NAME

Mail::Message − general message object

INHERITANCE

```
Mail::Message has extra code in
  Mail::Message::Construct
  Mail::Message::Construct::Bounce
  Mail::Message::Construct::Build
  Mail::Message::Construct::Forward
  Mail::Message::Construct::Read
  Mail::Message::Construct::Rebuild
  Mail::Message::Construct::Reply
  Mail::Message::Construct::Text

Mail::Message
  is a Mail::Reporter

Mail::Message is extended by
  Mail::Box::Message
  Mail::Message::Dummy
  Mail::Message::Part
  Mail::Message::Replace::MailInternet
```

SYNOPSIS

```
use Mail::Box::Manager;
my $mgr    = Mail::Box::Manager->new;
my $folder = $mgr->open(folder => 'InBox');
my $msg    = $folder->message(2);    # $msg is a Mail::Message now

my $subject = $msg->subject;         # The message's subject
my @cc      = $msg->cc;              # List of Mail::Address'es

my Mail::Message::Head $head = $msg->head;
my Mail::Message::Body $body = $msg->decoded;
$msg->decoded->print($outfile);

# Send a simple email
Mail::Message->build
  ( To             => 'you@example.com'
  , From           => 'me@example.com'
  , Subject        => "My subject"
  , data           => "Some plain text content"
  )->send(via => 'postfix');

my $reply_msg = Mail::Message->reply(...);
my $frwd_msg  = Mail::Message->forward(...);
```

DESCRIPTION

A `Mail::Message` object is a container for MIME-encoded message information, as defined by RFC2822. Everything what is not specificaly related to storing the messages in mailboxes (folders) is implemented in this class. Methods which are related to folders is implemented in the Mail::Box::Message extension.

The main methods are **get()**, to get information from a message header field, and **decoded()** to get the intended content of a message. But there are many more which can assist your program.

Complex message handling, like construction of replies and forwards, are implemented in separate

packages which are autoloaded into this class. This means you can simply use these methods as if they are part of this class. Those package add functionality to all kinds of message objects.

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

METHODS

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

Constructors

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

`$obj->`**clone**(%options)

Create a copy of this message. Returned is a `Mail::Message` object. The head and body, the log and trace levels are taken. Labels are copied with the message, but the delete and modified flags are not.

BE WARNED: the clone of any kind of message (or a message part) will **always** be a `Mail::Message` object. For example, a Mail::Box::Message's clone is detached from the folder of its original. When you use **Mail::Box::addMessage()** with the cloned message at hand, then the clone will automatically be coerced into the right message type to be added.

See also **Mail::Box::Message::copyTo()** and **Mail::Box::Message::moveTo()**.

```
-Option      --Default
 shallow       <false>
 shallow_body  <false>
 shallow_head  <false>
```

shallow => BOOLEAN

When a shallow clone is made, the header and body of the message will not be cloned, but shared. This is quite dangerous: for instance in some folder types, the header fields are used to store folder flags. When one of both shallow clones change the flags, that will update the header and thereby be visible in both.

There are situations where a shallow clone can be used safely. For instance, when **Mail::Box::Message::moveTo()** is used and you are sure that the original message cannot get undeleted after the move.

shallow_body => BOOLEAN

A rather safe bet, because you are not allowed to modify the body of a message: you may only set a new body with **body()**.

shallow_head => BOOLEAN

Only the head uses is reused, not the body. This is probably a bad choice, because the header fields can be updated, for instance when labels change.

example:

```
$copy = $msg->clone;
```

Mail::Message->**new**(%options)

```
-Option    --Defined in     --Default
 body                        undef
 body_type                   Mail::Message::Body::Lines
 deleted                     <false>
 field_type                  undef
 head                        undef
 head_type                   Mail::Message::Head::Complete
 labels                      {}
 log         Mail::Reporter  'WARNINGS'
 messageId                   undef
 modified                    <false>
```

```
trace       Mail::Reporter   'WARNINGS'
trusted                      <false>
```

body => OBJECT
Instantiate the message with a body which has been created somewhere before the message is constructed. The OBJECT must be a sub-class of Mail::Message::Body. See also**body()** and **storeBody()**.

body_type => CLASS
Default type of body to be created for **readBody()**.

deleted => BOOLEAN
Is the file deleted from the start?

