DAVID N. BLANK-EDELMAN

practical Perl tools: you never forget your first date (object)

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

AS THE MIGHTY BOOSH WOULD SAY, come with me now on a journey through time and space. Actually, that's a bit of an exaggeration. We're only going to go on a journey through time. Okay, too ambitious. New idea: we're going to look at Perl modules that deal with time. More precisely we're going to embark on a small survey of the most popular packages people use to handle date-like things in Perl today.

Before we dive in, I should mention that my predecessor in this column's pilot seat, Adam Turoff, wrote a date-related article back in 2005 in which he discussed a couple of the choices I'll mention below. We'll take a slightly different tack in this column (plus a decent amount has changed in the last five years), but I wanted to dispel any worry that the déjà vu feeling you might have is actually a glitch in the Matrix when they change something. Or maybe it is, I dunno.

Not the Built-In Stuff

For most of this column, I'm going to intentionally ignore both the built-in time functions that are part of the language and the time-related modules that ship with the core. Let's do a very quick review so we can get on to the more sophisticated stuff.

By far, the most used built-in function is time(), which will return a number like this:

```
1259340967
```

This is the "number of non-leap seconds since whatever time the system considers to be the epoch. . . . On most systems the epoch is 00:00:00 UTC, January 1, 1970." It is pretty common to use this function to grab the current time and then later on call it again to determine how much time has elapsed.

The very next thing most people do with the return value from time() is feed it to the localtime() (and less commonly) gmtime() functions. Both take that number of seconds and translate them into a list of the time buckets humans are likely to use, namely:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday, $isdst) =
                                        localtime (time);
```

The difference between localtime() and gmtime() is that localtime() attempts to return values based on the current time zone, and gmtime() returns it relative to GMT, also known as UTC.

The return values above are mostly self-explanatory except for the last few. $wday is the day of the week (starting at 0 for Sunday), $yday is the day of the year (also starting at 0), and $isdt is true if Daylight Saving Time is in effect. If remembering which item is in which position in a list like that is too annoying for you, Perl ships with Time::localtime and Time::gmtime modules that let you work with those values as an object instead of a list. This lets you write code that looks like this, to quote an example from the documentation:

```
use Time::localtime;
printf "Year is %d\n", localtime->year() + 1900;
```

The idea of representing dates as objects is something that will figure quite prominently in the rest of this column.

I should mention that this is not how I use localtime() most of the time. I mostly call localtime() in a scalar context, explicitly like this:

```
print scalar localtime; # output: Sat Nov 28 22:31:27 2009
```

In a scalar context, it returns a ctime()-like string (what you might expect to see if you typed "date"). I use this all the time for putting timestamps into logs and files. If you want to generate a string with a different format, that segues nicely to the last two functions found in the core Perl distribution I want to mention before we move on: strftime and mktime. The POSIX::strftime() function takes a format string followed by a list of values like those returned by localtime() and gmtime() and returns the appropriately formatted string. The example the documentation provides is:

```
use POSIX;
$str = POSIX::strftime( "%A, %B %d, %Y", 0, 0, 0, 12, 11, 95, 2 );
print "$str\n"; # prints "Tuesday, December 12, 1995"
```

If you wanted to return the number of seconds since the epoch, you would use POSIX::mktime() instead (again from the POSIX module docs):

```
use POSIX;
$time_t = POSIX::mktime( 33, 26, 12, 27, 10, 109 );
```

Keep It Small, Fast, and Simple

That's the motto of the module Date::Calc. So far we've just seen ways to bring time information into our programs but no real information on how to manipulate it. Simple calculations such as "Find the time exactly an hour from now" are easy, but how about "What's the first Monday in December of this year?" With Date::Calc, the answer is:

```
use Date::Calc qw(Nth_Weekday_of_Month_Year);
# year, month, day of week, Nth occurrence
print Nth_Weekday_of_Month_Year(2009,12,1,1); # prints "2009127"
```

Date::Calc has a ton of functions like:

- - leap_year($year) to determine if the year is a leap year
- - Monday_of_Week($week,$year) to determine the first day of that week
- - Add_Delta_Days($year,$month,$day, $Dd) to add $Dd days to the date

That's just a few, so be sure to see the documentation for the full scoop. I'd especially recommend you check out the Recipe section of the documentation, which offers you ways to answer questions like "How can I calculate the last business day (payday!) of a month?" and "How can I send a re-

minder to members of a group on the day before a meeting which occurs every first Friday of a month?"

There's a considerable amount of power to be found in this module, but it does have its drawbacks. The first is a dependence on a C compiler (ideally the C compiler the Perl binary was built with) to build the module. Key sections are written in C, which is how it manages to provide at least the "fast" in its motto. There are a number of situations where this requirement would rule out the use of the module, so the Date::Calc author has also re-implemented those sections in pure Perl. Date::Pcalc is the result, trading speed for portability and deployability. Date::Calc is also fairly low-level in its abstraction. Although it has an optional OOP wrapper module, on the whole the module doesn't offer a particularly object-oriented shiny glint by default. At the very least, you still have to think in terms of putting together more complex calculations using chains of smaller operations (which, in general, I think is a good approach). For more DoWhatIMean sauce, we're going to have to look elsewhere.

Simple Representations

One way to provide more DoWhatIMean-itude (okay, I promise that's the last time I use that phrase, at least in this column) is to adopt a better representation of the very things we've been talking about, namely dates and times. If we can somehow take more natural descriptions of them (e.g., "2009-11-29") and work with those descriptions in a relatively intuitive manner, it will make things much more pleasant. Certainly more pleasant than trying to work with the number of seconds since the birthday of the actor who played "Mitch" in the movie *Real Genius* (or some such other arbitrary point in time).

There are a number of modules that provide this sort of representation. For working strictly with dates (without times), one of the simplest is, you guessed it, Date::Simple. Date::Simple lets you write code like:

```
use Date::Simple (':all');
my $date = Date::Simple->new('2010-01-29');
print $date->year;          # prints 2010
print today();              # prints '2009-11-29' when this was written
print date('2009-12-31') - date(today());      # prints 32 on that day
```

If you are more interested in working with times and dates, we can come back to a concept we saw earlier with Time::localtime. The Time::Piece module "replaces the standard localtime and gmtime functions with implementations that return objects." This means if you start with:

```
use Time::Piece;
my $tobj = localtime;
```

you have an object that provides a substantial number of methods, such as:

```
print $tobj->min;
print $tobj->year;
print $tobj->monname;           # prints Nov
print $tobj->fullmonth;         # prints November
print $tobj->time;              # prints 23:32:00
print $tobj->date;              # prints 2009-11-29
print $tobj->day_of_year;
print $tobj->month_last_day;    # prints the last day of the month
```

Hopefully, you read along that list and noted that it got more interesting and powerful as the list went on. Besides the very legible OOP-ness of the above, we also get the chance to work with the objects in a reasonable fashion to do arithmetic. For example, let's assume we have two Time::Piece objects:

```
$diff = $tobj1 - $tobj2;
```

If we just printed $diff, we'd get the number of seconds between the time/dates those objects represented. But even cooler than that, $diff is actually an object itself. It's a Time::Seconds object (as defined by the Time::Seconds module in the Time::Piece distribution). Why is that cooler? Well, it means we get to call methods on the resulting object like $diff->seconds, $diff->minutes, $diff->days, $diff->weeks, $diff->months, and even $diff->years. It's kind of nice to be able to work with these representations without needing to do the conversion arithmetic by hand. Time::Piece can do other neat stuff (e.g., date comparisons); see the documentation for more details.

If both the Date::Simple and the Time::Piece abstractions appeal to you and you feel torn deciding between the two, you may find that Date::Piece can help with that inner struggle. According to the documentation, it "extends Date::Simple and connects it to Time::Piece." What this means in practice is you can work with just dates and then pin your result down to a specific time on a date. The example the documentation gives is:

```
use Date::Piece qw(date);

my $date = date('2007-11-22');
my $time = $date->at('16:42:35');
print $time, "\n"; # is a Time::Piece
```

More Complex Frameworks

We have two more stops on this tour. There are two larger date/time frameworks that have found favor in the Perl community. Both are all-encompassing and will do everything you could possibly want, stopping short only at making you breakfast. They have their own way of looking at the world of dates and times, which is why I think it is good to look at both to find one that meshes well with the way you want to work.

The first, Date::Manip, really made its reputation in the community on the strength of its date-parsing skills. It's all well and good to talk about working with more natural representations of dates and times, but all of the examples we've given so far assume the programmer gets to pull those representations out of thin air. Equally often, we get handed a file, be it a logfile or some other date/time-laden glob of data, and our first task is to somehow translate its representation of dates and times into a form we can manipulate. For simple date parsing, a module like Date::Parse in the TimeDate distribution will work well. For the more complex stuff, there's very little that can touch Date::Manip. In addition to handling computer-y dates like those we've seen (e.g., "Mon Nov 30 22:30:46 2009"), it will happily parse strings like:

```
January 30
2001-01-01
Mar052009
Dec 1st 1970
next year
last Wednesday in December
3rd Tuesday in September
last Tuesday in 1973
```

```
14th
today
yesterday
Jan 2 2009 at noon
in 3 days at midnight
2 weeks ago on Friday at 10:00
```

Date::Manip is much more than just a fancy date parser. The documentation says:

> Among other things, Date::Manip allows you to:
>
> - Enter a date in practically any format you choose.
> - Compare two dates, entered in widely different formats to determine which is earlier.
> - Extract any information you want from ANY date using a format string similar to the UNIX date command.
> - Determine the amount of time between two dates, or add an amount of time to a date to get a second date.
> - Work with dates using international formats (foreign month names, e.g., 12/10/95 referring to October rather than December).
> - Find a list of dates where a recurring event happens

To do all of this, Date::Manip concerns itself with dates, deltas (amount of time between dates), and recurrences. You can construct objects that represent each and rub them together in ways that make sense (e.g., apply deltas to dates). Both this framework and the next one we're going to see are swimming in documentation, so I won't go very far into how you actually work with the module. Here's a very small example of Date::Manip code:

```
use Date::Manip::Date;

my $date = new Date::Manip::Date;
my $err = $date->parse('in 3 days at midnight');
print $date->printf('%C'); # prints 'Fri Dec  4 00:00:00 EST 2009'
```

Looks pretty simple on the surface, and that's probably a good thing.

So what's not to like? To answer this question, I have to repeat some of Date::Manip's history before I get to pile on the caveats. Once upon a time, Date::Manip was a very large, monolithic module that was known not only for its parsing power but also less positively for its memory requirements and comparatively slow speeds (at least when compared to some of the other modules that we've looked at). Previous versions provided a strictly non-OOP interface, and that made some people unhappy too.

The author took all of these things to heart and embarked on a total rewrite, going from version 5.x to 6.0x to mark the change. Date::Manip 6.0x is still written entirely in Perl, which means it isn't going to make Speedy Gonzales look like Regular Gonzales (as they say on Futurama), but the other criticisms have largely been, or are being, addressed quite well. It has a new OOP interface that can do everything the old functional one can and more. It's now a set of modules so you can load what you need, and so on. One parting caveat that is important to mention before moving on: the 6.0x branch requires Perl 5.10.x. If you still haven't upgraded to the latest stable version of Perl, you will need to use an older 5.x version of Date::Manip.

If Date::Manip doesn't excite you, perhaps our last module framework, DateTime, will. In a previous column, I mentioned the Perl Email Project. That was an attempt to replace the scattered functionality found in existing email modules with a new set of simple, well-architected, and unified packages that could become the definitive way of handling mail in Perl. A similar

effort was undertaken to do the same with time and date handling. The result was the DateTime project (http://datetime.perl.org) which has yielded a whole framework of DateTime::* modules for date/time representation, manipulation, parsing, and so on.

The main module, DateTime, lets you write code like this:

```
use DateTime;
my $dt = DateTime->new(  year       => 1973,
                         month      => 9,
                         day        => 15,
                         hour       => 23,
                         minute     => 10,
                         second     => 0,
                         nanosecond => 0,
                         time_zone  => 'America/New_York',
                         );
print $dt->year;
print $dt->day;
```

Does that code look a little familiar? What if I add this code to the previous example:

```
my $dt1 = DateTime->now();
my $diff = $dt1 - $dt;
my ($years,$months) = $diff->in_units('years', 'months');
  # prints '36 years and 2 months old'
print "$years years and $months months old\n";
```

Yes, we've seen this sort of date/duration object representation and date calculation in previous sections of this column. The syntax is a little different, but besides one small twist (the '-' is overloaded so we can actually subtract one object from another), it's the same song.

The core DateTime module doesn't have all of the functionality we've seen in other modules, but it doesn't have to. It is meant to glue together with other modules in this framework. For example, DateTime makes it possible to load what it calls formatters that provide new format parsers and output routines. This makes it possible, for example, to extend DateTime to handle some of Date::Manip's impressive formats (courtesy of DateTime::Format::Natural) and some more, umm, esoteric ones such as "The big hand is on the twelve and the little hand is on the six" and "La grande aiguille est sur le douze et la petite aiguille est sur le six" (courtesy of "DateTime::Format::Baby"). See the datetime.perl.org Web site for a listing of other extension modules and further documentation on the module.

I'm not going to run out of time to talk about date and time handling in Perl, but I'm certainly going to run out of space, so let's end things here.

Take care, and I'll see you next time.