NAME

Mail::Message::Field − one line of a message header

INHERITANCE

```
Mail::Message::Field
  is a Mail::Reporter

Mail::Message::Field is extended by
  Mail::Message::Field::Fast
  Mail::Message::Field::Flex
  Mail::Message::Field::Full
```

SYNOPSIS

```
my $field = Mail::Message::Field->new(From => 'fish@tux.aq');
print $field->name;
print $field->body;
print $field->comment;
print $field->content;  # body & comment
$field->print(\*OUT);
print $field->string;
print "$field\n";
print $field->attribute('charset') || 'us-ascii';
```

DESCRIPTION

This implementation follows the guidelines of rfc2822 as close as possible, and may there produce a different output than implementations based on the obsolete rfc822. However, the old output will still be accepted.

These objects each store one header line, and facilitates access routines to the information hidden in it. Also, you may want to have a look at the added methods of a message:

```
my @from    = $message->from;
my $sender  = $message->sender;
my $subject = $message->subject;
my $msgid   = $message->messageId;

my @to      = $message->to;
my @cc      = $message->cc;
my @bcc     = $message->bcc;
my @dest    = $message->destinations;

my $other   = $message->get('Reply-To');
```

Extends ‘‘DESCRIPTION’’ in Mail::Reporter.

OVERLOADED

overload: **“”**

(stringification) produces the unfolded body of the field, which may be what you expect. This is what makes what the field object seems to be a simple string. The string is produced by **unfoldedBody()**.

example:

```
print $msg->get('subject');  # via overloading
print $msg->get('subject')->unfoldedBody; # same

my $subject = $msg->get('subject') || 'your mail';
print "Re: $subject\n";
```

overload: **0+**
(numification) When the field is numeric, the value will be returned. The result is produced by **toInt()**. If the value is not correct, a `0` is produced, to simplify calculations.

overload: <=>
(numeric comparison) Compare the integer field contents with something else.

example:

```
if($msg->get('Content-Length') > 10000) ...
if($msg->size > 10000) ... ; # same, but better
```

overload: **bool**
Always true, to make it possible to say `if($field)`.

overload: **cmp**
(string comparison) Compare the unfolded body of a field with another field or a string, using the buildin `cmp`.

METHODS

Extends ‘‘METHODS’’ in Mail::Reporter.

Constructors

Extends ‘‘Constructors’’ in Mail::Reporter.

`$obj`->**clone**()
Create a copy of this field object.

Mail::Message::Field->**new**($data)
See **Mail::Message::Field::Fast::new()**, **Mail::Message::Field::Flex::new()**, and **Mail::Message::Field::Full::new()**. By default, a `Fast` field is produced.

```
-Option--Defined in     --Default
 log     Mail::Reporter   'WARNINGS'
 trace   Mail::Reporter   'WARNINGS'
```

log => LEVEL
trace => LEVEL

The field

`$obj`->**isStructured**()
Mail::Message::Field->**isStructured**()
Some fields are described in the RFCs as being *structured*: having a well described syntax. These fields have common ideas about comments and the like, what they do not share with unstructured fields, like the `Subject` field.

example:

```
my $field = Mail::Message::Field->new(From => 'me');
if($field->isStructured)

Mail::Message::Field->isStructured('From');
```

`$obj`->**length**()
Returns the total length of the field in characters, which includes the field’s name, body and folding characters.

`$obj`->**nrLines**()
Returns the number of lines needed to display this header-line.

`$obj`->**print**([$fh])
Print the whole header-line to the specified file-handle. One line may result in more than one printed line, because of the folding of long lines. The `$fh` defaults to the selected handle.

`$obj->`**size**()
Returns the number of bytes needed to display this header-line, Same as **length()**.

`$obj->`**string**([$wrap])
Returns the field as string. By default, this returns the same as **folded()**. However, the optional `$wrap` will cause to re-fold to take place (without changing the folding stored inside the field).