field_type => CLASS
head => OBJECT
Instantiate the message with a head which has been created somewhere before the message is constructed. The OBJECT must be a (sub−)class of Mail::Message::Head. See also **head()**.

head_type => CLASS
Default type of head to be created for **readHead()**.

labels => ARRAY|HASH
Initial values of the labels. In case of Mail::Box::Message's, this shall reflect the state the message is in. For newly constructed Mail::Message's, this may be anything you want, because **coerce()** will take care of the folder specifics once the message is added to one.

log => LEVEL
messageId => STRING
The id on which this message can be recognized. If none specified and not defined in the header −−but one is needed— there will be one assigned to the message to be able to pass unique message-ids between objects.

modified => BOOLEAN
Flags this message as being modified from the beginning on. Usually, modification is auto-detected, but there may be reasons to be extra explicit.

trace => LEVEL
trusted => BOOLEAN
Is this message from a trusted source? If not, the content must be checked before use. This checking will be performed when the body data is decoded or used for transmission.

Constructing a message

`$obj`−>**bounce**([<$rg_object|%options>])
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Bounce

Mail::Message−>**build**([$message|$part|$body], `$content`)
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Build

Mail::Message−>**buildFromBody**($body, [$head], `$headers`)
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Build

`$obj`−>**forward**(%options)
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Forward

`$obj`−>**forwardAttach**(%options)
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Forward

`$obj`−>**forwardEncapsulate**(%options)
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Forward

`$obj`−>**forwardInline**(%options)
Inherited, see ''Constructing a message'' in Mail::Message::Construct::Forward

`$obj->`**forwardNo**(%options)
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Forward

`$obj->`**forwardPostlude**()
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Forward

`$obj->`**forwardPrelude**()
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Forward

`$obj->`**forwardSubject**(STRING)
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Forward

Mail::Message->**read**($fh|STRING|SCALAR|ARRAY, `%options`)
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Read

`$obj->`**rebuild**(%options)
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Rebuild

`$obj->`**reply**(%options)
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Reply

`$obj->`**replyPrelude**([STRING|$field|$address|ARRAY-$of-$things])
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Reply

`$obj->`**replySubject**(STRING)
Mail::Message->**replySubject**(STRING)
Inherited, see ‘‘Constructing a message’’ in Mail::Message::Construct::Reply

The message

`$obj->`**container**()
If the message is a part of another message, `container` returns the reference to the containing body.

example:

```
my Mail::Message $msg = ...
return unless $msg->body->isMultipart;
my $part   = $msg->body->part(2);

return unless $part->body->isMultipart;
my $nested = $part->body->part(3);

$nested->container;  # returns $msg->body
$nested->toplevel;   # returns $msg
$msg->container;     # returns undef
$msg->toplevel;      # returns $msg
$msg->isPart;        # returns false
$part->isPart;       # returns true
```

`$obj->`**isDummy**()
Dummy messages are used to fill holes in linked-list and such, where only a message-id is known, but not the place of the header of body data.

This method is also available for Mail::Message::Dummy objects, where this will return `true`. On any extension of `Mail::Message`, this will return `false`.

`$obj->`**isPart**()
Returns true if the message is a part of another message. This is the case for Mail::Message::Part extensions of `Mail::Message`.

`$obj->`**messageId**()
Retrieve the message’s id. Every message has a unique message-id. This id is used mainly for recognizing discussion threads.

`$obj->`**partNumber**()

Returns a string representing the location of this part. In case the top message is a single message, 'undef' is returned. When it is a multipart, '1' up to the number of multiparts is returned. A multi-level nested part may for instance return '2.5.1'.

Usually, this string is very short. Numbering follows the IMAP4 design, see RFC2060 section 6.4.5.

`$obj->`**print**([$fh])

Print the message to the FILE-HANDLE, which defaults to the selected filehandle, without the encapsulation sometimes required by a folder type, like **write()** does.

example:

```
$message->print(\*STDERR);  # to the error output
$message->print;            # to the selected file

my $out = IO::File->new('out', 'w');
$message->print($out);      # no encapsulation: no folder
$message->write($out);      # with encapsulation: is folder.
```

`$obj->`**send**([$mailer], `%options`)

Transmit the message to anything outside this Perl program. Returns false when sending failed even after retries.

The optional `$mailer` is a Mail::Transport::Send object. When the `$mailer` is not specified, one will be created and kept as default for the next messages as well.

