NAME

JSON − JSON (JavaScript Object Notation) encoder/decoder

SYNOPSIS

```
use JSON; # imports encode_json, decode_json, to_json and from_json.

# simple and fast interfaces (expect/generate UTF-8)

$utf8_encoded_json_text = encode_json $perl_hash_or_arrayref;
$perl_hash_or_arrayref  = decode_json $utf8_encoded_json_text;

# OO-interface

$json = JSON->new->allow_nonref;

$json_text   = $json->encode( $perl_scalar );
$perl_scalar = $json->decode( $json_text );

$pretty_printed = $json->pretty->encode( $perl_scalar ); # pretty-printing
```

DESCRIPTION

This module is a thin wrapper for JSON::XS−compatible modules with a few additional features. All the backend modules convert a Perl data structure to a JSON text and vice versa. This module uses JSON::XS by default, and when JSON::XS is not available, falls back on JSON::PP, which is in the Perl core since 5.14. If JSON::PP is not available either, this module then falls back on JSON::backportPP (which is actually JSON::PP in a different .pm file) bundled in the same distribution as this module. You can also explicitly specify to use Cpanel::JSON::XS, a fork of JSON::XS by Reini Urban.

All these backend modules have slight incompatibilities between them, including extra features that other modules don't support, but as long as you use only common features (most important ones are described below), migration from backend to backend should be reasonably easy. For details, see each backend module you use.

CHOOSING BACKEND

This module respects an environmental variable called `PERL_JSON_BACKEND` when it decides a backend module to use. If this environmental variable is not set, it tries to load JSON::XS, and if JSON::XS is not available, it falls back on JSON::PP, and then JSON::backportPP if JSON::PP is not available either.

If you always don't want it to fall back on pure perl modules, set the variable like this (`export` may be `setenv`, `set` and the likes, depending on your environment):

```
> export PERL_JSON_BACKEND=JSON::XS
```

If you prefer Cpanel::JSON::XS to JSON::XS, then:

```
> export PERL_JSON_BACKEND=Cpanel::JSON::XS,JSON::XS,JSON::PP
```

You may also want to set this variable at the top of your test files, in order not to be bothered with incompatibilities between backends (you need to wrap this in `BEGIN`, and set before actually `use`−ing JSON module, as it decides its backend as soon as it's loaded):

```
BEGIN { $ENV{PERL_JSON_BACKEND}='JSON::backportPP'; }
use JSON;
```

USING OPTIONAL FEATURES

There are a few options you can set when you `use` this module. These historical options are only kept for backward compatibility, and should not be used in a new application.

−support_by_pp

```
BEGIN { $ENV{PERL_JSON_BACKEND} = 'JSON::XS' }

use JSON −support_by_pp;
```

```
my $json = JSON->new;
# escape_slash is for JSON::PP only.
$json->allow_nonref->escape_slash->encode("/");
```

With this option, this module loads its pure perl backend along with its XS backend (if available), and lets the XS backend to watch if you set a flag only JSON::PP supports. When you do, the internal JSON::XS object is replaced with a newly created JSON::PP object with the setting copied from the XS object, so that you can use JSON::PP flags (and its slower `decode`/`encode` methods) from then on. In other words, this is not something that allows you to hook JSON::XS to change its behavior while keeping its speed. JSON::XS and JSON::PP objects are quite different (JSON::XS object is a blessed scalar reference, while JSON::PP object is a blessed hash reference), and can't share their internals.

To avoid needless overhead (by copying settings), you are advised not to use this option and just to use JSON::PP explicitly when you need JSON::PP features.

−convert_blessed_universally

```
use JSON -convert_blessed_universally;

my $json = JSON->new->allow_nonref->convert_blessed;
my $object = bless {foo => 'bar'}, 'Foo';
$json->encode($object); # => {"foo":"bar"}
```

JSON::XS−compatible backend modules don't encode blessed objects by default (except for their boolean values, which are typically blessed JSON::PP::Boolean objects). If you need to encode a data structure that may contain objects, you usually need to look into the structure and replace objects with alternative non-blessed values, or enable `convert_blessed` and provide a `TO_JSON` method for each object's (base) class that may be found in the structure, in order to let the methods replace the objects with whatever scalar values the methods return.

If you need to serialise data structures that may contain arbitrary objects, it's probably better to use other serialisers (such as Sereal or Storable for example), but if you do want to use this module for that purpose, `-convert_blessed_universally` option may help, which tweaks `encode` method of the backend to install `UNIVERSAL::TO_JSON` method (locally) before encoding, so that all the objects that don't have their own `TO_JSON` method can fall back on the method in the `UNIVERSAL` namespace. Note that you still need to enable `convert_blessed` flag to actually encode objects in a data structure, and `UNIVERSAL::TO_JSON` method installed by this option only converts blessed hash/array references into their unblessed clone (including private keys/values that are not supposed to be exposed). Other blessed references will be converted into null.

This feature is experimental and may be removed in the future.

−no_export

When you don't want to import functional interfaces from a module, you usually supply `()` to its `use` statement.

```
use JSON (); # no functional interfaces
```

If you don't want to import functional interfaces, but you also want to use any of the above options, add `-no_export` to the option list.

```
# no functional interfaces, while JSON::PP support is enabled.
use JSON -support_by_pp, -no_export;
```

FUNCTIONAL INTERFACE

This section is taken from JSON::XS. `encode_json` and `decode_json` are exported by default.

