NAME

HTML::Tree::AboutObjects −− article: "User's View of Object−Oriented Modules"

SYNOPSIS

```
# This an article, not a module.
```

DESCRIPTION

A User's View of Object-Oriented Modules

−− Sean M. Burke

The first time that most Perl programmers run into object-oriented programming when they need to use a module whose interface is object-oriented. This is often a mystifying experience, since talk of ''methods'' and ''constructors'' is unintelligible to programmers who thought that functions and variables was all there was to worry about.

Articles and books that explain object-oriented programming (OOP), do so in terms of how to program that way. That's understandable, and if you learn to write object-oriented code of your own, you'd find it easy to use object-oriented code that others write. But this approach is the *long* way around for people whose immediate goal is just to use existing object-oriented modules, but who don't yet want to know all the gory details of having to write such modules for themselves.

This article is for those programmers — programmers who want to know about objects from the perspective of using object-oriented modules.

Modules and Their Functional Interfaces

Modules are the main way that Perl provides for bundling up code for later use by yourself or others. As I'm sure you can't help noticing from reading *The Perl Journal*, CPAN (the Comprehensive Perl Archive Network) is the repository for modules (or groups of modules) that others have written, to do anything from composing music to accessing Web pages. A good deal of those modules even come with every installation of Perl.

One module that you may have used before, and which is fairly typical in its interface, is Text::Wrap. It comes with Perl, so you don't even need to install it from CPAN. You use it in a program of yours, by having your program code say early on:

```
use Text::Wrap;
```

and after that, you can access a function called `wrap`, which inserts line-breaks in text that you feed it, so that the text will be wrapped to seventy-two (or however many) columns.

The way this `use Text::Wrap` business works is that the module Text::Wrap exists as a file ''Text/Wrap.pm'' somewhere in one of your library directories. That file contains Perl code...

> Footnote: And mixed in with the Perl code, there's documentation, which is what you read with ''perldoc Text::Wrap''. The perldoc program simply ignores the code and formats the documentation text, whereas ''use Text::Wrap'' loads and runs the code while ignoring the documentation.

...which, among other things, defines a function called `Text::Wrap::wrap`, and then `exports` that function, which means that when you say `wrap` after having said ''use Text::Wrap'', you'll be actually calling the `Text::Wrap::wrap` function. Some modules don't export their functions, so you have to call them by their full name, like `Text::Wrap::wrap(...parameters...)`.

Regardless of whether the typical module exports the functions it provides, a module is basically just a container for chunks of code that do useful things. The way the module allows for you to interact with it, is its *interface*. And when, like with Text::Wrap, its interface consists of functions, the module is said to have a **functional interface**.

> Footnote: the term ''function'' (and therefore "function*al*") has various senses. I'm using the term here in its broadest sense, to refer to routines — bits of code that are called by some name and which

take parameters and return some value.

Using modules with functional interfaces is straightforward — instead of defining your own ‘‘wrap’’ function with `sub wrap { ... }`, you entrust ‘‘use Text::Wrap’’ to do that for you, along with whatever other functions its defines and exports, according to the module’s documentation. Without too much bother, you can even write your own modules to contain your frequently used functions; I suggest having a look at the `perlmod` man page for more leads on doing this.

Modules with Object-Oriented Interfaces

So suppose that one day you want to write a program that will automate the process of `ftp`ing a bunch of files from one server down to your local machine, and then off to another server.

A quick browse through search.cpan.org turns up the module ‘‘Net::FTP’’, which you can download and install it using normal installation instructions (unless your sysadmin has already installed it, as many have).

Like Text::Wrap or any other module with a familiarly functional interface, you start off using Net::FTP in your program by saying:

```
use Net::FTP;
```

However, that’s where the similarity ends. The first hint of difference is that the documentation for Net::FTP refers to it as a **class**. A class is a kind of module, but one that has an object-oriented interface.