The `%options` are mailer specific, and a mixture of what is usable for the creation of the mailer object and the sending itself. Therefore, see for possible options **Mail::Transport::Send::new()** and **Mail::Transport::Send::send()**. That object also provides a `trySend()` method which gives more low-level control.

example:

```
$message->send;
```

is short (but little less flexibile) for

```
my $mailer = Mail::Transport::SMTP->new(@smtpopts);
$mailer->send($message, @sendopts);
```

See examples/send.pl in the distribution of Mail::Box.

example:

```
$message->send(via => 'sendmail')
```

`$obj->`**size**()

Returns an estimated size of the whole message in bytes. In many occasions, the functions which process the message further, for instance **send()** or **print()** will need to add/change header lines or add CR characters, so the size is only an estimate with a few percent margin of the real result.

The computation assumes that each line ending is represented by one character (like UNIX, MacOS, and sometimes Cygwin), and not two characters (like Windows and sometimes Cygwin). If you write the message to file on a system which uses CR and LF to end a single line (all Windows versions), the result in that file will be at least **nrLines()** larger than this method returns.

`$obj->`**toplevel**()

Returns a reference to the main message, which will be the current message if the message is not part of another message.

`$obj->`**write**([$fh])

Write the message to the FILE-HANDLE, which defaults to the selected `$fh`, with all surrounding information which is needed to put it correctly in a folder file.

In most cases, the result of `write` will be the same as with **print()**. The main exception is for Mbox folder messages, which will get printed with their leading 'From ' line and a trailing blank. Each line of their body which starts with 'From ' will have an '>' added in front.

The header

`$obj->`**bcc**()

Returns the addresses which are specified on the `Bcc` header line (or lines) A list of Mail::Address objects is returned. `Bcc` stands for *Blind Carbon Copy*: destinations of the message which are not listed in the messages actually sent. So, this field will be empty for received messages, but may be present in messages you construct yourself.

`$obj->`**cc**()

Returns the addresses which are specified on the `Cc` header line (or lines) A list of Mail::Address objects is returned. `Cc` stands for *Carbon Copy*; the people addressed on this line recei ve the message informational, and are usually not expected to reply on its content.

`$obj->`**date**()

Method has been removed for reasons of consistency. Use**timestamp()** or `$msg->head->get('Date')`.

`$obj->`**destinations**()

Returns a list of Mail::Address objects which contains the combined info of active `To`, `Cc`, and `Bcc` addresses. Double addresses are removed if detectable.

`$obj->`**from**()

Returns the addresses from the senders. It is possible to have more than one address specified in the `From` field of the message, according to the specification. Therefore a list of Mail::Address objects is returned, which usually has length 1.

If you need only one address from a sender, for instance to create a ''original message by'' line in constructed forwarded message body, then use **sender()**.

example: using **from()** to get all sender addresses

```
my @from = $message->from;
```

`$obj->`**get**($fieldname)

Returns the value which is stored in the header field with the specified name. The `$fieldname` is case insensitive. The*unfolded body* of the field is returned, stripped from an y attributes. See **Mail::Message::Field::body()**.

If the field has multiple appearances in the header, only the last instance is returned. If you need more complex handing of fields, then call **Mail::Message::Head::get()** yourself. See **study()** when you want to be smart, doing the better (but slower) job.

example: the **get()** short-cut for header fields

```
print $msg->get('Content-Type'), "\n";
```

Is equivalent to:

```
print $msg->head->get('Content-Type')->body, "\n";
```

`$obj->`**guessTimestamp**()

Return an estimate on the time this message was sent. The data is derived from the header, where it can be derived from the `date` and `received` lines. For MBox-like folders you may get the date from the from-line as well.

This method may return `undef` if the header is not parsed or only partially known. If you require a time, then use the **timestamp()** method, described below.

example: using **guessTimestamp()** to get a transmission date

```
print "Receipt ", ($message->timestamp || 'unknown'), "\n";
```

$obj->**head**([$head])

Return (optionally after setting) the `$head` of this message. The head must be an (sub-)class of Mail::Message::Head. When the head is added, status information is taken from it and transformed into labels. More labels can be added by the LABELS hash. They are added later.

example:

```
my $header = Mail::Message::Head->new;
$msg->head($header);    # set
my $head = $msg->head;  # get
```

$obj->**nrLines**()

Returns the number of lines used for the whole message.