This module also exports `to_json` and `from_json` for backward compatibility. These are slower, and may expect/generate different stuff from what `encode_json` and `decode_json` do, depending on their options. It's better just to use Object-Oriented interfaces than using these two functions.

encode_json

```
$json_text = encode_json $perl_scalar
```

Converts the given Perl data structure to a UTF-8 encoded, binary string (that is, the string contains octets only). Croaks on error.

This function call is functionally identical to:

```
$json_text = JSON->new->utf8->encode($perl_scalar)
```

Except being faster.

decode_json

```
$perl_scalar = decode_json $json_text
```

The opposite of `encode_json`: expects an UTF-8 (binary) string and tries to parse that as an UTF-8 encoded JSON text, returning the resulting reference. Croaks on error.

This function call is functionally identical to:

```
$perl_scalar = JSON->new->utf8->decode($json_text)
```

Except being faster.

to_json

```
$json_text = to_json($perl_scalar[, $optional_hashref])
```

Converts the given Perl data structure to a Unicode string by default. Croaks on error.

Basically, this function call is functionally identical to:

```
$json_text = JSON->new->encode($perl_scalar)
```

Except being slower.

You can pass an optional hash reference to modify its behavior, but that may change what `to_json` expects/generates (see `ENCODING/CODESET FLAG NOTES` for details).

```
$json_text = to_json($perl_scalar, {utf8 => 1, pretty => 1})
# => JSON->new->utf8(1)->pretty(1)->encode($perl_scalar)
```

from_json

```
$perl_scalar = from_json($json_text[, $optional_hashref])
```

The opposite of `to_json`: expects a Unicode string and tries to parse it, returning the resulting reference. Croaks on error.

Basically, this function call is functionally identical to:

```
$perl_scalar = JSON->new->decode($json_text)
```

You can pass an optional hash reference to modify its behavior, but that may change what `from_json` expects/generates (see `ENCODING/CODESET FLAG NOTES` for details).

```
$perl_scalar = from_json($json_text, {utf8 => 1})
# => JSON->new->utf8(1)->decode($json_text)
```

JSON::is_bool

```
$is_boolean = JSON::is_bool($scalar)
```

Returns true if the passed scalar represents either JSON::true or JSON::false, two constants that act like `1` and `0` respectively and are also used to represent JSON `true` and `false` in Perl strings.

See MAPPING, below, for more information on how JSON values are mapped to Perl.

COMMON OBJECT-ORIENTED INTERFACE

This section is also taken from JSON::XS.

The object oriented interface lets you configure your own encoding or decoding style, within the limits of supported formats.

new

```
$json = JSON->new
```

Creates a new JSON::XS−compatible backend object that can be used to de/encode JSON strings. All boolean flags described below are by default *disabled* (with the exception of `allow_nonref`, which defaults to *enabled* since version `4.0`).

The mutators for flags all return the backend object again and thus calls can be chained:

```
my $json = JSON->new->utf8->space_after->encode({a => [1,2]})
=> {"a": [1, 2]}
```

ascii

```
$json = $json->ascii([$enable])

$enabled = $json->get_ascii
```

If `$enable` is true (or missing), then the `encode` method will not generate characters outside the code range `0..127` (which is ASCII). Any Unicode characters outside that range will be escaped using either a single \uXXXX (BMP characters) or a double \uHHHH\uLLLLL escape sequence, as per RFC4627. The resulting encoded JSON text can be treated as a native Unicode string, an ascii-encoded, latin1−encoded or UTF−8 encoded string, or any other superset of ASCII.

If `$enable` is false, then the `encode` method will not escape Unicode characters unless required by the JSON syntax or other flags. This results in a faster and more compact format.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

The main use for this flag is to produce JSON texts that can be transmitted over a 7−bit channel, as the encoded JSON texts will not contain any 8 bit characters.

```
JSON->new->ascii(1)->encode([chr 0x10401])
=> ["\ud801\udc01"]
```

latin1

```
$json = $json->latin1([$enable])

$enabled = $json->get_latin1
```

If `$enable` is true (or missing), then the `encode` method will encode the resulting JSON text as latin1 (or iso−8859−1), escaping any characters outside the code range `0..255`. The resulting string can be treated as a latin1−encoded JSON text or a native Unicode string. The `decode` method will not be affected in any way by this flag, as `decode` by default expects Unicode, which is a strict superset of latin1.

If `$enable` is false, then the `encode` method will not escape Unicode characters unless required by the JSON syntax or other flags.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

The main use for this flag is efficiently encoding binary data as JSON text, as most octets will not be escaped, resulting in a smaller encoded size. The disadvantage is that the resulting JSON text is encoded in latin1 (and must correctly be treated as such when storing and transferring), a rare encoding for JSON. It is therefore most useful when you want to store data structures known to contain binary data efficiently in files or databases, not when talking to other JSON encoders/decoders.

```
JSON->new->latin1->encode (["\x{89}\x{abc}"]
=> ["\x{89}\\u0abc"]    # (perl syntax, U+abc escaped, U+89 not)
```

utf8

```
$json = $json->utf8([$enable])

$enabled = $json->get_utf8
```

If `$enable` is true (or missing), then the `encode` method will encode the JSON result into UTF−8, as required by many protocols, while the `decode` method expects to be handled an UTF−8−encoded string.

Please note that UTF−8−encoded strings do not contain any characters outside the range `0..255`, they are thus useful for bytewise/binary I/O. In future versions, enabling this option might enable autodetection of the UTF−16 and UTF−32 encoding families, as described in RFC4627.

