

SCHEMATA

Mariano Montone (marianomontone@gmail.com)

Table of Contents

1	Introduction	1
2	Installation	2
3	Usage	3
3.1	Schema definition	3
3.2	Validation using schemas	3
3.3	Serialization using schemas	3
3.4	Unserialization using schemas	4
3.5	Patch and updates	4
4	Schema types	5
4.1	Type schemas	5
4.2	Object schema	5
4.3	List schema	5
4.4	Schema references	5
4.5	SATISFIES-SCHEMA type	6
4.6	SCHEMA-CLASS metaclass	6
5	Data generation	8
6	Reference	9
6.1	SCHEMATA package	9
7	Index	16

1 Introduction

Generic purpose schema library for serialization and validation of data.

This library is used by CL-REST-SERVER for API serialization and validation.

It features:

- Validation using schemas.
- Serialization and unserialization using schemas.
- Generation of (random) data from schemas. (incomplete implementation atm)
- JSON-Schema parsing (incomplete implementation atm).
- Integration with Common Lisp type system.
- A schema class metaclass.

2 Installation

With Quicklisp:

```
(ql:quickload "schemata")
```

3 Usage

3.1 Schema definition

Use [DEFSHEMA], page 9 for defining schemas.

Schema example:

```
(schemata:defschema customer
  (object "customer"
    ((id string :external-name "id" :accessor
      customer-id :documentation "customer id")
      (number string :external-name "number" :required nil
        :accessor customer-nr :documentation
          "customer number")
      (name string :external-name "name" :accessor
        customer-name :documentation "customer name")
      (address-1 string :external-name "address1"
        :required nil :documentation
          "customer first address")
      (address-2 string :external-name "address2"
        :required nil :documentation
          "customer second address")
      (postal-code string :external-name "postalcode"
        :required nil :documentation
          "postal code")
      (postal-area string :external-name "postalarea"
        :required nil :documentation
          "postal area")
      (country string :external-name "country" :required nil
        :documentation "country code")
      (phone string :external-name "phone" :required nil
        :documentation "phone")
      (fax string :external-name "fax" :required nil
        :documentation "fax")
      (email string :external-name "email" :required nil
        :documentation "email"))
    (:documentation "customer data fetched")))
```

3.2 Validation using schemas

Use [VALIDATE-WITH-SCHEMA], page 10.

3.3 Serialization using schemas

Via *generic-serializer* library.

Use [SERIALIZE-WITH-SCHEMA], page 10.

```
(with-output-to-string (s)
```

```
(gs:with-serializer-output s
  (gs:with-serializer :json
    (serialize-with-schema *schema* user))))
(with-output-to-string (s)
  (gs:with-serializer-output s
    (gs:with-serializer :json
      (serialize-with-schema
        (find-schema 'user-schema) *user*))))
```

3.4 Unserialization using schemas

Use [UNSERIALIZE-WITH-SCHEMA], page 9.

```
(unserialize-with-schema
  (find-schema 'user-schema)
  (json:decode-json-from-string data)
  :json)
```

3.5 Patch and updates

See [PATCH-WITH-SCHEMA], page 10 and [POPULATE-WITH-SCHEMA], page 10.

4 Schema types

4.1 Type schemas

Schemas can be built from Common Lisp types:

```
SCHEMATA> (defparameter *s* (schema string))
*S*
SCHEMATA> *s*
#<TYPE-SCHEMA STRING {1006FBBD13}>
SCHEMATA> (validate-with-schema *s* "22")
NIL
SCHEMATA> (validate-with-schema *s* 22 :error-p nil)
#<VALIDATION-ERROR "~s is not of type: ~a" {100152EB13}>
```

4.2 Object schema

Object schemas are built using the syntax: (object name attributes options). Attributes are specified as: (attribute-name attribute-type &rest options).

The `attribute-type` is parsed as a schema.

Possible attribute options are: ‘required’, ‘required-message’, ‘default’, ‘accessor’, ‘writer’, ‘reader’, ‘parser’, ‘validator’, ‘add-validator’, ‘formatter’, ‘external-name’, ‘serializer’, ‘unserializer’, ‘slot’.

Example:

```
SCHEMATA> (schema (object person
                    ((name string)
                     (age integer :required nil))))
#<OBJECT-SCHEMA {1001843543}>
```

4.3 List schema

Homogeneous list of schemas are specified via `list-of`.

Example:

```
SCHEMATA> (defparameter *s* (schema (list-of integer)))
*S*
SCHEMATA> (validate-with-schema *s* '(1 2 "foo"))
; Evaluation aborted on #<SCHEMATA:VALIDATION-ERROR "~s is not of type: ~a" {1006ECA323}>.
SCHEMATA> (validate-with-schema *s* '(1 2 3))
NIL
```

4.4 Schema references

Defined schemas can be referenced via either ‘(schema schema-name)’ or ‘(ref schema-name)’ (they are identical).

