



# class BSON::Document

BSON Encodable and Decodable document

## Table of Contents

- 1 [Synopsis](#)
- 2 [Description](#)
- 3 [Supported types](#)
- 4 [Operators](#)
  - 4.1 [postcircumfix { }](#)
  - 4.2 [postcircumfix < >](#)
- 5 [Methods](#)
  - 5.1 [new](#)
  - 5.2 [perl](#)
  - 5.3 [Str](#)
  - 5.4 [autovivify](#)
  - 5.5 [accept-hash](#)
  - 5.6 [convert-rat](#)
  - 5.7 [find-key](#)
  - 5.8 [of](#)
  - 5.9 [method elems](#)
  - 5.10 [kv](#)
  - 5.11 [pairs](#)
  - 5.12 [keys](#)
  - 5.13 [values](#)
  - 5.14 [modify-array](#)
  - 5.15 [encode](#)
  - 5.16 [decode](#)

```
unit package BSON;  
class Document does Associative { ... }
```

## Synopsis

```
use BSON::Document;  
  
# Document usage  
my BSON::Document $d .= new;  
$d<name> = 'Jose';  
$d<address> = street => '24th', city => 'NY';  
$d<keywords> = [<perl6 language programming>];  
  
# Automatic generating subdocuments  
$d.autovivify(True);  
$d<a><b><c><d><e> = 10;  
  
# Encoding and decoding  
my Buf $b = $d.encode;  
my BSON::Document $d2 .= new;  
$d2.decode($b);
```

# Description

Document storage with Hash like behavior used mainly to communicate with a mongodb server. It can also be used as a serialized storage format. The main difference with the Hash is that this class keeps the input order of inserted key-value pairs which is important for the use with mongodb.

Every form of nesting with e.g. pairs is converted into a `BSON::Document`. Other classes are needed to handle types such as Javascript, ObjectId and Binary. These classes are automatically loaded when `BSON::Document` is loaded.

E.g.

```
use BSON::Document;

my BSON::Document $d .= new;
$d<javascript> = BSON::Javascript.new(:javascript("function(x){return x;}"));
$d<datetime> = DateTime.now;
$d<regex> = BSON::Regex.new( :regex('abc|def'), :options<is>);
```

# Supported types

There are BSON specifications [mentioned on their site](#) which are deprecated or used internally only. These are not implemented.

There are quite a few more perl6 container types like (Fat)Rat, Bag, Set etc. Now binary types are possible it might be an idea to put these perl6 types into binary. There are 127 user definable types in that BSON binary specification, so place enough to put it there, also because when javascript is run on the server it would not be able to cope with these types.

Impl	Type/ subtype	BSON spec	Perl6	Note
x	1	64-bit Double	Num	Rat is converted to num
x	2	UTF-8 string	Str	
x	3	Embedded document.	BSON::Document	
x	4	Array document	Array	
x	5	All kinds of binary data	BSON::Binary	
x	5/0	Generic type		
	5/1	Function		
-	5/2	Binary old, deprecated		
-	5/3	UUID old, deprecated		
x	5/4	UUID		
x	5/5	MD5		
	5/128	Int larger/smaller than 64 bit	Int	
	5/129		FatRat	
	5/130		Rat	When precision is an issue
-	6	Undefined value - Deprecated		
x	7	ObjectId	BSON::ObjectId	
x	8	Boolean "true" / "false"	Bool	
x	9	int64 UTC datetime	DateTime	
x	10	Null value	Undefined type	
x	11	Regular expression(perl 5 like)	BSON::Regex	
-	12	DBPointer - Deprecated		
x	13	Javascript code	BSON::Javascript	
-	14	Symbol - Deprecated		
x	15	Javascript code with scope	BSON::Javascript	
x	16	32 bit integers.	Int	
x	17	Timestamp, converted to uint64.	BSON::Timestamp	
x	18	64 bit integers.	Int	
	19	128 bit decimal floating point	Decimal::D128	

## Encoding/Decoding a bytestream from/to perl6

The types currently supported are marked with a 'x'. '-' will not be implemented and <empty> is a future thingy.

## Operators

### postcircumfix { }

```
$d{'full address'} = 'my-street 45, new york';
```

### postcircumfix < >

```
$d<name> = 'Mr Foo and Mrs Bar';
```

## Methods

### new

```
multi method new ( List $l = () )
multi method new ( Pair $p )
multi method new ( Seq $s )
multi method new ( Buf $b )
```

Some examples to call new

```
my BSON::Document $d;

# empty document
$d .= new;

# Initialize with a Buf, Previously received from a mongodb server or
# from a previous encoding
$d .= new($bson-encoded-document);

# Initialize with a Seq
$d .= new: ('a' ... 'z') Z=> 120..145;

# Initialize with a List
$d .= new: ( a => 10, b => 11);
```

Initialize a new document.

## perl

```
method perl ( --> Str )
```

Return objects structure.