$obj->**sender**()

Returns exactly one address, which is the originator of this message. The returned Mail::Address object is taken from the `Sender` header field, unless that field does not exists, in which case the first address from the `From` field is taken. If none of both provide an address, `undef` is returned.

example: using **sender()** to get exactly one sender address

```
my $sender = $message->sender;
print "Reply to: ", $sender->format, "\n" if defined $sender;
```

$obj->**study**($fieldname)

Study the content of a field, like **get()** does, with as main difference that a Mail::Message::Field::Full object is returned. These objects stringify to an utf8 decoded representation of the data contained in the field, where **get()** does not decode. When the field does not exist, then `undef` is returned. See **Mail::Message::Field::study()**.

example: the **study()** short-cut for header fields

```
print $msg->study('to'), "\n";
```

Is equivalent to:

```
print $msg->head->study('to'), "\n";       # and
print $msg->head->get('to')->study, "\n";
```

or better:
if(my `$to` = `$msg`->study('to')) { print "$to\n" }
if(my `$to` = `$msg`->get('to')) { print `$to`->study, "\n" }

$obj->**subject**()

Returns the message's subject, or the empty string. The subject may have encoded characters in it; use **study()** to get rit of that.

example: using **subject()** to get the message's subject

```
print $msg->subject;
print $msg->study('subject');
```

$obj->**timestamp**()

Get a good timestamp for the message, doesn't matter how much work it is. The value returned is compatible with the platform dependent result of function **time()**.

In these days, the timestamp as supplied by the message (in the `Date` field) is not trustable at all: many spammers produce illegal or unreal dates to influence their location in the displayed folder.

To start, the received headers are tried for a date (see **Mail::Message::Head::Complete::recvstamp()**) and only then the `Date` field. In very rare cases, only with some locally produced messages, no stamp can be found.

`$obj->`**to**()

Returns the addresses which are specified on the `To` header line (or lines). A list of Mail::Address objects is returned. The people addressed here are the targets of the content, and should read it contents carefully.

example: using **to()** to get all primar destination addresses

```
my @to = $message->to;
```

The body

`$obj->`**body**([$body])

Return the body of this message. BE WARNED that this returns you an object which may be encoded: use **decoded()** to get a body with usable data.

With options, a new `$body` is set for this message. This is **not** for normal use unless you understand the consequences: you change the message content without changing the message-ID. The right way to go is via

```
$message = Mail::Message->buildFromBody($body);  # or
$message = Mail::Message->build($body);          # or
$message = $origmsg->forward(body => $body);
```

The `$body` must be an (sub-)class of Mail::Message::Body. In this case, information from the specified body will be copied into the header. The body object will be encoded if needed, because messages written to file or transmitted shall not contain binary data. The converted body is returned.

When `$body` is `undef`, the current message body will be dissected from the message. All relation will be cut. The body is returned, and can be connected to a different message.

example:

```
my $body      = $msg->body;
my @encoded   = $msg->body->lines;

my $new       = Mail::Message::Body->new(mime_type => 'text/html');
my $converted = $msg->body($new);
```

`$obj->`**contentType**()

Returns the content type header line, or `text/plain` if it is not defined. The parameters will be stripped off.

`$obj->`**decoded**(%options)

Decodes the body of this message, and returns it as a body object. Short for `$msg->body->decoded` All `%options` are passed-on.

`$obj->`**encode**(%options)

Encode the message to a certain format. Read the details in the dedicated manual page Mail::Message::Body::Encode. The`%options` which can be specified here are those of the **Mail::Message::Body::encode()** method.

`$obj->`**isMultipart**()

Check whether this message is a multipart message (has attachments). To find this out, we need at least the header of the message; there is no need to read the body of the message to detect this.

`$obj->`**isNested**()

Returns `true` for `message/rfc822` messages and message parts.

`$obj->`**parts**([<'ALL'|'ACTIVE'|'DELETED'|'RECURSE'|$filter>])

Returns the *parts* of this message. Maybe a bit inconvenient: it returns the message itself when it is not a multipart.

Usually, the term *part* is used with *multipart* messages: messages which are encapsulated in the body of a message. To abstract this concept: this method will return you all header-body combinations

which are stored within this message **except** the multipart and message/rfc822 wrappers. Objects returned are `Mail::Message`'s and Mail::Message::Part's.