Whereas modules like Text::Wrap provide bits of useful code as *functions*, to be called like `function(...parameters...)` or like `PackageName::function(...parameters...)`, Net::FTP and other modules with object-oriented interfaces provide **methods**. Methods are sort of like functions in that they have a name and parameters; but methods look different, and are different, because you have to call them with a syntax that has a class name or an object as a special argument. I’ll explain the syntax for method calls, and then later explain what they all mean.

Some methods are meant to be called as **class methods**, with the class name (same as the module name) as a special argument. Class methods look like this:

```
ClassName->methodname(parameter1, parameter2, ...)
ClassName->methodname()   # if no parameters
ClassName->methodname     # same as above
```

which you will sometimes see written:

```
methodname ClassName (parameter1, parameter2, ...)
methodname ClassName      # if no parameters
```

Basically all class methods are for making new objects, and methods that make objects are called "**constructors**‘‘ (and the process of making them is called ’’constructing‘‘ or ’’instantiating‘‘). Constructor methods typically have the name ’’new‘‘, or something including ’’new‘‘ (’’new_from_file‘‘, etc.); but they can conceivably be named anything — DBI’s constructor method is named ’’connect", for example.

The object that a constructor method returns is typically captured in a scalar variable:

```
$object = ClassName->new(param1, param2...);
```

Once you have an object (more later on exactly what that is), you can use the other kind of method call syntax, the syntax for **object method** calls. Calling object methods is just like class methods, except that instead of the ClassName as the special argument, you use an expression that yields an ‘‘object’’. Usually this is just a scalar variable that you earlier captured the output of the constructor in. Object method calls look like this:

```
$object->methodname(parameter1, parameter2, ...);
$object->methodname()   # if no parameters
$object->methodname     # same as above
```

which is occasionally written as:

```
methodname $object (parameter1, parameter2, ...)
methodname $object      # if no parameters
```

Examples of method calls are:

```
my $session1 = Net::FTP->new("ftp.myhost.com");
  # Calls a class method "new", from class Net::FTP,
  #  with the single parameter "ftp.myhost.com",
  #  and saves the return value (which is, as usual,
  #  an object), in $session1.
  # Could also be written:
  #  new Net::FTP('ftp.myhost.com')
$session1->login("sburke","aoeuaoeu")
  || die "failed to login!\n";
   # calling the object method "login"
print "Dir:\n", $session1->dir(), "\n";
$session1->quit;
  # same as $session1->quit()
print "Done\n";
exit;
```

Incidentally, I suggest always using the syntaxes with parentheses and "->" in them,

Footnote: the character-pair "->" is supposed to look like an arrow, not "negative greater-than"!

and avoiding the syntaxes that start out "methodname `$object`" or "methodname ModuleName". When everything's going right, they all mean the same thing as the "->" variants, but the syntax with "->" is more visually distinct from function calls, as well as being immune to some kinds of rare but puzzling ambiguities that can arise when you're trying to call methods that have the same name as subroutines you've defined.

But, syntactic alternatives aside, all this talk of constructing objects and object methods begs the question — what*is* an object? There are se veral angles to this question that the rest of this article will answer in turn: what can you do with objects? what's in an object? what's an object value? and why do some modules use objects at all?

What Can You Do with Objects?

You've seen that you can make objects, and call object methods with them. But what are object methods for? The answer depends on the class:

A Net::FTP object represents a session between your computer and an FTP server. So the methods you call on a Net::FTP object are for doing whatever you'd need to do across an FTP connection. You make the session and log in:

```
my $session = Net::FTP->new('ftp.aol.com');
die "Couldn't connect!" unless defined $session;
  # The class method call to "new" will return
  # the new object if it goes OK, otherwise it
  # will return undef.

$session->login('sburke', 'p@ssw3rD')
 || die "Did I change my password again?";
  # The object method "login" will give a true
  # return value if actually logs in, otherwise
  # it'll return false.
```

You can use the session object to change directory on that session:

```
$session->cwd("/home/sburke/public_html")
   || die "Hey, that was REALLY supposed to work!";
 # if the cwd fails, it'll return false
```

...get files from the machine at the other end of the session...

```
foreach my $f ('log_report_ua.txt', 'log_report_dom.txt',
               'log_report_browsers.txt')
{
  $session->get($f) || warn "Getting $f failed!"
};
```

...and plenty else, ending finally with closing the connection:

```
$session->quit();
```

In short, object methods are for doing things related to (or with) whatever the object represents. For FTP sessions, it's about sending commands to the server at the other end of the connection, and that's about it — there, methods are for doing something to the world outside the object, and the objects is just something that specifies what bit of the world (well, what FTP session) to act upon.