If `$enable` is false, then the `encode` method will return the JSON string as a (non-encoded) Unicode string, while `decode` expects thus a Unicode string. Any decoding or encoding (e.g. to UTF−8 or UTF−16) needs to be done yourself, e.g. using the Encode module.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

Example, output UTF−16BE−encoded JSON:

```
use Encode;
$jsontext = encode "UTF-16BE", JSON->new->encode ($object);
```

Example, decode UTF−32LE−encoded JSON:

```
use Encode;
$object = JSON->new->decode (decode "UTF-32LE", $jsontext);
```

pretty

```
$json = $json->pretty([$enable])
```

This enables (or disables) all of the `indent`, `space_before` and `space_after` (and in the future possibly more) flags in one call to generate the most readable (or most compact) form possible.

indent

```
$json = $json->indent([$enable])

$enabled = $json->get_indent
```

If `$enable` is true (or missing), then the `encode` method will use a multiline format as output, putting every array member or object/hash key-value pair into its own line, indenting them properly.

If `$enable` is false, no newlines or indenting will be produced, and the resulting JSON text is guaranteed not to contain any `newlines`.

This setting has no effect when decoding JSON texts.

space_before

```
$json = $json->space_before([$enable])

$enabled = $json->get_space_before
```

If `$enable` is true (or missing), then the `encode` method will add an extra optional space before the `:` separating keys from values in JSON objects.

If `$enable` is false, then the `encode` method will not add any extra space at those places.

This setting has no effect when decoding JSON texts. You will also most likely combine this setting with `space_after`.

Example, space_before enabled, space_after and indent disabled:

```
{"key" :"value"}
```

space_after

```
$json = $json->space_after([$enable])

$enabled = $json->get_space_after
```

If `$enable` is true (or missing), then the `encode` method will add an extra optional space after the `:` separating keys from values in JSON objects and extra whitespace after the `,` separating key-value pairs and array members.

If `$enable` is false, then the `encode` method will not add any extra space at those places.

This setting has no effect when decoding JSON texts.

Example, space_before and indent disabled, space_after enabled:

```
{"key": "value"}
```

relaxed

```
$json = $json->relaxed([$enable])

$enabled = $json->get_relaxed
```

If `$enable` is true (or missing), then `decode` will accept some extensions to normal JSON syntax (see below). `encode` will not be affected in any way. *Be aware that this option makes you accept invalid JSON texts as if they were valid!*. I suggest only to use this option to parse application-specific files written by humans (configuration files, resource files etc.)

If `$enable` is false (the default), then `decode` will only accept valid JSON texts.

Currently accepted extensions are:

- list items can have an end-comma

  JSON *separates* array elements and key-value pairs with commas. This can be annoying if you write JSON texts manually and want to be able to quickly append elements, so this extension accepts comma at the end of such items not just between them:

  ```
  [
     1,
     2, <- this comma not normally allowed
  ]
  {
     "k1": "v1",
     "k2": "v2", <- this comma not normally allowed
  }
  ```

- shell-style '#'−comments

  Whenever JSON allows whitespace, shell-style comments are additionally allowed. They are terminated by the first carriage-return or line-feed character, after which more white-space and comments are allowed.

  ```
  [
     1, # this comment not allowed in JSON
        # neither this one...
  ]
  ```

canonical

```
$json = $json->canonical([$enable])

$enabled = $json->get_canonical
```

If `$enable` is true (or missing), then the `encode` method will output JSON objects by sorting their keys. This is adding a comparatively high overhead.

If `$enable` is false, then the `encode` method will output key-value pairs in the order Perl stores them (which will likely change between runs of the same script, and can change even within the same run from 5.18 onwards).

This option is useful if you want the same data structure to be encoded as the same JSON text (given the same overall settings). If it is disabled, the same hash might be encoded differently even if contains the same data, as key-value pairs have no inherent ordering in Perl.

This setting has no effect when decoding JSON texts.

This setting has currently no effect on tied hashes.

allow_nonref

```
$json = $json->allow_nonref([$enable])

$enabled = $json->get_allow_nonref
```

Unlike other boolean options, this option is enabled by default beginning with version `4.0`.

If `$enable` is true (or missing), then the `encode` method can convert a non-reference into its corresponding string, number or null JSON value, which is an extension to RFC4627. Likewise, `decode` will accept those JSON values instead of croaking.

If `$enable` is false, then the `encode` method will croak if it isn't passed an arrayref or hashref, as JSON texts must either be an object or array. Likewise, `decode` will croak if given something that is not a JSON object or array.

Example, encode a Perl scalar as JSON value with enabled `allow_nonref`, resulting in an invalid JSON text:

```
JSON->new->allow_nonref->encode ("Hello, World!")
=> "Hello, World!"
```

allow_unknown

```
$json = $json->allow_unknown ([$enable])

$enabled = $json->get_allow_unknown
```

If `$enable` is true (or missing), then `encode` will *not* throw an exception when it encounters values it cannot represent in JSON (for example, filehandles) but instead will encode a JSON `null` value. Note that blessed objects are not included here and are handled separately by c<allow_blessed>.

If `$enable` is false (the default), then `encode` will throw an exception when it encounters anything it cannot encode as JSON.

This option does not affect `decode` in any way, and it is recommended to leave it off unless you know your communications partner.

allow_blessed

```
$json = $json->allow_blessed([$enable])

$enabled = $json->get_allow_blessed
```

See "OBJECT SERIALISATION" for details.

If `$enable` is true (or missing), then the `encode` method will not barf when it encounters a blessed reference that it cannot convert otherwise. Instead, a JSON `null` value is encoded instead of the object.