The option default to 'ALL', which will return the message itself for single-parts, the nested content of a message/rfc822 object, respectively the parts of a multipart without recursion. In case of 'RECURSE', the parts of multiparts will be collected recursively. This option cannot be combined with the other options, which you may want: it that case you have to test yourself.

'ACTIVE' and 'DELETED' check for the deleted flag on messages and message parts. The `$filter` is a code reference, which is called for each part of the message; each part as `RECURSE` would return.

example:

```
my @parts = $msg->parts;           # $msg not multipart: returns ($msg)
my $parts = $msg->parts('ACTIVE'); # returns ($msg)

$msg->delete;
my @parts = $msg->parts;           # returns ($msg)
my $parts = $msg->parts('ACTIVE'); # returns ()
```

Flags

`$obj`->**delete**()

Flag the message to be deleted, which is a shortcut for
`$msg`->label(deleted => time); The real deletion only takes place on a synchronization of the folder. See **deleted()** as well.

The time stamp of the moment of deletion is stored as value, but that is not always preserved in the folder (depends on the implementation). When the same message is deleted more than once, the first time stamp will stay.

example:

```
$message->delete;
$message->deleted(1);  # exactly the same
$message->label(deleted => 1);
delete $message;
```

`$obj`->**deleted**([BOOLEAN])

Set the delete flag for this message. Without argument, the method returns the same as **isDeleted()**, which is preferred. When a true value is given, **delete()** is called.

example:

```
$message->deleted(1);          # delete
$message->delete;              # delete (preferred)

$message->deleted(0);          # undelete

if($message->deleted) {...}    # check
if($message->isDeleted) {...}  # check (preferred)
```

`$obj`->**isDeleted**()

Short-cut for
`$msg`->label('deleted')

For some folder types, you will get the time of deletion in return. This depends on the implementation.

example:

```
next if $message->isDeleted;
```

```
if(my $when = $message->isDeleted) {
   print scalar localtime $when;
}
```

$obj->**isModified**()
Returns whether this message is flagged as being modified. Modifications are changes in header lines, when a new body is set to the message (dangerous), or when labels change.

$obj->**label**($label|PAIRS)
Return the value of the `$label`, optionally after setting some values. In case of setting values, you specify key-value PAIRS.

Labels are used to store knowledge about handling of the message within the folder. Flags about whether a message was read, replied to, or scheduled for deletion.

Some labels are taken from the header's `Status` and `X-Status` lines. Folder types like MH define a separate label file, and Maildir adds letters to the message filename. But the MailBox labels are always the same.

example:

```
print $message->label('seen');
if($message->label('seen')) {...};
$message->label(seen => 1);

$message->label(deleted => 1);  # same as $message->delete
```

$obj->**labels**()
Returns all known labels. In SCALAR context, it returns the knowledge as reference to a hash. This is a reference to the original data, but you shall *not* change that data directly: call `label` for changes!

In LIST context, you get a list of names which are defined. Be warned that they will not all evaluate to true, although most of them will.

$obj->**labelsToStatus**()
When the labels were changed, that may effect the `Status` and/or `X-Status` header lines of mbox messages. Read about the relation between these fields and the labels in the DETAILS chapter.

The method will carefully only affect the result of **modified()** when there is a real change of flags, so not for each call to **label()**.

$obj->**modified**([BOOLEAN])
Returns (optionally after setting) whether this message is flagged as being modified. See **isModified()**.

$obj->**statusToLabels**()
Update the labels according the status lines in the header. See the description in the DETAILS chapter.

The whole message as text

$obj->**file**()
Inherited, see ‘‘The whole message as text’’ in Mail::Message::Construct::Text

$obj->**lines**()
Inherited, see ‘‘The whole message as text’’ in Mail::Message::Construct::Text

$obj->**printStructure**([$fh|undef],[$indent])
Inherited, see ‘‘The whole message as text’’ in Mail::Message::Construct::Text

$obj->**string**()
Inherited, see ‘‘The whole message as text’’ in Mail::Message::Construct::Text

Internals

`$obj->`**clonedFrom**()
Returns the `$message` which is the source of this message, which was created by a **clone()** operation.