Example:

```
SCHEMATA> (defschema person
```



```

      (object person
        ((name string))))
#<OBJECT-SCHEMA {1006F8A813}>
SCHEMATA> (defparameter *list-of-person* (schema (list-of (ref person))))
*LIST-OF-PERSON*
SCHEMATA> *list-of-person*
#<LIST-SCHEMA {1006F8C2A3}>
SCHEMATA> (parse-with-schema *list-of-person* '(((("name" . "Mariano")) ((("name" . "Peter"))
(((NAME . "Mariano")) ((NAME . "Peter"))))
SCHEMATA> (validate-with-schema *list-of-person* '(((("name" . 22)) ((("name" . "Peter")))))
; processing (DEFMETHOD SCHEMA-VALIDATE ...); Evaluation aborted on #<SB-PCL::NO-APPLICABLE
SCHEMATA> (validate-with-schema *list-of-person* '(((("name" . 22)) ((("name" . "Peter")))))
; Evaluation aborted on #<SCHEMATA:VALIDATION-ERROR "~s is not of type: ~a" {10082EB883}>..
SCHEMATA> (validate-with-schema *list-of-person* '(((("name" . "Mariano")) ((("name" . "Peter
NIL
SCHEMATA> (validate-with-schema *list-of-person* '(((("names" . "Mariano")) ((("name" . "Pete
; Evaluation aborted on #<SCHEMATA:VALIDATION-ERROR "Attributes not part of schema: ~a" {10

```

4.5 SATISFIES-SCHEMA type

Schemata integrates with the Lisp type system via the SATISFIES-SCHEMA type. Schemas can be thought as types over data. Defined schemas can be checked using TYPEP and CHECK-TYPE with the type ‘(satisfies-schema schema-name)’.

Example:

```

SCHEMATA> (defschema string-schema string)
#<TYPE-SCHEMA STRING {10019DA8B3}>
SCHEMATA> (typep "foo" '(satisfies-schema string-schema))
T
SCHEMATA> (typep 22 '(satisfies-schema string-schema))
NIL
SCHEMATA> (let ((x "foo"))
  (check-type x (satisfies-schema string-schema))
  x)
"foo"

```

4.6 SCHEMA-CLASS metaclass

SCHEMA-CLASS classes get an schema attached.

Example:

```

SCHEMATA> (def-schema-class person ()
  ((name :type string :initarg :name)
   (age :type integer :required nil :initarg :age)))
#<SCHEMA-CLASS SCHEMATA::PERSON>

SCHEMATA> (validate-with-schema (find-class 'person) '(((("name" . "Mariano") ("age" . 22))))
NIL

```

```
SCHEMATA> (validate-with-schema (find-class 'person) '(("name" . "Mariano") ("age" . 'asdf))  
#<VALIDATION-ERROR 'ASDF is not of type: INTEGER {100109F833}>
```

```
SCHEMATA> (generic-serializer:with-serializer :json  
            (generic-serializer:serialize (make-instance 'person :name "Mariano" :age 44)))  
{"name":"Mariano","age":44}
```

5 Data generation

Schemata can generate random data from schemas. It uses `check-it` library generators for that.

Load `schemata-generators` system.

Then call `check-it:generate` with a schema object.

Example:

```
(defschema person
  (object person
    ((name string)
     (age integer :required nil)
     (friend (ref person) :required nil))))
(generate (find-schema 'person))
```

can generate:

```
((NAME . "21p7E0w8")
 (FRIEND (NAME . "hD39Dwo")
  (FRIEND (NAME . "QFC67xg206") (AGE . 4)
   (FRIEND (NAME . "bRtUL1z51")
    (FRIEND (NAME . "0")
     (FRIEND (NAME . "ddB57idmh32C4T") (AGE . 1)
      (FRIEND (NAME . "eNKzc") (AGE . 8))))))))))
```

For more control over the generation, attach a generator to schemas via `:generator` initarg.

6 Reference

6.1 SCHEMATA package

SCHEMATA [PACKAGE]

External definitions

Macros

SCHEMATA:SCHEMA (*schema-def*) [Macro]
 Wrapper macro for schema definitions.

SCHEMATA:DEFSHEMA (*name schema*) [Macro]
 Register SCHEMA under NAME. The schema can then be accessed via
 FIND-SCHEMA.