`$obj->`**toDisclose**()
Returns whether this field can be disclosed to other people, for instance when sending the message to another party. Returns a`true` or `false` condition. See also **Mail::Message::Head::Complete::printUndisclosed()**.

Access to the name

`$obj->`**Name**()
Returns the name of this field in original casing. See **name()** as well.

`$obj->`**name**()
Returns the name of this field, with all characters lower-cased for ease of comparison. See **Name()** as well.

`$obj->`**wellformedName**([STRING])
(Instance method class method) As instance method, the current field's name is correctly formatted and returned. When a STRING is used, that one is formatted.

example:

```
print Mail::Message::Field->Name('content-type')
  # -->  Content-Type

my $field = $head->get('date');
print $field->Name;
  # -->  Date
```

Access to the body

`$obj->`**body**()
This method may be what you want, but usually, the **foldedBody()** and **unfoldedBody()** are what you are looking for. This method is cultural heritage, and should be avoided.

Returns the body of the field. When this field is structured, it will be **stripped** from everything what is behind the first semi-color (`;`). In any case, the string is unfolded. Whether the field is structured is defined by **isStructured()**.

`$obj->`**folded**()
Returns the folded version of the whole header. When the header is shorter than the wrap length, a list of one line is returned. Otherwise more lines will be returned, all but the first starting with at least one blank. See also**f oldedBody()** to get the same information without the field's name.

In scalar context, the lines are delived into one string, which is a little faster because that's the way they are stored internally...

example:

```
my @lines = $field->folded;
print $field->folded;
print scalar $field->folded; # faster
```

`$obj->`**foldedBody**([$body])
Returns the body as a set of lines. In scalar context, this will be one line containing newlines. Be warned about the newlines when you do pattern matching on the result of this method.

The optional `$body` argument changes the field's body. The folding of the argument must be correct.

`$obj->`**stripCFWS**([STRING])
Mail::Message::Field->**stripCFWS**([STRING])
Remove the *comments* and *folding white spaces* from the STRING. Without string and only as instance method, the **unfoldedBody()** is being stripped and returned.

WARNING: This operation is only allowed for structured header fields (which are defined by the various RFCs as being so. You don't want parts within braces which are in the Subject header line to be removed, to give an example.

`$obj->`**unfoldedBody**([$body, [$wrap]])
Returns the body as one single line, where all folding information (if available) is removed. This line will also NOT end on a new-line.

The optional `$body` argument changes the field's body. The right folding is performed before assignment. The`$wrap` may be specified to enforce a folding size.

example:

```
my $body = $field->unfoldedBody;
print "$field";   # via overloading
```

Access to the content

`$obj->`**addresses**()
Returns a list of Mail::Address objects, which represent the e-mail addresses found in this header line.

example:

```
my @addr = $message->head->get('to')->addresses;
my @addr = $message->to;
```

`$obj->`**attribute**(`$name`, [$value])
Get the value of an attribute, optionally after setting it to a new value. Attributes are part of some header lines, and hide themselves in the comment field. If the attribute does not exist, then `undef` is returned. The attribute is still encoded.

example:

```
my $field = Mail::Message::Field->new(
 'Content-Type: text/plain; charset="us-ascii"');

print $field->attribute('charset');
  # --> us-ascii

print $field->attribute('bitmap') || 'no'
  # --> no

$field->atrribute(filename => '/tmp/xyz');
$field->print;
  # --> Content-Type: text/plain; charset="us-ascii";
  #       filename="/tmp/xyz"
  # Automatically folded, and no doubles created.
```

`$obj->`**attributes**()
Returns a list of key-value pairs, where the values are not yet decoded. Keys may appear more than once.

example:

```
my @pairs = $head->get('Content-Disposition')->attributes;
```

`$obj->`**comment**([STRING])
Returns the unfolded comment (part after a semi-colon) in a structureed header-line. optionally after setting it to a new STRING first. When `undef` is specified as STRING, the comment is removed. Whether the field is structured is defined by **isStructured()**.