Mail::Message->**coerce**($message, `%options`)
Coerce a `$message` into a Mail::Message. In some occasions, for instance where you add a message to a folder, this coercion is automatically called to ensure that the correct message type is stored.

The coerced message is returned on success, otherwise `undef`. The coerced message may be a reblessed version of the original message or a new object. In case the message has to be specialized, for instance from a general Mail::Message into a Mail::Box::Mbox::Message, no copy is needed. However, to coerce a Mail::Internet object into a Mail::Message, a lot of copying and converting will take place.

Valid MESSAGEs which can be coerced into Mail::Message objects are of type

- Any type of Mail::Box::Message
- MIME::Entity objects, using Mail::Message::Convert::MimeEntity
- Mail::Internet objects, using Mail::Message::Convert::MailInternet
- Email::Simple objects, using Mail::Message::Convert::EmailSimple
- Email::Abstract objects

Mail::Message::Part's, which are extensions of `Mail::Message`'s, can also be coerced directly from a Mail::Message::Body.

example:

```
my $folder  = Mail::Box::Mbox->new;
my $message = Mail::Message->build(...);

my $coerced = Mail::Box::Mbox::Message->coerce($message);
$folder->addMessage($coerced);
```

Simpler replacement for the previous two lines:

```
my $coerced = $folder->addMessage($message);
```

`$obj->`**isDelayed**()
Check whether the message is delayed (not yet read from file). Returns true or false, dependent on the body type.

`$obj->`**readBody**(`$parser`, `$head`, [$bodytype])
Read a body of a message. The `$parser` is the access to the folder's file, and the `$head` is already read. Information from the`$head` is used to create e xpectations about the message's length, but also to determine the mime-type and encodings of the body data.

The `$bodytype` determines which kind of body will be made and defaults to the value specified by new(body_type). `$bodytype` may be the name of a body class, or a reference to a routine which returns the body's class when passed the `$head` as only argument.

`$obj->`**readFromParser**(`$parser`, [$bodytype])
Read one message from file. The `$parser` is opened on the file. First **readHead()** is called, and the head is stored in the message. Then **readBody()** is called, to produce a body. Also the body is added to the message without decodings being done.

The optional `$bodytype` may be a body class or a reference to a code which returns a body-class based on the header.

`$obj->`**readHead**(`$parser`, [$class])
Read a head into an object of the specified `$class`. The`$class` def aults to new(head_type). The `$parser` is the access to the folder's file.

`$obj->`**recursiveRebuildPart**($part, `%options`)
Inherited, see ''Internals'' in Mail::Message::Construct::Rebuild

`$obj->`**storeBody**($body)
Where the **body()** method can be used to set and get a body, with all the necessary checks, this method is bluntly adding the specified body to the message. No conversions, not checking.

`$obj->`**takeMessageId**([STRING])
Take the message-id from the STRING, or create one when the `undef` is specified. If not STRING nor `undef` is given, the current header of the message is requested for the value of the `'Message-ID'` field.

Angles (if present) are removed from the id.

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

`$obj->`**AUTOLOAD**()
Inherited, see ''METHODS'' in Mail::Message::Construct

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

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

`$obj->`**logPriority**($level)
Mail::Message->**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

`$obj->`**shortSize**([$value])
Mail::Message->**shortSize**([$value])
Represent an integer `$value` representing the size of file or memory, (which can be large) into a short string using M and K (Megabytes and Kilobytes). Without `$value`, the size of the message head is used.

`$obj->`**shortString**()
Convert the message header to a short string (without trailing newline), representing the most important facts (for debugging purposes only). For now, it only reports size and subject.

`$obj->`**trace**([$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

`$obj->`**destruct**()
Remove the information contained in the message object. This will be ignored when more than one reference to the same message object exists, because the method has the same effect as assigning `undef` to the variable which contains the reference. Normal garbage collection will call `DESTROY()` when possible.

This method is only provided to hide differences with messages which are located in folders: their **Mail::Box::Message::destruct()** works quite differently.

example: of Mail::Message destruct

```
my $msg = Mail::Message->read;
$msg->destruct;
$msg = undef;    # same
```

DETAILS

Structure of a Message

A MIME-compliant message is build upon two parts: the *header* and the *body*.

The header

The header is a list of fields, some spanning more than one line (*folded*) each telling something about the message. Information stored in here are for instance the sender of the message, the receivers of the message, when it was transported, how it was transported, etc. Headers can grow quite large.