SCHEMATA:DEF-SCHEMA-CLASS (*name direct-superclasses direct-slots* [Macro]
&rest options)
 Helper macro to define schema classes

Generic functions

SCHEMATA:SCHEMA-TYPE (*sb-pcl::object*) [Generic-Function]

SCHEMATA:OBJECT-NAME (*sb-pcl::object*) [Generic-Function]

SCHEMATA:OBJECT-CLASS (*sb-pcl::object*) [Generic-Function]

SCHEMATA:SCHEMA-GENERATOR (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-REQUIRED-P (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-ACCESSOR (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-ADD-VALIDATOR (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-EXTERNAL-NAME (*sb-pcl::object*) [Generic-Function]

SCHEMATA:UNSERIALIZE-WITH-SCHEMA (*schema data format*) [Generic-Function]

SCHEMATA:ATTRIBUTE-FORMATTER (*sb-pcl::object*) [Generic-Function]

SCHEMATA:SCHEMA-DOCUMENTATION (*sb-pcl::object*) [Generic-Function]

SCHEMATA:PARSE-WITH-SCHEMA (*schema string-or-data*) [Generic-Function]
 Parses the string to an association list using the schema

SCHEMATA:ATTRIBUTE-TYPE (*sb-pcl::object*) [Generic-Function]

SCHEMATA:OBJECT-ATTRIBUTES (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-NAME (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-PARSER (*sb-pcl::object*) [Generic-Function]

SCHEMATA:ATTRIBUTE-VALIDATOR (*sb-pcl::object*) [Generic-Function]

Functions

SCHEMATA:ATTRIBUTE-TYPE-NAME (*attribute*) [Function]

SCHEMATA:ATTRIBUTE-READER (*attribute*) [Function]

SCHEMATA:POPULATE-WITH-SCHEMA (*schema object data* **&key** *exclude*) [Function]
 Populate CLOS objects from data + schema. Attributes members of EXCLUDE parameter are not populated.

SCHEMATA:ATTRIBUTE-OPTIONAL-P (*attribute*) [Function]

SCHEMATA:SCHEMA-CLASS-SCHEMA (*schema-class*) [Function]
 Generate a schema using the schema class meta info

SCHEMATA:SERIALIZE-WITH-SCHEMA (*schema input* **&optional** (*serializer generic-serializer::*serializer**) (*stream generic-serializer::*serializer-output**)) [Function]

SCHEMATA:SCHEMA-SPEC (*schema*) [Function]

SCHEMATA:FIND-OBJECT-ATTRIBUTE (*object attribute-name* **&key** (*error-p t*)) [Function]

SCHEMATA:PATCH-WITH-SCHEMA (*schema object data*) [Function]
 Populate CLOS objects from data + schema. Only populates attributes available in DATA, validating them. Useful for PATCH rest api operations implementations. DATA should be an association list.

SCHEMATA:ATTRIBUTE-WRITER (*attribute*) [Function]

SCHEMATA:VALIDATION-ERROR (*message* **&rest** *args*) [Function]

SCHEMATA:FIND-SCHEMA (*name* **&optional** (*error-p t*)) [Function]
 Find a schema definition by name

SCHEMATA:VALIDATE-WITH-SCHEMA (*schema data* **&key** (*collect-errors* **collect-validation-errors**) (*error-p* **signal-validation-errors**)) [Function]

Validate input using schema. Useful for validating resource operations posted content (for :post and :put methods). Input can be a string or an association list.

Args: - schema (symbol or schema): The schema - data (alist): The data to validate.
 - format (keyword): The data format. - collect-errors (boolean): If true, collect all the validation errors. If false, return the first validation error found. Default: true.
 - error-p (boolean): If true, when validation errors are found, a validation error is signaled. If false, the validation errors are returned as the function result and no error is signaled.

SCHEMATA:GENERATE-SCHEMA-FROM-CLASS (*class*) [Function]
 Generate a schema from CLASS, using reflection.

Classes

SCHEMATA:OBJECT-SCHEMA

[Class]

Class precedence list: `object-schema`, `schema`, `standard-object`, `t`

Slots:

- `name` — type: (or string symbol); initarg: `:name`; reader: `schemata:object-name`; writer: `(setf schemata:object-name)`
The name of the object.
- `attributes` — type: list; initarg: `:attributes`; reader: `schemata:object-attributes`; writer: `(setf schemata:object-attributes)`
- `class` — type: (or null symbol); initarg: `:class`; reader: `schemata:object-class`; writer: `(setf schemata:object-class)`
- `ignore-unknown-attributes` — type: boolean; initarg: `:ignore-unknown-attributes`; reader: `schemata::ignore-unknown-attributes`; writer: `(setf schemata::ignore-unknown-attributes)`
- `serializer` — type: (or null trivial-types:function-designator); initarg: `:serializer`; reader: `schemata::object-serializer`; writer: `(setf schemata::object-serializer)`
- `unserializer` — type: (or null trivial-types:function-designator); initarg: `:unserializer`; reader: `schemata::object-unserializer`; writer: `(setf schemata::object-unserializer)`

SCHEMATA:CONS-SCHEMA

[Class]

Schema for CONSes.