With most other classes, however, the object itself stores some kind of information, and it typically makes no sense to do things with such an object without considering the data that's in the object.

What's *in* an Object?

An object is (with rare exceptions) a data structure containing a bunch of attributes, each of which has a value, as well as a name that you use when you read or set the attribute's value. Some of the object's attributes are private, meaning you'll never see them documented because they're not for you to read or write; but most of the object's documented attributes are at least readable, and usually writeable, by you. Net::FTP objects are a bit thin on attributes, so we'll use objects from the class Business::US_Amort for this example. Business::US_Amort is a very simple class (available from CPAN) that I wrote for making calculations to do with loans (specifically, amortization, using US-style algorithms).

An object of the class Business::US_Amort represents a loan with particular parameters, i.e., attributes. The most basic attributes of a ''loan object'' are its interest rate, its principal (how much money it's for), and it's term (how long it'll take to repay). You need to set these attributes before anything else can be done with the object. The way to get at those attributes for loan objects is just like the way to get at attributes for any class's objects: through accessors. An **accessor** is simply any method that accesses (whether reading or writing, AKA getting or putting) some attribute in the given object. Moreover, accessors are the **only** way that you can change an object's attributes. (If a module's documentation wants you to know about any other way, it'll tell you.)

Usually, for simplicity's sake, an accessor is named after the attribute it reads or writes. With Business::US_Amort objects, the accessors you need to use first are `principal`, `interest_rate`, and `term`. Then, with at least those attributes set, you can call the `run` method to figure out several things about the loan. Then you can call various accessors, like `total_paid_toward_interest`, to read the results:

```
use Business::US_Amort;
my $loan = Business::US_Amort->new;
# Set the necessary attributes:
$loan->principal(123654);
$loan->interest_rate(9.25);
$loan->term(20); # twenty years

# NOW we know enough to calculate:
$loan->run;

# And see what came of that:
print
```

```
"Total paid toward interest: A WHOPPING ",
$loan->total_paid_interest, "!!\n";
```

This illustrates a convention that's common with accessors: calling the accessor with no arguments (as with `$loan`->total_paid_interest) usually means to read the value of that attribute, but providing a value (as with `$loan`->term(20)) means you want that attribute to be set to that value. This stands to reason: why would you be providing a value, if not to set the attribute to that value?

Although a loan's term, principal, and interest rates are all single numeric values, an objects values can any kind of scalar, or an array, or even a hash. Moreover, an attribute's value(s) can be objects themselves. For example, consider MIDI files (as I wrote about in TPJ#13): a MIDI file usually consists of several tracks. A MIDI file is complex enough to merit being an object with attributes like its overall tempo, the file-format variant it's in, and the list of instrument tracks in the file. But tracks themselves are complex enough to be objects too, with attributes like their track-type, a list of MIDI commands if they're a MIDI track, or raw data if they're not. So I ended up writing the MIDI modules so that the ''tracks'' attribute of a MIDI::Opus object is an array of objects from the class MIDI::Track. This may seem like a runaround — you ask what's in one object, and get *another* object, or several! But in this case, it exactly reflects what the module is for — MIDI files contain MIDI tracks, which then contain data.

What is an Object Value?

When you call a constructor like Net::FTP->new(*hostname*), you get back an object value, a value you can later use, in combination with a method name, to call object methods.

Now, so far we've been pretending, in the above examples, that the variables `$session` or `$loan` *are* the objects you're dealing with. This idea is innocuous up to a point, but it's really a misconception that will, at best, limit you in what you know how to do. The reality is not that the variables `$session` or `$query` are objects; it's a little more indirect — they *hold* values that symbolize objects. The kind of value that `$session` or `$query` hold is what I'm calling an object value.