If `$enable` is false (the default), then `encode` will throw an exception when it encounters a blessed object that it cannot convert otherwise.

This setting has no effect on `decode`.

convert_blessed

```
$json = $json->convert_blessed([$enable])

$enabled = $json->get_convert_blessed
```

See "OBJECT SERIALISATION" for details.

If `$enable` is true (or missing), then `encode`, upon encountering a blessed object, will check for the availability of the `TO_JSON` method on the object's class. If found, it will be called in scalar context and the resulting scalar will be encoded instead of the object.

The `TO_JSON` method may safely call die if it wants. If `TO_JSON` returns other blessed objects, those will be handled in the same way. `TO_JSON` must take care of not causing an endless recursion cycle (== crash) in this case. The name of `TO_JSON` was chosen because other methods called by the Perl core (== not by

the user of the object) are usually in upper case letters and to avoid collisions with any `to_json` function or method.

If `$enable` is false (the default), then `encode` will not consider this type of conversion.

This setting has no effect on `decode`.

allow_tags (since version 3.0)

```
$json = $json->allow_tags([$enable])

$enabled = $json->get_allow_tags
```

See "OBJECT SERIALISATION" for details.

If `$enable` is true (or missing), then `encode`, upon encountering a blessed object, will check for the availability of the `FREEZE` method on the object's class. If found, it will be used to serialise the object into a nonstandard tagged JSON value (that JSON decoders cannot decode).

It also causes `decode` to parse such tagged JSON values and deserialise them via a call to the `THAW` method.

If `$enable` is false (the default), then `encode` will not consider this type of conversion, and tagged JSON values will cause a parse error in `decode`, as if tags were not part of the grammar.

boolean_values (since version 4.0)

```
$json->boolean_values([$false, $true])

($false,  $true) = $json->get_boolean_values
```

By default, JSON booleans will be decoded as overloaded `$JSON::false` and `$JSON::true` objects.

With this method you can specify your own boolean values for decoding – on decode, JSON `false` will be decoded as a copy of `$false`, and JSON `true` will be decoded as `$true` ("copy" here is the same thing as assigning a value to another variable, i.e. `$copy = $false`).

This is useful when you want to pass a decoded data structure directly to other serialisers like YAML, Data::MessagePack and so on.

Note that this works only when you `decode`. You can set incompatible boolean objects (like boolean), but when you `encode` a data structure with such boolean objects, you still need to enable `convert_blessed` (and add a `TO_JSON` method if necessary).

Calling this method without any arguments will reset the booleans to their default values.

`get_boolean_values` will return both `$false` and `$true` values, or the empty list when they are set to the default.

filter_json_object

```
$json = $json->filter_json_object([$coderef])
```

When `$coderef` is specified, it will be called from `decode` each time it decodes a JSON object. The only argument is a reference to the newly-created hash. If the code references returns a single scalar (which need not be a reference), this value (or rather a copy of it) is inserted into the deserialised data structure. If it returns an empty list (NOTE: *not* `undef`, which is a valid scalar), the original deserialised hash will be inserted. This setting can slow down decoding considerably.

When `$coderef` is omitted or undefined, any existing callback will be removed and `decode` will not change the deserialised hash in any way.

Example, convert all JSON objects into the integer 5:

```
my $js = JSON->new->filter_json_object(sub { 5 });
# returns [5]
$js->decode('[{}]');
# returns 5
$js->decode('{"a":1, "b":2}');
```

filter_json_single_key_object

```
$json = $json->filter_json_single_key_object($key [=> $coderef])
```

Works remotely similar to `filter_json_object`, but is only called for JSON objects having a single key named `$key`.

This `$coderef` is called before the one specified via `filter_json_object`, if any. It gets passed the single value in the JSON object. If it returns a single value, it will be inserted into the data structure. If it returns nothing (not even `undef` but the empty list), the callback from `filter_json_object` will be called next, as if no single-key callback were specified.

If `$coderef` is omitted or undefined, the corresponding callback will be disabled. There can only ever be one callback for a given key.

As this callback gets called less often then the `filter_json_object` one, decoding speed will not usually suffer as much. Therefore, single-key objects make excellent targets to serialise Perl objects into, especially as single-key JSON objects are as close to the type-tagged value concept as JSON gets (it's basically an ID/VALUE tuple). Of course, JSON does not support this in any way, so you need to make sure your data never looks like a serialised Perl hash.

Typical names for the single object key are `__class_whatever__`, or `$__dollars_are_rarely_used__$` or `}ugly_brace_placement`, or even things like `__class_md5sum(classname)__`, to reduce the risk of clashing with real hashes.

Example, decode JSON objects of the form `{ "__widget__" => <id> }` into the corresponding `$WIDGET{<id>}` object:

```
# return whatever is in $WIDGET{5}:
JSON
   ->new
   ->filter_json_single_key_object (__widget__ => sub {
         $WIDGET{ $_[0] }
      })
   ->decode ('{"__widget__": 5')

# this can be used with a TO_JSON method in some "widget" class
# for serialisation to json:
sub WidgetBase::TO_JSON {
   my ($self) = @_;

   unless ($self->{id}) {
      $self->{id} = ..get..some..id..;
      $WIDGET{$self->{id}} = $self;
   }

   { __widget__ => $self->{id} }
}
```

max_depth

```
$json = $json->max_depth([$maximum_nesting_depth])

$max_depth = $json->get_max_depth
```

Sets the maximum nesting level (default `512`) accepted while encoding or decoding. If a higher nesting

level is detected in JSON text or a Perl data structure, then the encoder and decoder will stop and croak at that point.

