NAME

Encode::Encoder -- Object Oriented Encoder

SYNOPSIS

```
use Encode::Encoder;
# Encode::encode("ISO-8859-1", $data);
Encode::Encoder->new($data)->iso_8859_1; # OOP way
# shortcut
use Encode::Encoder qw(encoder);
encoder($data)->iso_8859_1;
# you can stack them!
encoder($data)->iso_8859_1->base64; # provided base64() is defined
# you can use it as a decoder as well
encoder($base64)->bytes('base64')->latin1;
# stringified
print encoder($data)->utf8->latin1; # prints the string in latin1
# numified
encoder("\x{abcd}\x{ef}g")->utf8 == 6; # true. bytes::length($data)
```

ABSTRACT

Encode::Encoder allows you to use Encode in an object-oriented style. This is not only more intuitive than a functional approach, but also handier when you want to stack encodings. Suppose you want your UTF-8 string converted to Latin1 then Base64: you can simply say

```
my $base64 = encoder($utf8)->latin1->base64;
```

instead of

```
my $latin1 = encode("latin1", $utf8);
my $base64 = encode_base64($utf8);
```

or the lazier and more convoluted

```
my $base64 = encode_base64(encode("latin1", $utf8));
```

Description

Here is how to use this module.

- There are at least two instance variables stored in a hash reference, {data} and {encoding}.
- When there is no method, it takes the method name as the name of the encoding and encodes the instance *data* with *encoding*. If successful, the instance *encoding* is set accordingly.
- You can retrieve the result via ->data but usually you don't have to because the stringify operator () is overridden to do exactly that.

Predefined Methods

This module predefines the methods below:

`$e` = Encode::Encoder->new([$data, `$encoding`]);
: returns an encoder object. Its data is initialized with `$data` if present, and its encoding is set to `$encoding` if present.

  When `$encoding` is omitted, it defaults to utf8 if `$data` is already in utf8 or (empty string) otherwise.

encoder()
: is an alias of Encode::Encoder->*new()*. This one is exported on demand.

`$e`->data([$data])
: When `$data` is present, sets the instance data to `$data` and returns the object itself. Otherwise, the current instance data is returned.

`$e->`encoding([$encoding])
: When `$encoding` is present, sets the instance encoding to `$encoding` and returns the object itself. Otherwise, the current instance encoding is returned.

`$e->`bytes([$encoding])
: decodes instance data from `$encoding`, or the instance encoding if omitted. If the conversion is successful, the instance encoding will be set to .

  The name *bytes* was deliberately picked to avoid namespace tainting — this module may be used as a base class so method names that appear in Encode::Encoding are avoided.

Example: base64 transcoder

This module is designed to work with Encode::Encoding. To make the Base64 transcoder example above really work, you could write a module like this:

```
package Encode::Base64;
use parent 'Encode::Encoding';
__PACKAGE__->Define('base64');
use MIME::Base64;
sub encode{
my ($obj, $data) = @_;
return encode_base64($data);
}
sub decode{
my ($obj, $data) = @_;
return decode_base64($data);
}
1;
__END__
```

And your caller module would be something like this:

```
use Encode::Encoder;
use Encode::Base64;

# now you can really do the following

encoder($data)->iso_8859_1->base64;
encoder($base64)->bytes('base64')->latin1;
```

Operator Overloading

This module overloads two operators, stringify () and numify (0+).

Stringify dumps the data inside the object.

Numify returns the number of bytes in the instance data.

They come in handy when you want to print or find the size of data.

SEE ALSO

Encode, Encode::Encoding