To understand what kind of value this is, first think about the other kinds of scalar values you know about: The first two scalar values you probably ever ran into in Perl are **numbers** and **strings**, which you learned (or just assumed) will usually turn into each other on demand; that is, the three-character string ''2.5'' can become the quantity two and a half, and vice versa. Then, especially if you started using`perl -w` early on, you learned about the **undefined value**, which can turn into 0 if you treat it as a number, or the empty-string if you treat it as a string.

> Footnote: You may *also* have been learning about references, in which case you're ready to hear that object values are just a kind of reference, except that they reflect the class that created thing they point to, instead of merely being a plain old array reference, hash reference, etc. *If* this mak es makes sense to you, and you want to know more about how objects are implemented in Perl, have a look at the `perltoot` man page.

And now you're learning about **object values**. An object value is a value that points to a data structure somewhere in memory, which is where all the attributes for this object are stored. That data structure as a whole belongs to a class (probably the one you named in the constructor method, like ClassName->new), so that the object value can be used as part of object method calls.

If you want to actually *see* what an object value is, you might try just saying ''print `$object`''. That'll get you something like this:

```
Net::FTP=GLOB(0x20154240)
```

or

```
Business::US_Amort=HASH(0x15424020)
```

That's not very helpful if you wanted to really get at the object's insides, but that's because the object value is only a symbol for the object. This may all sound very abstruse and metaphysical, so a real-world allegory might be very helpful:

> You get an advertisement in the mail saying that you have been (im)personally selected to have the rare privilege of applying for a credit card. For whatever reason, *this* offer sounds good to you, so you

fill out the form and mail it back to the credit card company. They gleefully approve the application and create your account, and send you a card with a number on it.

Now, you can do things with the number on that card — clerks at stores can ring up things you want to buy, and charge your account by keying in the number on the card. You can pay for things you order online by punching in the card number as part of your online order. You can pay off part of the account by sending the credit card people some of your money (well, a check) with some note (usually the pre-printed slip) that has the card number for the account you want to pay toward. And you should be able to call the credit card company's computer and ask it things about the card, like its balance, its credit limit, its APR, and maybe an itemization of recent purchases ad payments.

Now, what you're *really* doing is manipulating a credit card *account*, a completely abstract entity with some data attached to it (balance, APR, etc). But for ease of access, you have a credit card *number* that is a symbol for that account. Now, that symbol is just a bunch of digits, and the number is effectively meaningless and useless in and of itself — but in the appropriate context, it's understood to *mean* the credit card account you're accessing.

This is exactly the relationship between objects and object values, and from this analogy, several facts about object values are a bit more explicable:

* An object value does nothing in and of itself, but it's useful when you use it in the context of an `$object->`method call, the same way that a card number is useful in the context of some operation dealing with a card account.

Moreover, several copies of the same object value all refer to the same object, the same way that making several copies of your card number won't change the fact that they all still refer to the same single account (this is true whether you're ''copying'' the number by just writing it down on different slips of paper, or whether you go to the trouble of forging exact replicas of your own plastic credit card). That's why this:

```
$x = Net::FTP->new("ftp.aol.com");
$x->login("sburke", "aoeuaoeu");
```

does the same thing as this:

```
$x = Net::FTP->new("ftp.aol.com");
$y = $x;
$z = $y;
$z->login("sburke", "aoeuaoeu");
```

That is, `$z` and `$y` and `$x` are three different *slots* for values, but what's in those slots are all object values pointing to the same object — you don't have three different FTP connections, just three variables with values pointing to the some single FTP connection.

* You can't tell much of anything about the object just by looking at the object value, any more than you can see your credit account balance by holding the plastic card up to the light, or by adding up the digits in your credit card number.

* You can't just make up your own object values and have them work — they can come only from constructor methods of the appropriate class. Similarly, you get a credit card number *only* by having a bank approve your application for a credit card account — at which point *they* let *you* know what the number of your new card is.