The *comment* part of a header field often contains `attributes`. Often it is preferred to use **attribute()** on them.

`$obj->`**study**()
Study the header field in detail: turn on the full parsing and detailed understanding of the content of the fields. Mail::Message::Field::Fast and Mail::Message::Field::Fast objects will be transformed into any Mail::Message::Field::Full object.

example:

```
my $subject = $msg->head->get('subject')->study;
my $subject = $msg->head->study('subject');  # same
my $subject = $msg->study('subject');        # same
```

`$obj->`**toDate**([$time])
Mail::Message::Field->**toDate**([$time])
Convert a timestamp into an rfc2822 compliant date format. This differs from the default output of `localtime` in scalar context. Without argument, the `localtime` is used to get the current time. `$time` can be specified as one numeric (like the result of `time()`) and as list (like produced by c<**localtime()**> in list context).

Be sure to have your timezone set right, especially when this script runs automatically.

example:

```
my $now = time;
Mail::Message::Field->toDate($now);
Mail::Message::Field->toDate(time);

Mail::Message::Field->toDate(localtime);
Mail::Message::Field->toDate;      # same
# returns something like:
#     Wed, 28 Aug 2002 10:40:25 +0200
```

`$obj->`**toInt**()
Returns the value which is related to this field as integer. A check is performed whether this is right.

Other methods

`$obj->`**dateToTimestamp**(STRING)
Mail::Message::Field->**dateToTimestamp**(STRING)
Convert a STRING which represents and RFC compliant time string into a timestamp like is produced by the `time` function.

Internals

`$obj->`**consume**(`$line` | <$name,<$body|$objects>>)
Accepts a whole field `$line`, or a pair with the field's `$name` and `$body`. In the latter case, the `$body` data may be specified as array of `$objects` which are stringified. Returned is a nicely formatted pair of two strings: the field's name and a folded body.

This method is called by **new()**, and usually not by an application program. The details about converting the `$objects` to a field content are explained in ''Specifying field data''.

`$obj->`**defaultWrapLength**([$length])
Any field from any header for any message will have this default wrapping. This is maintained in one global variable. Without a specified `$length`, the current value is returned. The default is 78.

`$obj`->**fold**(`$name`, `$body`, [$maxchars])
Mail::Message::Field->**fold**(`$name`, `$body`, [$maxchars])
Make the header field with `$name` fold into multiple lines. Wrapping is performed by inserting newlines before a blanks in the `$body`, such that no line exceeds the `$maxchars` and each line is as long as possible.

The RFC requests for folding on nice spots, but this request is mainly ignored because it would make folding too slow.

`$obj`->**setWrapLength**([$length])
Force the wrapping of this field to the specified `$length` characters. The wrapping is performed with **fold()** and the results stored within the field object.

example: refolding the field

```
$field->setWrapLength(99);
```

`$obj`->**stringifyData**(STRING|ARRAY|$objects)
This method implements the translation of user supplied objects into ascii fields. The process is explained in ''Specifying field data''.

`$obj`->**unfold**(STRING)
The reverse action of **fold()**: all lines which form the body of a field are joined into one by removing all line terminators (even the last). Possible leading blanks on the first line are removed as well.

Error handling
Extends ''Error handling'' in Mail::Reporter.

`$obj`->**AUTOLOAD**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**addReport**($object)
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**defaultTrace**([$level]|[$loglevel, `$tracelevel`]|[$level, `$callback`])
Mail::Message::Field->**defaultTrace**([$level]|[$loglevel, `$tracelevel`]|[$level, `$callback`])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**errors**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**log**([$level, [$strings]])
Mail::Message::Field->**log**([$level, [$strings]])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**logPriority**($level)
Mail::Message::Field->**logPriority**($level)
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**logSettings**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**notImplemented**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**report**([$level])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**reportAll**([$level])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**trace**([$level])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj->`**warnings**()
Inherited, see ‘‘Error handling’’ in Mail::Reporter

Cleanup
Extends ‘‘Cleanup’’ in Mail::Reporter.