Nesting level is defined by number of hash− or arrayrefs that the encoder needs to traverse to reach a given point or the number of `{` or `[` characters without their matching closing parenthesis crossed to reach a given character in a string.

Setting the maximum depth to one disallows any nesting, so that ensures that the object is only a single hash/object or array.

If no argument is given, the highest possible setting will be used, which is rarely useful.

See ''SECURITY CONSIDERATIONS'' in JSON::XS for more info on why this is useful.

max_size

```
$json = $json->max_size([$maximum_string_size])

$max_size = $json->get_max_size
```

Set the maximum length a JSON text may have (in bytes) where decoding is being attempted. The default is `0`, meaning no limit. When `decode` is called on a string that is longer then this many bytes, it will not attempt to decode the string but throw an exception. This setting has no effect on `encode` (yet).

If no argument is given, the limit check will be deactivated (same as when `0` is specified).

See ''SECURITY CONSIDERATIONS'' in JSON::XS for more info on why this is useful.

encode

```
$json_text = $json->encode($perl_scalar)
```

Converts the given Perl value or data structure to its JSON representation. Croaks on error.

decode

```
$perl_scalar = $json->decode($json_text)
```

The opposite of `encode`: expects a JSON text and tries to parse it, returning the resulting simple scalar or reference. Croaks on error.

decode_prefix

```
($perl_scalar, $characters) = $json->decode_prefix($json_text)
```

This works like the `decode` method, but instead of raising an exception when there is trailing garbage after the first JSON object, it will silently stop parsing there and return the number of characters consumed so far.

This is useful if your JSON texts are not delimited by an outer protocol and you need to know where the JSON text ends.

```
JSON->new->decode_prefix ("[1] the tail")
=> ([1], 3)
```

ADDITIONAL METHODS

The following methods are for this module only.

backend

```
$backend = $json->backend
```

Since 2.92, `backend` method returns an abstract backend module used currently, which should be JSON::Backend::XS (which inherits JSON::XS or Cpanel::JSON::XS), or JSON::Backend::PP (which inherits JSON::PP), not to monkey-patch the actual backend module globally.

If you need to know what is used actually, use `isa`, instead of string comparison.

is_xs

```
$boolean = $json->is_xs
```

Returns true if the backend inherits JSON::XS or Cpanel::JSON::XS.

is_pp

```
$boolean = $json->is_pp
```

Returns true if the backend inherits JSON::PP.

property

```
$settings = $json->property()
```

Returns a reference to a hash that holds all the common flag settings.

```
$json = $json->property('utf8' => 1)
$value = $json->property('utf8') # 1
```

You can use this to get/set a value of a particular flag.

boolean

```
$boolean_object = JSON->boolean($scalar)
```

Returns `$JSON::true` if `$scalar` contains a true value, `$JSON::false` otherwise. You can use this as a full-qualified function (`JSON::boolean($scalar)`).

INCREMENTAL PARSING

This section is also taken from JSON::XS.

In some cases, there is the need for incremental parsing of JSON texts. While this module always has to keep both JSON text and resulting Perl data structure in memory at one time, it does allow you to parse a JSON stream incrementally. It does so by accumulating text until it has a full JSON object, which it then can decode. This process is similar to using `decode_prefix` to see if a full JSON object is available, but is much more efficient (and can be implemented with a minimum of method calls).

This module will only attempt to parse the JSON text once it is sure it has enough text to get a decisive result, using a very simple but truly incremental parser. This means that it sometimes won't stop as early as the full parser, for example, it doesn't detect mismatched parentheses. The only thing it guarantees is that it starts decoding as soon as a syntactically valid JSON text has been seen. This means you need to set resource limits (e.g. `max_size`) to ensure the parser will stop parsing in the presence if syntax errors.

The following methods implement this incremental parser.

incr_parse

```
$json->incr_parse( [$string] ) # void context

$obj_or_undef = $json->incr_parse( [$string] ) # scalar context

@obj_or_empty = $json->incr_parse( [$string] ) # list context
```

This is the central parsing function. It can both append new text and extract objects from the stream accumulated so far (both of these functions are optional).

If `$string` is given, then this string is appended to the already existing JSON fragment stored in the `$json` object.

After that, if the function is called in void context, it will simply return without doing anything further. This can be used to add more text in as many chunks as you want.

If the method is called in scalar context, then it will try to extract exactly *one* JSON object. If that is successful, it will return this object, otherwise it will return `undef`. If there is a parse error, this method will croak just as `decode` would do (one can then use `incr_skip` to skip the erroneous part). This is the most common way of using the method.

And finally, in list context, it will try to extract as many objects from the stream as it can find and return them, or the empty list otherwise. For this to work, there must be no separators (other than whitespace) between the JSON objects or arrays, instead they must be concatenated back-to-back. If an error occurs, an exception will be raised as in the scalar context case. Note that in this case, any previously-parsed JSON texts will be lost.

Example: Parse some JSON arrays/objects in a given string and return them.

```
my @objs = JSON->new->incr_parse ("[5][7][1,2]");
```

incr_text

```
$lvalue_string = $json->incr_text
```

This method returns the currently stored JSON fragment as an lvalue, that is, you can manipulate it. This *only* works when a preceding call to `incr_parse` in *scalar context* successfully returned an object. Under all other circumstances you must not call this function (I mean it. although in simple tests it might actually work, it *will* fail under real world conditions). As a special exception, you can also call this method before having parsed anything.