Now, there's even more to the fact that you can't just make up your own object value: even though you can print an object value and get a string like ''Net::FTP=GLOB(0x20154240)'', that string is just a *representation* of an object value.

Internally, an object value has a basically different type from a string, or a number, or the undefined value — if`$x` holds a real string, then that v alue's slot in memory says "this is a value of type *string*, and its characters are...'', whereas if it's an object value, the value's slot in memory says, ''this is a value of type *reference*, and the location in memory that it points to is..." (and by looking at what's at that location, Perl can tell the class of what's there).

Perl programmers typically don't have to think about all these details of Perl's internals. Many other

languages force you to be more conscious of the differences between all of these (and also between types of numbers, which are stored differently depending on their size and whether they have fractional parts). But Perl does its best to hide the different types of scalars from you — it turns numbers into strings and back as needed, and takes the string or number representation of undef or of object values as needed. However, you can't go from a string representation of an object value, back to an object value. And that's why this doesn't work:

```
$x = Net::FTP->new('ftp.aol.com');
$y = Net::FTP->new('ftp.netcom.com');
$z = Net::FTP->new('ftp.qualcomm.com');
$all = join(' ', $x,$y,$z);           # !!!
...later...
($aol, $netcom, $qualcomm) = split(' ', $all);  # !!!
$aol->login("sburke", "aoeuaoeu");
$netcom->login("sburke", "qjkxqjkx");
$qualcomm->login("smb", "dhtndhtn");
```

This fails because `$aol` ends up holding merely the **string representation** of the object value from `$x`, not the object value itself — when `join` tried to join the characters of the ''strings'' `$x`, `$y`, and `$z`, Perl saw that they weren't strings at all, so it gave `join` their string representations.

Unfortunately, this distinction between object values and their string representations doesn't really fit into the analogy of credit card numbers, because credit card numbers really *are* numbers — even thought they don't express any meaningful quantity, if you stored them in a database as a quantity (as opposed to just an ASCII string), that wouldn't stop them from being valid as credit card numbers.

This may seem rather academic, but there's there's two common mistakes programmers new to objects often make, which make sense only in terms of the distinction between object values and their string representations:

The first common error involves forgetting (or never having known in the first place) that when you go to use a value as a hash key, Perl uses the string representation of that value. When you want to use the numeric value two and a half as a key, Perl turns it into the three-character string ''2.5''. But if you then want to use that string as a number, Perl will treat it as meaning two and a half, so you're usually none the wiser that Perl converted the number to a string and back. But recall that Perl can't turn strings back into objects — so if you tried to use a Net::FTP object value as a hash key, Perl actually used its string representation, like ''Net::FTP=GLOB(0x20154240)'', but that string is unusable as an object value. (Incidentally, there's a module Tie::RefHash that implements hashes that *do* let you use real object-values as keys.)

The second common error with object values is in trying to save an object value to disk (whether printing it to a file, or storing it in a conventional database file). All you'll get is the string, which will be useless.

When you want to save an object and restore it later, you may find that the object's class already provides a method specifically for this. For example, MIDI::Opus provides methods for writing an object to disk as a standard MIDI file. The file can later be read back into memory by a MIDI::Opus constructor method, which will return a new MIDI::Opus object representing whatever file you tell it to read into memory. Similar methods are available with, for example, classes that manipulate graphic images and can save them to files, which can be read back later.

But some classes, like Business::US_Amort, provide no such methods for storing an object in a file. When this is the case, you can try using any of the Data::Dumper, Storable, or FreezeThaw modules. Using these will be unproblematic for objects of most classes, but it may run into limitations with others. For example, a Business::US_Amort object can be turned into a string with Data::Dumper, and that string written to a file. When it's restored later, its attributes will be accessible as normal. But in the unlikely case that the loan object was saved in mid-calculation, the calculation may not be resumable. This is because of the way that that *particular* class does its calculations, but similar limitations may occur with objects from other classes.

But often, even *wanting* to save an object is basically wrong — what would saving an ftp *session* even

mean? Saving the hostname, username, and password? current directory on both machines? the local TCP/IP port number? In the case of ''saving'' a Net::FTP object, you're better off just saving whatever details you actually need for your own purposes, so that you can make a new object later and just set those values for it.

