protected>
  AuthType Cookbook::Authenticate
  AuthName protected
  PerlAuthenHandler Cookbook::Authenticate->authenticate
  require valid-user
</Directory>
```

Note from both the configuration and the solution code that neither `authen_cred()`
or `authen_ses_key()` are mod_perl handlers; all of the details of the actual cookie
setting and parsing, as well as the return statuses, are hidden. This allows you to spend
your time programming what is truly unique to your environment instead of
supporting an entire authentication framework.
Even though you are required to override authen_cred() and authen_ses_key() as part of the API, because Apache::AuthCookie is subclassable, you can also override default Apache::AuthCookie functionality. In our case, we chose to override the logout() method. This allows us to present a custom HTML page after clearing all the user credentials.

Although this may seem like a lot of work just to get authentication working for a single directory, compared to doing cookie-based authentication using normal CGI it is a snap. The majority of the directives are just initial Apache::AuthCookie overhead, which need be set up only once to serve any number of protected directories. And remember the actual amount of code required of the programmer—with just two user-supplied methods, you can readily use an existing module customized to fit your own requirements.

Now that you know method handlers exist and how to implement them, you can see that this example is really no different from the Apache::SSI example of Recipe 10.2—both Apache::AuthCookie and Apache::SSI rely on method handlers to make method inheritance possible. Where it is different is conceptually—here we are explicitly calling mod_perl handlers we have not defined, fully leveraging the power of object-oriented programming in a very nontransparent way. After you experience this kind of flexibility, you may never go back to ordinary handlers again.

### 10.5. Subclassing the Apache Class

You want to alter the behavior of one or more Apache class methods.

**Technique**

Create your own class that inherits from the Apache class and retrieve the Apache request object from your class.

```perl
package Cookbook::Apache;

use Apache;

use strict;
```
@Cookbook::Apache::ISA = qw(Apache);

sub new {
    my ($class, $r) = @_; 
    $r ||= Apache->request; 
    return bless { r => $r }, $class; 
}

sub bytes_sent {
    # This overrides the Apache bytes_sent() method, and
    # simply returns the value in (rounded) KB.
    return sprintf("%.0f", shift->SUPER::bytes_sent / 1024);
}

This class can now be used from within a handler.

class Cookbook::Logger;

use Apache::Constants qw(OK);
use Cookbook::Apache;
use strict;

sub handler {
    my $r = Cookbook::Apache->new(shift);
    my $kb = $r->bytes_sent;
    $r->warn("Sent $kb KB");
    return OK;
}

Comments
Subclassing the Apache class follows the same basic procedure described in the previous recipes. Just modify the $ISA array to add the Apache class to the inheritance tree for your class, and then override Apache methods by simply defining them within your package. As is true of all properly inheritable classes, the original Apache methods are still available via the SUPER pseudo-class, as shown in the example code.

After you declare your subclass, you have to create a constructor method so that you can access the Apache request object through your class instead of through the Apache class. To do this, create a new() method that returns a hash reference bless()ed into your subclass. The key to making this work is that the hash reference must contain an _r or _r key, which points to an Apache request object. After that, the hash may contain whatever keys and values you desire.

```perl
# Store some private data within the request object.
return bless { _r => Apache->request,
               _dbh => $dbh }, $class;
```

Keep in mind that this data will get cleared each time you retrieve a new request object unless you take the appropriate steps within your new() constructor to make the data persistent across the entire request. If the only reason you are creating a subclass is to store per-request data, then consider the pnotes() method described in Recipe 8.11 instead.

### 10.6. Subclassing the Apache Class Using XS

You want to alter the behavior of one or more Apache class methods using an XS routine.

**Technique**
Use h2xs to build the stub of the module, then follow these detailed instructions.

**Comments**
In some circumstances you may want to subclass Apache in order to use your own XS routine instead of a Perl routine. Here we describe how to alter the previous recipe using a bytes_sent() method written in C.
We begin by creating an XS-based module `Cookbook::Apache`, which can access the request object from an XS routine. This follows many of the same steps as in Recipe 3.22; the essential files needed are `Apache.xs`:

**Listing 10.1  Apache.xs**

```c
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "mod_perl.h"

MODULE = Cookbook::Apache         PACKAGE = Cookbook::Apache

PROTOTYPES: ENABLE

double
_bytes_sent(r)
Apache r

CODE:
    RETVAL = (double) r->bytes_sent / 1024;