That means you can only use this function to look at or manipulate text before or after complete JSON objects, not while the parser is in the middle of parsing a JSON object.

This function is useful in two cases: a) finding the trailing text after a JSON object or b) parsing multiple JSON objects separated by non-JSON text (such as commas).

incr_skip

```
$json->incr_skip
```

This will reset the state of the incremental parser and will remove the parsed text from the input buffer so far. This is useful after `incr_parse` died, in which case the input buffer and incremental parser state is left unchanged, to skip the text parsed so far and to reset the parse state.

The difference to `incr_reset` is that only text until the parse error occurred is removed.

incr_reset

```
$json->incr_reset
```

This completely resets the incremental parser, that is, after this call, it will be as if the parser had never parsed anything.

This is useful if you want to repeatedly parse JSON objects and want to ignore any trailing data, which means you have to reset the parser after each successful decode.

MAPPING

Most of this section is also taken from JSON::XS.

This section describes how the backend modules map Perl values to JSON values and vice versa. These mappings are designed to ‘‘do the right thing’’ in most circumstances automatically, preserving round-tripping characteristics (what you put in comes out as something equivalent).

For the more enlightened: note that in the following descriptions, lowercase *perl* refers to the Perl interpreter, while uppercase *Perl* refers to the abstract Perl language itself.

JSON –> PERL

object

A JSON object becomes a reference to a hash in Perl. No ordering of object keys is preserved (JSON does not preserver object key ordering itself).

array

A JSON array becomes a reference to an array in Perl.

string

A JSON string becomes a string scalar in Perl – Unicode codepoints in JSON are represented by the same codepoints in the Perl string, so no manual decoding is necessary.

number

A JSON number becomes either an integer, numeric (floating point) or string scalar in perl, depending on its range and any fractional parts. On the Perl level, there is no difference between those as Perl handles all the conversion details, but an integer may take slightly less memory and might represent more values exactly than floating point numbers.

If the number consists of digits only, this module will try to represent it as an integer value. If that

fails, it will try to represent it as a numeric (floating point) value if that is possible without loss of precision. Otherwise it will preserve the number as a string value (in which case you lose roundtripping ability, as the JSON number will be re-encoded to a JSON string).

Numbers containing a fractional or exponential part will always be represented as numeric (floating point) values, possibly at a loss of precision (in which case you might lose perfect roundtripping ability, but the JSON number will still be re-encoded as a JSON number).

Note that precision is not accuracy – binary floating point values cannot represent most decimal fractions exactly, and when converting from and to floating point, this module only guarantees precision up to but not including the least significant bit.

true, false
: These JSON atoms become `JSON::true` and `JSON::false`, respectively. They are overloaded to act almost exactly like the numbers `1` and `0`. You can check whether a scalar is a JSON boolean by using the `JSON::is_bool` function.

null
: A JSON null atom becomes `undef` in Perl.

shell-style comments (`# text`)
: As a nonstandard extension to the JSON syntax that is enabled by the `relaxed` setting, shell-style comments are allowed. They can start anywhere outside strings and go till the end of the line.

tagged values (`(tag) value`).
: Another nonstandard extension to the JSON syntax, enabled with the `allow_tags` setting, are tagged values. In this implementation, the *tag* must be a perl package/class name encoded as a JSON string, and the *value* must be a JSON array encoding optional constructor arguments.

See ''OBJECT SERIALISATION'', below, for details.

PERL -> JSON

The mapping from Perl to JSON is slightly more difficult, as Perl is a truly typeless language, so we can only guess which JSON type is meant by a Perl value.

hash references
: Perl hash references become JSON objects. As there is no inherent ordering in hash keys (or JSON objects), they will usually be encoded in a pseudo-random order. This module can optionally sort the hash keys (determined by the *canonical* flag), so the same data structure will serialise to the same JSON text (given same settings and version of the same backend), but this incurs a runtime overhead and is only rarely useful, e.g. when you want to compare some JSON text against another for equality.

array references
: Perl array references become JSON arrays.

other references
: Other unblessed references are generally not allowed and will cause an exception to be thrown, except for references to the integers `0` and `1`, which get turned into `false` and `true` atoms in JSON. You can also use `JSON::false` and `JSON::true` to improve readability.

```
encode_json [\0,JSON::true]      # yields [false,true]
```

JSON::true, JSON::false, JSON::null
: These special values become JSON true and JSON false values, respectively. You can also use `\1` and `\0` directly if you want.

blessed objects
: Blessed objects are not directly representable in JSON, but `JSON::XS` allows various ways of handling objects. See ''OBJECT SERIALISATION'', below, for details.

simple scalars
: Simple Perl scalars (any scalar that is not a reference) are the most difficult objects to encode: this module will encode undefined scalars as JSON `null` values, scalars that have last been used in a string

context before encoding as JSON strings, and anything else as number value:

```
# dump as number
encode_json [2]                      # yields [2]
encode_json [-3.0e17]                # yields [-3e+17]
my $value = 5; encode_json [$value]  # yields [5]

# used as string, so dump as string
print $value;
encode_json [$value]                 # yields ["5"]

# undef becomes null
encode_json [undef]                  # yields [null]
```

You can force the type to be a string by stringifying it:

```
my $x = 3.1; # some variable containing a number
"$x";        # stringified
$x .= "";    # another, more awkward way to stringify
print $x;    # perl does it for you, too, quite often
```

You can force the type to be a number by numifying it:

```
my $x = "3"; # some variable containing a string
$x += 0;     # numify it, ensuring it will be dumped as a number
$x *= 1;     # same thing, the choice is yours.
```

You can not currently force the type in other, less obscure, ways. Tell me if you need this capability (but don't forget to explain why it's needed :).