So Why Do Some Modules Use Objects?

All these details of using objects are definitely enough to make you wonder — is it worth the bother? If you're a module author, writing your module with an object-oriented interface restricts the audience of potential users to those who understand the basic concepts of objects and object values, as well as Perl's syntax for calling methods. Why complicate things by having an object-oriented interface?

A somewhat esoteric answer is that a module has an object-oriented interface because the module's insides are written in an object-oriented style. This article is about the basics of object-oriented *interfaces*, and it'd be going far afield to explain what object-oriented *design* is. But the short story is that object-oriented design is just one way of attacking messy problems. It's a way that many programmers find very helpful (and which others happen to find to be far more of a hassle than it's worth, incidentally), and it just happens to show up for you, the module user, as merely the style of interface.

But a simpler answer is that a functional interface is sometimes a hindrance, because it limits the number of things you can do at once — limiting it, in fact, to one. For many problems that some modules are meant to solve, doing without an object-oriented interface would be like wishing that Perl didn't use filehandles. The ideas are rather simpler — just imagine that Perl let you access files, but *only* one at a time, with code like:

```
open("foo.txt") || die "Can't open foo.txt: $!";
while(readline) {
  print $_ if /bar/;
}
close;
```

That hypothetical kind of Perl would be simpler, by doing without filehandles. But you'd be out of luck if you wanted to read from one file while reading from another, or read from two and print to a third.

In the same way, a functional FTP module would be fine for just uploading files to one server at a time, but it wouldn't allow you to easily write programs that make need to use *several* simultaneous sessions (like ''look at server A and server B, and if A has a file called X.dat, then download it locally and then upload it to server B — except if B has a file called Y.dat, in which case do it the other way around'').

Some kinds of problems that modules solve just lend themselves to an object-oriented interface. For those kinds of tasks, a functional interface would be more familiar, but less powerful. Learning to use object-oriented modules' interfaces does require becoming comfortable with the concepts from this article. But in the end it will allow you to use a broader range of modules and, with them, to write programs that can do more.

[end body of article]

[Author Credit]

Sean M. Burke has contributed several modules to CPAN, about half of them object-oriented.

[The next section should be in a greybox:]

The Gory Details

For sake of clarity of explanation, I had to oversimplify some of the facts about objects. Here's a few of the gorier details:

* Every example I gave of a constructor was a class method. But object methods can be constructors, too, if the class was written to work that way: `$new` = `$old`->copy, `$node_y` = `$node_x`->new_subnode, or the like.

* I've given the impression that there's two kinds of methods: object methods and class methods. In fact, the same method can be both, because it's not the kind of method it is, but the kind of calls it's written to accept — calls that pass an object, or calls that pass a class-name.

* The term ‘‘object value’’ isn’t something you’ll find used much anywhere else. It’s just my shorthand for what would properly be called an ‘‘object reference’’ or ‘‘reference to a blessed item’’. In fact, people usually say ‘‘object’’ when they properly mean a reference to that object.

* I mentioned creating objects with *con*structors, but I didn’t mention destroying them with *de*structor — a destructor is a kind of method that you call to tidy up the object once you’re done with it, and want it to neatly go away (close connections, delete temporary files, free up memory, etc). But because of the way Perl handles memory, most modules won’t require the user to know about destructors.

* I said that class method syntax has to have the class name, as in `$session` = **Net::FTP**−>new($host). Actually, you can instead use any expression that returns a class name: `$ftp_class` = ’Net::FTP’; `$session` = **`$ftp_class`**−>new($host). Moreover, instead of the method name for object− or class-method calls, you can use a scalar holding the method name: `$foo`−>**`$method`**($host). But, in practice, these syntaxes are rarely useful.

And finally, to learn about objects from the perspective of writing your own classes, see the `perltoot` documentation, or Damian Conway’s exhaustive and clear book *Object Oriented Perl* (Manning Publications 1999, ISBN 1−884777−79−1).

BACK

Return to the HTML::Tree docs.