OUTPUT:
    RETVAL
```

which defines a routine `_bytes_sent()`, which uses the `bytes_sent()` method of the C request object. The module file `Apache.pm`:

**Listing 10.2  Apache.pm**

```perl
package Cookbook::Apache;

use Apache;

use 5.006;
use DynaLoader;

use strict;

our @ISA = qw(DynaLoader Apache);
our $VERSION = '0.01';

__PACKAGE__->bootstrap($VERSION);

sub new {
```
my ($class, $r) = @_;  
$r ||= Apache::request;  
return bless { r => $r }, $class;

sub bytes_sent {  
 return sprintf("%.0f", shift->bytes_sent);  
}  

is similar to that of the previous recipe, but the bytes_sent() method overrides the standard Apache method with the XS _bytes_sent() routine. Note that no routines are exported from this module, which is a general feature of object-oriented modules.

The module is built and installed as in Recipe 3.22, using the appropriate Makefile.PL and typemap files. Use of this module from within a handler is exactly the same as the example of the pure Perl module of the previous recipe; all of the gory XS details are thus shielded from the end user.

10.7 Subclassing Apache::Registry

You are using Apache::Registry but need to enhance a particular feature for your immediate needs.

**Technique**

Subclass Apache::RegistryNG (not Apache::Registry), provided as part of the standard mod_perl distribution.

```perl
package Cookbook::Registry;

use Apache::Constants qw(OK NOT_FOUND);
use Apache::RegistryNG;

use strict;

@Cookbook::Registry::ISA = qw(Apache::RegistryNG);
```
sub sub_wrap {
    # Allow Registry scripts to use the handler syntax.
    my($pr, $code, $package) = @_;

    $code    ||= $pr->{'code'};
    $package ||= $pr->{'namespace'};

    # Replace the package identifier with the one
    # generated by Apache::Registry.
    (my $sub = $$code) =~ s/^(package ).*/$1$package;/;

    $pr->{'sub'} = \$sub;
}

sub can_compile {
    # Only check for readable files with content.
    my $pr = shift;
    my $r = $pr->{r};

    if (-r $r->finfo && -s _) {
        $pr->{'mtime'} = -M _;
        return OK;
    }

    $r->log_error($r->filename, " not found or unable to stat");
    return NOT_FOUND;
}

sub run {
    my ($pr, @args) = @_; 

    # Capture both the Apache::RegistryNG return code
    # and the return code of our handler.
    my ($ng_rc, $rc) = shift->SUPER::run(@args);

    # If RegistryNG executed OK, return the return code from our handler.
    return $ng_rc == OK ? $rc : $ng_rc;
}
1;
Comments
Although Apache::Registry is an amazing accomplishment, at some point it may fall just a little bit short of the functionality you are seeking and you will look into modifying the source code. At that point, you will probably take a step back and wonder whether there is not a better way—Apache::Registry is a complex animal that is not easily tamed. We are happy to report that there is indeed a better way.

Apache::RegistryNG is a subclassable, object-oriented handler with all the functionality of Apache::Registry divided into separate class methods. It was designed as a next generation (thus the NG), cleaner replacement for Apache::Registry that would lend itself to easy extensibility. Although it actually is a subclass of Apache::PerlRun, the handler() methods are different in that, like Apache::Registry and unlike Apache::PerlRun, the script is cached in memory. However, script namespace protection is governed by the physical filename and not by the URI, which is subtly different from Apache::Registry. Of course, if you do not like that particular aspect of Apache::RegistryNG, you can always subclass it.

Implementing a subclass of Apache::RegistryNG is the same as subclassing any other Perl module—just set @ISA and continue about your business. Unfortunately, you will have to dig through the Apache::PerlRun code to find out the names and functions for all the methods, because documentation for both of these classes is rather sparse. The good news is that the code is relatively straightforward and easy to trace due to the modular design. One important thing to note is that Apache::RegistryNG is object-oriented and expects to be called as a method handler, as in the example configuration later in this recipe.

The previous example replaces the sub_wrap(), can_compile(), and run() methods. All of these methods are defined in Apache::PerlRun and inherited by Apache::RegistryNG. can_compile() ordinarily has the job of checking for the proper Apache configuration options and file permissions. sub_wrap() is a bit more cryptic. It has the job of turning a script into a module by wrapping the code in a handler() subroutine and placing the code within in a protected package namespace. After the code has been compiled, run() actually executes the routine.

In our example, we do things only slightly differently than the default methods. Normally Apache::Registry disguises the called script so that it looks like a handler by adding a package declaration and handler() subroutine. Instead we assume that the script really is a complete handler. This means that we can eliminate most of the checks in the can_compile() method. We only need to test if the file is readable and
has content. The sub_wrap() method remains the same with one exception—we replace the handler's package identifier with the namespace that Apache::RegistryNG generates and expects to see at runtime. run() also needs a little modification. The standard version returns the return code for Apache::RegistryNG back to Apache, not return code of the handler() subroutine we are running. Fortunately, run() does return both values when called in a list context; we add some logic to capture both and return the appropriate value.

Although this is an interesting illustration of how to extend the functionality of Apache::RegistryNG, the implications in this particular case are worth noting. The net effect is that the new handler creates an environment where handlers are run like scripts, which is the exact reverse of what Apache::Registry ends up doing. A configuration similar to