Since version 2.91_01, JSON::PP uses a different number detection logic that converts a scalar that is possible to turn into a number safely. The new logic is slightly faster, and tends to help people who use older perl or who want to encode complicated data structure. However, this may results in a different JSON text from the one JSON::XS encodes (and thus may break tests that compare entire JSON texts). If you do need the previous behavior for better compatibility or for finer control, set PERL_JSON_PP_USE_B environmental variable to true before you `use` JSON.

Note that numerical precision has the same meaning as under Perl (so binary to decimal conversion follows the same rules as in Perl, which can differ to other languages). Also, your perl interpreter might expose extensions to the floating point numbers of your platform, such as infinities or NaN's – these cannot be represented in JSON, and it is an error to pass those in.

JSON.pm backend modules trust what you pass to `encode` method (or `encode_json` function) is a clean, validated data structure with values that can be represented as valid JSON values only, because it's not from an external data source (as opposed to JSON texts you pass to `decode` or `decode_json`, which JSON backends consider tainted and don't trust). As JSON backends don't know exactly what you and consumers of your JSON texts want the unexpected values to be (you may want to convert them into null, or to stringify them with or without normalisation (string representation of infinities/NaN may vary depending on platforms), or to croak without conversion), you're advised to do what you and your consumers need before you encode, and also not to numify values that may start with values that look like a number (including infinities/NaN), without validating.

OBJECT SERIALISATION

As JSON cannot directly represent Perl objects, you have to choose between a pure JSON representation (without the ability to deserialise the object automatically again), and a nonstandard extension to the JSON syntax, tagged values.

SERIALISATION

What happens when this module encounters a Perl object depends on the `allow_blessed`, `convert_blessed` and `allow_tags` settings, which are used in this order:

1. `allow_tags` is enabled and the object has a `FREEZE` method.
   In this case, `JSON` creates a tagged JSON value, using a nonstandard extension to the JSON syntax.

   This works by invoking the `FREEZE` method on the object, with the first argument being the object to serialise, and the second argument being the constant string `JSON` to distinguish it from other serialisers.

   The `FREEZE` method can return any number of values (i.e. zero or more). These values and the package/classname of the object will then be encoded as a tagged JSON value in the following format:

   ```
   ("classname")[FREEZE return values...]
   ```

   e.g.:

   ```
   ("URI")["http://www.google.com/"]
   ("MyDate")[2013,10,29]
   ("ImageData::JPEG")["Z3...VlCg=="]
   ```

   For example, the hypothetical `My::Object` `FREEZE` method might use the objects `type` and `id` members to encode the object:

   ```
   sub My::Object::FREEZE {
      my ($self, $serialiser) = @_;

      ($self->{type}, $self->{id})
   }
   ```

2. `convert_blessed` is enabled and the object has a `TO_JSON` method.
   In this case, the `TO_JSON` method of the object is invoked in scalar context. It must return a single scalar that can be directly encoded into JSON. This scalar replaces the object in the JSON text.

   For example, the following `TO_JSON` method will convert all URI objects to JSON strings when serialised. The fact that these values originally were URI objects is lost.

   ```
   sub URI::TO_JSON {
      my ($uri) = @_;
      $uri->as_string
   }
   ```

3. `allow_blessed` is enabled.
   The object will be serialised as a JSON null value.

4. none of the above
   If none of the settings are enabled or the respective methods are missing, this module throws an exception.

DESERIALISATION

For deserialisation there are only two cases to consider: either nonstandard tagging was used, in which case `allow_tags` decides, or objects cannot be automatically be deserialised, in which case you can use postprocessing or the `filter_json_object` or `filter_json_single_key_object` callbacks to get some real objects our of your JSON.

This section only considers the tagged value case: a tagged JSON object is encountered during decoding and `allow_tags` is disabled, a parse error will result (as if tagged values were not part of the grammar).

If `allow_tags` is enabled, this module will look up the `THAW` method of the package/classname used during serialisation (it will not attempt to load the package as a Perl module). If there is no such method, the decoding will fail with an error.

Otherwise, the `THAW` method is invoked with the classname as first argument, the constant string `JSON` as second argument, and all the values from the JSON array (the values originally returned by the `FREEZE` method) as remaining arguments.

The method must then return the object. While technically you can return any Perl scalar, you might have to enable the `allow_nonref` setting to make that work in all cases, so better return an actual blessed reference.

As an example, let's implement a `THAW` function that regenerates the `My::Object` from the `FREEZE` example earlier:

```
sub My::Object::THAW {
   my ($class, $serialiser, $type, $id) = @_;

   $class->new (type => $type, id => $id)
}
```

ENCODING/CODESET FLAG NOTES

This section is taken from JSON::XS.

The interested reader might have seen a number of flags that signify encodings or codesets – `utf8`, `latin1` and `ascii`. There seems to be some confusion on what these do, so here is a short comparison:

`utf8` controls whether the JSON text created by `encode` (and expected by `decode`) is UTF–8 encoded or not, while `latin1` and `ascii` only control whether `encode` escapes character values outside their respective codeset range. Neither of these flags conflict with each other, although some combinations make less sense than others.