In MailBox, each message object manages exactly one header object (a Mail::Message::Head) and one body object (a Mail::Message::Body). The header contains a list of header fields, which are represented by Mail::Message::Field objects.

The body

The body contains the ‘‘payload’’: the data to be transferred. The data can be encoded, only accessible with a specific application, and may use some weird character-set, like Vietnamese; the MailBox distribution tries to assist you with handling these e-mails without the need to know all the details. This additional information (‘‘meta-information’’) about the body data is stored in the header. The header contains more information, for instance about the message transport and relations to other messages.

Message object implementation

The general idea about the structure of a message is

```
Mail::Message
 |  |
 |  `-has-one--Mail::Message::Body
 |
 `----has-one--Mail::Message::Head
                  |
                  `-has-many--Mail::Message::Field
```

However: there are about 7 kinds of body objects, 3 kinds of headers and 3 kinds of fields. You will usually not see too much of these kinds, because they are merely created for performance reasons and can be used all the same, with the exception of the multipart bodies.

A multipart body is either a Mail::Message::Body::Multipart (mime type `multipart/*`) or a Mail::Message::Body::Nested (mime type `message/rfc822`). These bodies are more complex:

```
Mail::Message::Body::Multipart
 |
 `-has-many--Mail::Message::Part
              |  |
              |  `-has-one--Mail::Message::Body
              |
              `----has-one--Mail::Message::Head
```

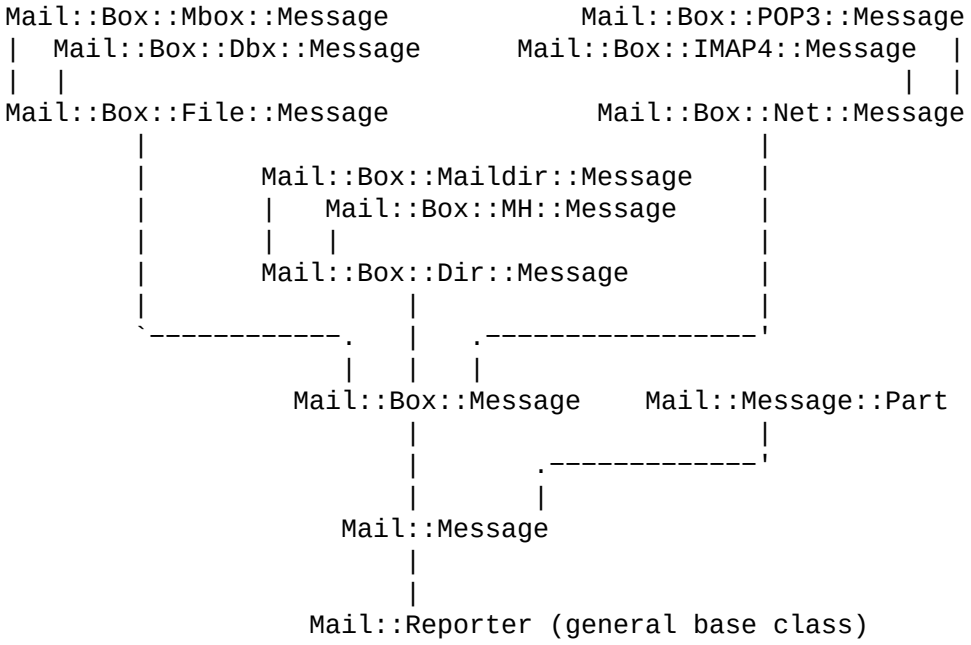
Before you try to reconstruct multiparts or nested messages yourself, you can better take a look at Mail::Message::Construct::Rebuild.

Message class implementation

The class structure of messages is very close to that of folders. For instance, a Mail::Box::File::Message relates to a Mail::Box::File folder.

As extra level of inheritance, it has a Mail::Message, which is a message without location. And there is a special case of message: Mail::Message::Part is a message encapsulated in a multipart body.

The message types are:



By far most folder features are implemented in Mail::Box, so available to all folder types. Sometimes, features which appear in only some of the folder types are simulated for folders that miss them, like sub-folder support for MBOX.