`$obj->`**DESTROY**()
Inherited, see ‘‘Cleanup’’ in Mail::Reporter

DETAILS

Field syntax

Fields are stored in the header of a message, which are represented by Mail::Message::Head objects. A field is a combination of a *name*, *body*, and *attributes*. Especially the term ‘‘body’’ is cause for confusion: sometimes the attributes are considered to be part of the body.

The name of the field is followed by a colon (":“, not preceded by blanks, but followed by one blank). Each attribute is preceded by a separate semi-colon (”;"). Names of fields are case-insensitive and cannot contain blanks.

. Example: of fields

Correct fields:

```
Field: hi!
Content-Type: text/html; charset=latin1
```

Incorrect fields, but accepted:

```
Field : wrong, blank before colon
Field:                  # wrong, empty
Field:not nice, blank preferred after colon
One Two: wrong, blank in name
```

Folding fields

Fields which are long can be folded to span more than one line. The real limit for lines in messages is only at 998 characters, however such long lines are not easy to read without support of an application. Therefore rfc2822 (which defines the message syntax) specifies explicitly that field lines can be re-formatted into multiple sorter lines without change of meaning, by adding new-line characters to any field before any blank or tab.

Usually, the lines are reformatted to create lines which are 78 characters maximum. Some applications try harder to fold on nice spots, like before attributes. Especially the`Received` field is often manually folded into some nice layout. In most cases however, it is preferred to produce lines which are as long as possible but max 78.

BE WARNED that all fields can be subjected to folding, and that you usually want the unfolded value.

. Example: of field folding

```
Subject: this is a short line, and not folded

Subject: this subject field is much longer, and therefore
 folded into multiple
 lines, although one more than needed.
```

Structured fields

The rfc2822 describes a large number of header fields explicitly. These fields have a defined meaning. For some of the fields, like the `Subject` field, the meaning is straight forward the contents itself. These fields are the *Unstructured Fields*.

Other fields have a well defined internal syntax because their content is needed by e−mail applications. For instance, the `To` field contains addresses which must be understood by all applications in the same way. These are the *Structured Fields*, see **isStructured()**.

Comments in fields

Stuctured fields can contain comments, which are pieces of text enclosed in parenthesis. These comments can be placed close to anywhere in the line and must be ignored be the application. Not all applications are capable of handling comments correctly in all circumstances.

. Example: of field comments

```
To: mailbox (Mail::Box mailinglist) <mailbox@overmeer.net>
Date: Thu, 13 Sep 2001 09:40:48 +0200 (CEST)
Subject: goodbye (was: hi!)
```

On the first line, the text ‘‘Mail::Box mailinglist’’ is used as comment. Be warned that rfc2822 explicitly states that comments in e−mail address specifications should not be considered to contain any usable information.

On the second line, the timezone is specified as comment. The `Date` field format has no way to indicate the timezone of the sender, but only contains the timezone difference to UTC, however one could decide to add this as comment. Application must ignore this data because the `Date` field is structured.

The last field is unstructured. The text between parentheses is an integral part of the subject line.

Getting a field

As many programs as there are handling e−mail, as many variations on accessing the header information are requested. Be careful which way you access the data: read the variations described here and decide which solution suites your needs best.

*Using **get()** field*

The `get()` interface is copied from other Perl modules which can handle e−mail messages. Many applications which simply replace Mail::Internet objects by Mail::Message objects will work without modification.

There is more than one get method. The exact results depend on which get you use. When **Mail::Message::get()** is called, you will get the unfolded, stripped from comments, stripped from attributes contents of the field as **string**. Character-set encodings will still be in the string. If the same fieldname appears more than once in the header, only the last value is returned.