Care has been taken to make all flags symmetrical with respect to `encode` and `decode`, that is, texts encoded with any combination of these flag values will be correctly decoded when the same flags are used – in general, if you use different flag settings while encoding vs. when decoding you likely have a bug somewhere.

Below comes a verbose discussion of these flags. Note that a "codeset" is simply an abstract set of character-codepoint pairs, while an encoding takes those codepoint numbers and *encodes* them, in our case into octets. Unicode is (among other things) a codeset, UTF–8 is an encoding, and ISO–8859–1 (= latin 1) and ASCII are both codesets *and* encodings at the same time, which can be confusing.

`utf8` flag disabled

When `utf8` is disabled (the default), then `encode`/`decode` generate and expect Unicode strings, that is, characters with high ordinal Unicode values (> 255) will be encoded as such characters, and likewise such characters are decoded as-is, no changes to them will be done, except "(re–)interpreting" them as Unicode codepoints or Unicode characters, respectively (to Perl, these are the same thing in strings unless you do funny/weird/dumb stuff).

This is useful when you want to do the encoding yourself (e.g. when you want to have UTF–16 encoded JSON texts) or when some other layer does the encoding for you (for example, when printing to a terminal using a filehandle that transparently encodes to UTF–8 you certainly do NOT want to UTF–8 encode your data first and have Perl encode it another time).

`utf8` flag enabled

If the `utf8`–flag is enabled, `encode`/`decode` will encode all characters using the corresponding UTF–8 multi-byte sequence, and will expect your input strings to be encoded as UTF–8, that is, no "character" of the input string must have any value > 255, as UTF–8 does not allow that.

The `utf8` flag therefore switches between two modes: disabled means you will get a Unicode string in Perl, enabled means you get an UTF–8 encoded octet/binary string in Perl.

`latin1` or `ascii` flags enabled

With `latin1` (or `ascii`) enabled, `encode` will escape characters with ordinal values > 255 (> 127 with `ascii`) and encode the remaining characters as specified by the `utf8` flag.

If `utf8` is disabled, then the result is also correctly encoded in those character sets (as both are proper subsets of Unicode, meaning that a Unicode string with all character values < 256 is the same thing as a ISO–8859–1 string, and a Unicode string with all character values < 128 is the same thing as an ASCII string in Perl).

If `utf8` is enabled, you still get a correct UTF-8-encoded string, regardless of these flags, just some more characters will be escaped using `\uXXXX` then before.

Note that ISO-8859-1-*encoded* strings are not compatible with UTF-8 encoding, while ASCII-encoded strings are. That is because the ISO-8859-1 encoding is NOT a subset of UTF-8 (despite the ISO-8859-1 *codeset* being a subset of Unicode), while ASCII is.

Surprisingly, `decode` will ignore these flags and so treat all input values as governed by the `utf8` flag. If it is disabled, this allows you to decode ISO-8859-1- and ASCII-encoded strings, as both strict subsets of Unicode. If it is enabled, you can correctly decode UTF-8 encoded strings.

So neither `latin1` nor `ascii` are incompatible with the `utf8` flag - they only govern when the JSON output engine escapes a character or not.

The main use for `latin1` is to relatively efficiently store binary data as JSON, at the expense of breaking compatibility with most JSON decoders.

The main use for `ascii` is to force the output to not contain characters with values > 127, which means you can interpret the resulting string as UTF-8, ISO-8859-1, ASCII, KOI8-R or most about any character set and 8-bit-encoding, and still get the same data structure back. This is useful when your channel for JSON transfer is not 8-bit clean or the encoding might be mangled in between (e.g. in mail), and works because ASCII is a proper subset of most 8-bit and multibyte encodings in use in the world.

BACKWARD INCOMPATIBILITY

Since version 2.90, stringification (and string comparison) for `JSON::true` and `JSON::false` has not been overloaded. It shouldn't matter as long as you treat them as boolean values, but a code that expects they are stringified as "true" or "false" doesn't work as you have expected any more.

```
if (JSON::true eq 'true') {  # now fails

print "The result is $JSON::true now."; # => The result is 1 now.
```

And now these boolean values don't inherit JSON::Boolean, either. When you need to test a value is a JSON boolean value or not, use `JSON::is_bool` function, instead of testing the value inherits a particular boolean class or not.

BUGS

Please report bugs on backend selection and additional features this module provides to RT or GitHub issues for this module:

<https://rt.cpan.org/Public/Dist/Display.html?Queue=JSON>

<https://github.com/makamaka/JSON/issues>

As for bugs on a specific behavior, please report to the author of the backend module you are using.

As for new features and requests to change common behaviors, please ask the author of JSON::XS (Marc Lehmann, <schmorp[at]schmorp.de>) first, by email (important!), to keep compatibility among JSON.pm backends.

SEE ALSO

JSON::XS, Cpanel::JSON::XS, JSON::PP for backends.

JSON::MaybeXS, an alternative that prefers Cpanel::JSON::XS.

`RFC4627`(<http://www.ietf.org/rfc/rfc4627.txt>)

RFC7159 (<http://www.ietf.org/rfc/rfc7159.txt>)

RFC8259 (<http://www.ietf.org/rfc/rfc8259.txt>)

AUTHOR

Makamaka Hannyaharamitu, <makamaka[at]cpan.org>

JSON::XS was written by Marc Lehmann <schmorp[at]schmorp.de>

The release of this new version owes to the courtesy of Marc Lehmann.

CURRENT MAINTAINER

Kenichi Ishigaki, <ishigaki[at]cpan.org>

COPYRIGHT AND LICENSE