## Str

```
method Str ( --> Str )
```

Return type and location of the object.

## autovivify

```
method autovivify ( Bool :$on = True, Bool :$instance-only = False )
```

By default it is set to **False** and will throw an exception with an message like 'Cannot modify an immutable Any' when an attempt is made like in the following.piece of code

```
my BSON::Document $d .= new;
$d<a><b> = 10;          # Throw error
```

To have this feature one must turn this option on like so;

```
my BSON::Document $d .= new;
$d.autovivify(:on);
$d<a><b> = 10;
```

**NOTE:** Testing for items will also create the entries if they weren't there.**NOTE:** :instance-only does not have much use at the moment.

## accept-hash

```
method accept-hash ( Bool :$accept = True, Bool :$instance-only = False )
```

By default it is set to `False` and will throw an exception with a message like 'Cannot use hash values'. This is explicitly done to keep input order. When it is turned off try something like below to see what is meant;

```
my BSON::Document $d .= new;
$d.accept-hash(:accept);
$d<q> = {
  a => 120, b => 121, c => 122, d => 123, e => 124, f => 125, g => 126,
  h => 127, i => 128, j => 129, k => 130, l => 131, m => 132, n => 133,
  o => 134, p => 135, q => 136, r => 137, s => 138, t => 139, u => 140,
  v => 141, w => 142, x => 143, y => 144, z => 145
};

say $d<q>.keys;
# Outputs [x p k h g z a y v s q e d m f c w o n u t b j i r l]
```

**NOTE: :instance-only does not have much use at the moment.**

## convert-rat

```
method convert-rat (
  Bool $accept = True,
  Bool :$accept-precision-loss = False,
  Bool :$instance-only = False
)
```

When rational numbers are accepted, they will be converted to `double(Num)`, if precision is not compromised. If the precision is compromised then an exception is thrown. However when `:accept-precision-loss` is also set then the conversion always takes place regardless of precision loss. This is useful for dealing with messy input, since rational numbers cannot be encoded at this time.

Plans are to convert to binary when precision is an issue instead of throwing an exception. One cannot use server side calculations then when converted to binary.

**NOTE: :instance-only does not have much use at the moment.**

## find-key

```
multi method find-key ( Int:D $idx --> Str )
multi method find-key ( Str:D $key --> Int )
```

Search for index and find key or search for key and return index. It returns an undefined value if `$idx` or `$key` is not found.

```
use Test;
use BSON::Document;
my $d = BSON::Document.new: ('a' ... 'z') Z=> 120..145;

is $d<b>, $d[$d.find-key('b')], 'Value on key and found index are the same';
is $d.find-key(2), 'c', "Index 2 is mapped to key 'c'";
```

## of

```
method of ( )
```

Returns type of object. NOTE: I'm not sure if this is the normal practice of such a method. Need to investigate further

## method elems

```
method elems ( --> Int )
```

Return the number of pairs in the document

## kv

```
method kv ( --> List )
```

Return a list of keys and values in the same order as entered.

```
use BSON::Document;
my $d = BSON::Document.new: ('a' ... 'z') Z=> 120..145;
say $d.kv;
# Outputs: [a 120 b 121 c 122 d 123 ... x 143 y 144 z 145]
```

## pairs

```
method pairs ( --> List )
```

Return a list of pairs in the same order as entered.

## keys

```
method keys ( --> List )
```

Return a list of keys in the same order as entered.

```
use BSON::Document;
my $d = BSON::Document.new: ('a' ... 'z') Z=> 120..145;
say $d.keys;
# Outputs: [a b c d ... x y z]
```

## values

```
method values ( --> List )
```

Return a list of value in the same order as entered.

```
use BSON::Document;
my $d = BSON::Document.new: ('a' ... 'z') Z=> 120..145;
say $d.values;
# Outputs: [120 121 122 123 ... 143 144 145]
```

## modify-array

```
method modify-array ( Str $key, Str $operation, $data --> List )
```

Use as

```
BSON::Document $d .= new:(docs => []);
$d.modify-array( 'docs', 'push', (a => 1, b => 2));
```

Modify an array in a document afterwards. This method is necessary to apply changes because when doing it directly like **\$d<docs>.push: (c = 2);**> it wouldn't be encoded because the document object is not aware of these changes.

This is a slow method because every change will trigger an encoding procedure in the background. When a whole array needs to be entered then it is a lot faster to make the array first and then assign it to an entry in the document e.g;

```
BSON::Document $d .= new;  
my $arr = [];  
for ^10 -> $i {  
    $arr.push($i);  
}  
$d<myarray> = $arr;
```

## encode

```
method encode ( --> Buf )
```

Encode entire document and return a BSON encoded byte buffer.

## decode

```
method decode ( Buf $data --> Nil )
```

Decode a BSON encoded byte buffer to produce a document. Decoding also takes place when providing a byte buffer to `new()`.