When **Mail::Message::Head::get()** is called in scalar context, the last field with the specified name is returned as field **object**. This object strinigfies into the unfolded contents of the field, including attributes and comments. In list context, all appearances of the field in the header are returned as objects.

BE WARNED that some lines seem unique, but are not according to the official rfc. For instance, `To` fields can appear more than once. If your program calls `get('to')` in scalar context, some information is lost.

. Example: of using get()

```
print $msg−>get('subject') || 'no subject';
print $msg−>head−>get('subject') || 'no subject';

my @to = $msg−>head−>get('to');
```

*Using **study()** field*

As the name `study` already implies, this way of accessing the fields is much more thorough but also slower. The`study` of a field is lik e a `get`, but provides easy access to the content of the field and handles character-set decoding correctly.

The **Mail::Message::study()** method will only return the last field with that name as object. **Mail::Message::Head::study()** and **Mail::Message::Field::study()** return all fields when used in list context.

. Example: of using study()

```
print $msg->study('subject') || 'no subject';
my @rec  = $msg->head->study('Received');

my $from = $msg->head->get('From')->study;
my $from = $msg->head->study('From');  # same
my @addr = $from->addresses;
```

Using resent groups

Some fields belong together in a group of fields. For instance, a set of lines is used to define one step in the mail transport process. Each step adds a `Received` line, and optionally some `Resent-*` lines and `Return-Path`. These groups of lines shall stay together and in order when the message header is processed.

The `Mail::Message::Head::ResentGroup` object simplifies the access to these related fields. These resent groups can be deleted as a whole, or correctly constructed.

. Example: of using resent groups

```
my $rgs = $msg->head->resentGroups;
$rgs[0]->delete if @rgs;

$msg->head->removeResentGroups;
```

The field's data

There are many ways to get the fields info as object, and there are also many ways to process this data within the field.

Access to the field

- **string()**

  Returns the text of the body exactly as will be printed to file when **print()** is called, so name, main body, and attributes.

- **foldedBody()**

  Returns the text of the body, like **string()**, but without the name of the field.

- **unfoldedBody()**

  Returns the text of the body, like **foldedBody()**, but then with all new-lines removed. This is the normal way to get the content of unstructured fields. Character-set encodings will still be in place. Fields are stringified into their unfolded representation.

- **stripCFWS()**

  Returns the text of structured fields, where new-lines and comments are removed from the string. This is a good start for parsing the field, for instance to find e-mail addresses in them.

- **Mail::Message::Field::Full::decodedBody()**

  Studied fields can produce the unfolded text decoded into utf8 strings. This is an expensive process, but the only correct way to get the field's data. More useful for people who are not living in ASCII space.

- Studied fields

  Studied fields have powerful methods to provide ways to access and produce the contents of (structured) fields exactly as the involved rfcs prescribe.

Using simplified field access

Some fields are accessed that often that there are support methods to provide simplified access. All these methods are called upon a message directly.

. Example: of simplified field access

```
print $message->subject;
print $message->get('subject') || '';  # same

my @from = $message->from; # returns addresses
$message->reply->send if $message->sender;
```

The `sender` method will return the address specified in the `Sender` field, or the first named in the `From` field. It will return`undef` in case no address is kno wn.

Specifying field data

Field data can be anything, strongly dependent on the type of field at hand. If you decide to construct the fields very carefully via some Mail::Message::Field::Full extension (like via Mail::Message::Field::Addresses objects), then you will have protection build-in. However, you can bluntly create any Mail::Message::Field object based on some data.

When you create a field, you may specify a string, object, or an array of strings and objects. On the moment, objects are only used to help the construction on e-mail addresses, however you may add some of your own.

The following rules (implemented in **stringifyData()**) are obeyed given the argument is:

- a string

  The string must be following the (complicated) rules of the rfc2822, and is made field content as specified. When the string is not terminated by a new-line (`"\n"`) it will be folded according to the standard rules.