Two strange other message types are defined: the Mail::Message::Dummy, which fills holes in Mail::Box::Thread::Node lists, and a Mail::Box::Message::Destructed, this is an on purpose demolished message to reduce memory consumption.

Labels

Labels (also named ‘‘Flags’’) are used to indicate some special condition on the message, primary targeted on organizational issues: which messages are already read or should be deleted. There is a very strong user relation to labels.

The main complication is that each folder type has its own way of storing labels. To give an indication: MBOX folders use `Status` and `X-Status` header fields, MH uses a `.mh-sequences` file, MAILDIR encodes the flags in the message’s filename, and IMAP has flags as part of the protocol.

Besides, some folder types can store labels with user defined names, where other lack that feature. Some folders have case-insensitive labels, other don’t. Read all about the specifics in the manual page of the

message type you actually have.

Predefined labels

To standardize the folder types, MailBox has defined the following labels, which can be used with the **label()** and **labels()** methods on all kinds of messages:

- deleted

  This message is flagged to be deleted once the folder closes. Be very careful about the concept of 'delete' in a folder context : it is only a flag, and does not involve immediate action! This means, for instance, that the memory which is used by Perl to store the message is not released immediately (see **destruct()** if you need to).

  The methods **delete()**, **deleted()**, and **isDeleted()** are only short-cuts for managing the `delete` label (as of MailBox 2.052).

- draft

  The user has prepared this message, but is has not been send (yet). This flag is not automatically added to a message by MailBox, and has only a meaning in user applications.

- flagged

  Messages can be *flagged* for some purpose, for instance as result of a search for spam in a folder. The **Mail::Box::messages()** method can be used to collect all these flagged messages from the folder.

  Probably it is more useful to use an understandable name (like `spam`) for these selections, however these self-defined labels can not stored in all folder types.

- old

  The message was already in the folder when it was opened the last time, so was not recently added to the folder. This flag will never automatically be set by MailBox, because it would probably conflict with the user's idea of what is old.

- passed

  Not often used or kept, this flag indicates that the message was bounced or forwarded to someone else.

- replied

  The user (or application) has sent a message back to the sender of the message, as response of this one. This flag is automatically set if you use **reply()**, but not with **forward()** or **bounce()**.

- seen

  When this flag is set, the receiver of the message has consumed the message. A mail user agent (MUA) will set this flag when the user has opened the message once.

Status and X-Status fields

Mbox folders have no special means of storing information about messages (except the message separator line), and therefore have to revert to adding fields to the message header when something special comes up. This feature is also enabled for POP3, although whether that works depends on the POP server.

All applications which can handle mbox folders support the `Status` and `X-Status` field convensions. The following encoding is used:

```
Flag   Field       Label
R      Status   => seen    (Read)
O      Status   => old     (not recent)
A      X-Status => replied (Answered)
F      X-Status => flagged
```

There is no special flag for `deleted`, which most other folders support: messages flagged to be deleted will never be written to a folder file when it is closed.

DIAGNOSTICS

Error: Cannot coerce a `$class` object into a `$class` object

Error: Cannot include forward source as `$include`.

Unknown alternative for the forward(include). Valid choices are `NO`, `INLINE`, `ATTACH`, and `ENCAPSULATE`.

Error: Cannot include reply source as `$include`.

Unknown alternative for the `include` option of **reply()**. Valid choices are `NO`, `INLINE`, and `ATTACH`.

Error: Method bounce requires To, Cc, or Bcc

The message **bounce()** method forwards a received message off to someone else without modification; you must specified it's new destination. If you have the urge not to specify any destination, you probably are looking for **reply()**. When you wish to modify the content, use **forward()**.

Error: Method forwardAttach requires a preamble

Error: Method forwardEncapsulate requires a preamble

Error: No address to create forwarded to.

If a forward message is created, a destination address must be specified.

Error: No default mailer found to send message.

The message **send()** mechanism had not enough information to automatically find a mail transfer agent to sent this message. Specify a mailer explicitly using the `via` options.

Error: No rebuild rule `$name` defined.

Error: Only **build()** Mail::Message's; they are not in a folder yet

You may wish to construct a message to be stored in a some kind of folder, but you need to do that in two steps. First, create a normal Mail::Message, and then add it to the folder. During this **Mail::Box::addMessage()** process, the message will get **coerce()**−d into the right message type, adding storage information and the like.

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.

Error: coercion starts with some object

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