```
PerlModule Cookbook::Registry

Alias /handler-bin /usr/local/apache/lib/perl
<Location /handler-bin>
  SetHandler perl-script
  PerlHandler Cookbook::Registry->handler
</Location>
```

would be able to serve an ordinary mod_perl content handler, such as the Cookbook::SendAnyDoc example from Recipe 6.7, via a URL like

```
http://localhost/handler-bin/Cookbook/SendAnyDoc.pm/docs/file.pdf
```

The effect is unique in that it allows you to bind a single <Location> or <Directory> container to more than one handler. This has the possibility of being a convenient environment for handler development—your handler is still governed by the same rules as normal handlers regarding code that resides outside of the handler() subroutine, whereas functionality like that provided by Apache::Reload is built-in.

It should be recognized that although this particular example may be an interesting approach for module development, it is not really something that should make its way into a production system due to the total lack of security. If the ideas of using a single <Location> for all your handlers sounds appealing, consider using the more secure and better tested Apache::Dispatch instead, as described in Recipe 15.2. This example is hereby labeled “for educational purposes only.”
10.8. Subclassing Apache::Request

You want to add functionality to the Apache::Request class.

**Technique**
Create a subclass of Apache::Request similar to the following example by adding or overriding some methods. Then use the new subclass in your script or handler instead of Apache::Request.

```perl
package Cookbook::TransformRequest;

use Apache::Request;

use strict;

@Cookbook::TransformRequest::ISA = qw(Apache::Request);

sub new {
    my ($class, $r, $input_transform, $output_transform) = @_
    return bless {
        r => Apache::Request->new($r),
        input_transform => $input_transform,
        output_transform => $output_transform,
    }, $class;
}

sub param {
    my ($self, $field) = @_
    my $transform = $self->{input_transform};
    return $self->SUPER::param($field) unless ($transform);
    return map { defined($_) ? &$transform($_) : undef}
        $self->SUPER::param($field);  
}
```
sub print {
    my ($self, @args) = @_;

    @args = @{$self->{output_transform}}(@args)
    if $self->{output_transform};

    $self->SUPER::print(@args);
}

Comments
The Apache::Request class, described in great detail in Chapter 3, gives us most of the functionality needed to read form fields and send output. At some point though, you will find yourself wanting to extend or enhance Apache::Request. Instead of cluttering up your code with repeated constructs, consider subclassing the Apache::Request class. It's quite simple and the concepts are very similar to subclassing the Apache class described in Recipe 10.5.

Our example class, Cookbook::TransformRequest, customizes Apache::Request by allowing you to transform input from the param() method or output via the print() method. This can be used for many purposes, including character set translation and post-processing data. It is usable as is or can be subclassed further to make a whole set of objects. Refer to it and the following class diagram as we describe the process for creating Apache::Request subclasses.

The first step in creating an Apache::Request subclass is to define a new Perl package and class. Designate the parent class by setting the @ISA array to Apache::Request. Your subclass must implement a new() method that returns a hash reference bless()ed into your subclass. This hash must contain the key r that contains a fully functional Apache::Request object.

Next, add functionality to the class. Create new methods to implement added functionality, or override methods to change the default Apache::Request behavior. For our example we chose the latter; we override the print() and param() methods so we can transform all input and output. Note that param() is a true Apache::Request method but print() is actually inherited from the Apache class—we are two levels deep now.
We already added some functionality to the `new()` method derived from `Apache::Request`. Two new arguments take function references that implement the input or output transform routines. If defined, the transforms are applied to input or output transparently—our scripts and handlers need not know the difference. The reimplemented `param()` method applies the transform to form fields, whereas the reimplemented `print()` method transforms all output. Note the extensive use of the `SUPER` construct to call methods in the parent class.

Finally we can use this new class in a script or handler. The following simpleminded code implements automatically uppercased output:

```perl
package Cookbook::UPPERCASE;

use Apache::Constants qw(OK);

use Cookbook::TransformRequest;

use strict;
```
sub handler {
    my $r = Cookbook::TransformRequest->new(shift, undef, sub {uc join('', @_)});

    $r->send_http_header('text/plain);
    $r->print("all output is in Upper case");

    return OK;
}
1;

This is great—we can now easily transform the output in any odd way by merely altering each and every script or handler to define the output transformation. There is a better way however: Subclass the subclass. You will find this is necessary when things get more complicated. For example, consider a more complex class that translates all output into "pirate speak" whenever the query string contains pirate=1. We can define a new class Cookbook::PirateRequest that encapsulates the behavior:

package Cookbook::PirateRequest;

use Cookbook::TransformRequest;

@Cookbook::PirateRequest::ISA = qw(Cookbook::TransformRequest);

use strict;

sub as_pirate {
    my $arg = join('', @_);

    $arg =~ s/ boy/ matey/g;
    $arg =~ s/ yes/ aye/g;
    $arg =~ s/ my/ me/g;
    $arg =~ s/ treasure/ booty/g;

    return 'Argh! ' . $arg;
}

sub new {
    my($class, $r) = @_;
Now we just replace all instances of *Apache::Request* with *Cookbook::PirateRequest*. After this is done our code will automatically switch between pirate and normal output. For example, the code

```perl
package Cookbook::UsePirate;
use Apache::Constants qw(OK);
use Cookbook::PirateRequest;
use strict;

sub handler {
    my $r = Cookbook::PirateRequest->new(shift);
    $r->send_http_header('text/plain');
    $r->print("Fetch my treasure boy");
    return OK;
}
```

will output *Argh! Fetch me booty matey* if `pirate=1` is specified in the URL; otherwise, it just outputs *Fetch my treasure boy*.

To see the full power of this approach consider the problem of dealing with character sets. Often the character set used by the Web browser does not match that used internally by the application (often the Unicode’s UTF-8 encoding). We could add conversion routines all over the place, but this task is tedious and error prone. Instead, we can create a module that does all of this transformation logic, like this:
package Cookbook::UTF8Request;

use Cookbook::TransformRequest ();
use Text::Iconv ();

use strict;

sub new {
    my($class, $r, $charset) = @_; 

    my $to_unicode = Text::Iconv->new($charset, 'UTF-8');
    my $from_unicode = Text::Iconv->new('UTF-8', $charset);

    my $input_transform = sub {$to_unicode->convert(@_)};
    my $output_transform = sub {$from_unicode->convert(@_)};

    return Cookbook::TransformRequest->new($r, $input_transform, $output_transform);
}

1;

We use the Text::Iconv module (available on CPAN) to convert between character sets. To use this module we only need to add a character set argument (which you will probably find via the user's setting or a cookie) and pass that into Cookbook::UTF8Request's new() method. All of the transformations into and out of UTF-8 are handled transparently.

The following example illustrates the usage of the Cookbook::UTF8Request module by defining a simple handler named Cookbook::Japanese. This handler displays a form with some Japanese characters. To show the UTF-8 functionality we use Perl escape sequences starting with \x instead of raw UTF-8 text.

package Cookbook::Japanese;

use Cookbook::UTF8Request;

use strict;

sub handler {

my $r = Cookbook::UTF8Request->new(shift, 'iso-2022-jp');
$r->send_http_header('text/html; charset=iso-2022-jp');
my $yes = '\x{3059}\x{308B}';
my $no = '\x{3057}\x{306A}\x{3044}';
my ($name) = $r->param('name');
$r->print(<<HERE);
<html>
<body>
  Name is '$name'<br>
  <form method="POST">
    <input type="text" name="name"><input type="submit">
  </form>
  Yes ($yes), No ($no)
</body>
</html>
HERE
}
1;

This handler, like many others, uses simple print() and param() methods to display a
form and process form input. The only difference is that we use the
Cookbook::UTF8Request module, and we explicitly set the character set to iso-2022-jp
in our Content-Type header.