- a Mail::Address object

  The most used Perl object to parse and produce address lines. This object does not understand character set encodings in phrases.

- a Mail::Identity object

  As part of the User::Identity distribution, this object has full understanding of the meaning of one e-mail address, related to a person. All features defined by rfc2822 are implemented.

- a User::Identity object

  A person is specified, which may have more than one Mail::Identity's defined. Some methods, like **Mail::Message::reply()** and **Mail::Message::forward()** try to select the right e-mail address smart (see their method descriptions), but in other cases the first e-mail address found is used.

- a User::Identity::Collection::Emails object

  All Mail::Identity objects in the collection will be included in the field as a group carying the name of the collection.

- any other object

  For all other objects, the stringification overload is used to produce the field content.

- an ARRAY

  You may also specify an array with a mixture of any of the above. The elements will be joined as comma-separated list. If you do not want comma's inbetween, you will have to process the array yourself.

. Example: specifying simple field data

```
my $f = Mail::Message::Field->new(Subject => 'hi!');
my $b = Mail::Message->build(Subject => 'monkey');
```

. Example: s specifying e-mail addresses for a field

```
use Mail::Address;
my $fish = Mail::Address->new('Mail::Box', 'fish@tux.aq');
print $fish->format;   # ==> Mail::Box <fish@tux.aq>
my $exa  = Mail::Address->new(undef, 'me@example.com');
print $exa->format;    # ==> me@example.com

my $b = $msg->build(To => "you@example.com");
my $b = $msg->build(To => $fish);
my $b = $msg->build(To => [ $fish, $exa ]);

my @all = ($fish, "you@example.com", $exa);
my $b = $msg->build(To => \@all);
my $b = $msg->build(To => [ "xyz", @all ]);
```

. Example: specifying identities for a field

```
use User::Identity;
my $patrik = User::Identity->new
 ( name      => 'patrik'
 , full_name => "Patrik Fältström"  # from rfc
 , charset   => "ISO-8859-1"
 );
$patrik->add
 ( email    => "him@home.net"
 );

my $b = $msg->build(To => $patrik);

$b->get('To')->print;
  # ==> =?ISO-8859-1?Q?Patrik_F=E4ltstr=F6m?=
  #     <him@home.net>
```

Field class implementation

For performance reasons only, there are three types of fields: the fast, the flexible, and the full understander:

- Mail::Message::Field::Fast

  `Fast` objects are not derived from a `Mail::Reporter`. The consideration is that fields are so often created, and such a small objects at the same time, that setting-up a logging for each of the objects is relatively expensive and not really useful. The fast field implementation uses an array to store the data: that will be faster than using a hash. Fast fields are not easily inheritable, because the object creation and initiation is merged into one method.

- Mail::Message::Field::Flex

  The flexible implementation uses a hash to store the data. The **new()** and `init` methods are split, so this object is extensible.

- Mail::Message::Field::Full

  With a full implementation of all applicable RFCs (about 5), the best understanding of the fields is reached. However, this comes with a serious memory and performance penalty. These objects are created from fast or flex header fields when **study()** is called.

DIAGNOSTICS

Warning: Field content is not numerical: `$content`

The numeric value of a field is requested (for instance the `Lines` or `Content-Length` fields should be numerical), however the data contains weird characters.

Warning: Illegal character in field name `$name`

A new field is being created which does contain characters not permitted by the RFCs. Using this field in messages may break other e−mail clients or transfer agents, and therefore mutulate or extinguish your message.

Error: Package `$package` does not implement `$method`.

Fatal error: the specific package (or one of its superclasses) does not implement this method where it should. This message means that some other related classes do implement this method however the class at hand does not. Probably you should investigate this and probably inform the author of the package.

SEE ALSO

This module is part of Mail-Message distribution version 3.012, built on February 11, 2022. Website: *http://perl.overmeer.net/CPAN/*

LICENSE