Syntax: `(cons car-schema cdr-schema)`

Examples:

`(schema (cons symbol string))`

Class precedence list: `cons-schema`, `schema`, `standard-object`, `t`

Slots:

- `car-schema` — type: `t`; initarg: `:car-schema`; reader: `schemata::car-schema`; writer: `(setf schemata::car-schema)`
The schema of CAR.
- `cdr-schema` — type: `t`; initarg: `:cdr-schema`; reader: `schemata::cdr-schema`; writer: `(setf schemata::cdr-schema)`
The schema of CDR.

SCHEMATA:LIST-OF-SCHEMA

[Class]

Schema for list with elements of certain type/schema.

Syntax: `(list-of schema)`

Examples:

`(schema (list-of string))` `(schema (list-of (or string number)))`

Class precedence list: `list-of-schema`, `schema`, `standard-object`, `t`

Slots:

- `elements-schema` — type: (not null); initarg: `:elements-schema`; reader: `schemata::elements-schema`; writer: `(setf schemata::elements-schema)`

Schema of the elements of the list

SCHEMATA:PLIST-OF-SCHEMA [Class]

Class precedence list: `plist-of-schema`, `schema`, `standard-object`, `t`

Slots:

- `key-schema` — type: `t`; initarg: `:key-schema`; reader: `schemata::key-schema`; writer: `(setf schemata::key-schema)`
- `value-schema` — type: `t`; initarg: `:value-schema`; reader: `schemata::value-schema`; writer: `(setf schemata::value-schema)`

SCHEMATA:SCHEMA [Class]

Class precedence list: `schema`, `standard-object`, `t`

Slots:

- `documentation` — type: (or null string); initarg: `:documentation`; reader: `schemata:schema-documentation`; writer: `(setf schemata:schema-documentation)`
- `generator` — type: `t`; initarg: `:generator`; reader: `schemata:schema-generator`; writer: `(setf schemata:schema-generator)`

SCHEMATA:ALIST-OF-SCHEMA [Class]

Schema for association lists with certain type of keys and values.

Syntax: `(alist-of (key-schema . value-schema) &rest options)`

Examples:

`(schema (alist-of (keyword . string)))`

Class precedence list: `alist-of-schema`, `schema`, `standard-object`, `t`

Slots:

- `key-schema` — type: `t`; initarg: `:key-schema`; reader: `schemata::key-schema`; writer: `(setf schemata::key-schema)`
- `value-schema` — type: `t`; initarg: `:value-schema`; reader: `schemata::value-schema`; writer: `(setf schemata::value-schema)`

SCHEMATA:PLIST-SCHEMA [Class]

Class precedence list: `plist-schema`, `schema`, `standard-object`, `t`

Slots:

- `members` — type: `t`; initarg: `:members`; reader: `schemata::plist-members`; writer: `(setf schemata::plist-members)`
- `required-keys` — type: `t`; initarg: `:required-keys`; reader: `schemata::required-keys`; writer: `(setf schemata::required-keys)`
- `optional-keys` — type: `t`; initarg: `:optional-keys`; reader: `schemata::optional-keys`; writer: `(setf schemata::optional-keys)`
- `allow-other-keys` — type: `t`; initarg: `:allow-other-keys`; reader: `schemata::allow-other-keys-p`; writer: `(setf schemata::allow-other-keys-p)`

SCHEMATA:LIST-SCHEMA

[Class]

Schema for lists.

Syntax: (list &rest schemas)

Examples:

```
(schema (list string number)) (schema (list symbol number boolean))
```

Data matches when it is a list of the same size and the list schemas match. For instance, for the schema: (list symbol number symbol), '(foo 33 bar) matches, but '(foo 33) does not.

Class precedence list: `list-schema`, `schema`, `standard-object`, `t`

Slots:

- `schemas` — type: `t`; initarg: `:schemas`; reader: `schemata::list-schemas`; writer: `(setf schemata::list-schemas)`

SCHEMATA:SCHEMA-REFERENCE-SCHEMA

[Class]

Class precedence list: `schema-reference-schema`, `schema`, `standard-object`, `t`

Slots:

- `name` — type: `symbol`; initarg: `:schema-name`; reader: `schemata::schema-name`; writer: `(setf schemata::schema-name)`

SCHEMATA:ALIST-SCHEMA

[Class]

Schema for association lists with certain keys and values.

Syntax: (alist alist &rest options)

where alist is a list of conses with key and schema.

Examples:

```
(schema (alist ((:x . string)(:y . number)))) (schema (alist ((:x . string)(:y . number))
:optional-keys (:y)))
```

Class precedence list: `alist-schema`, `schema`, `standard-object`, `t`

Slots:

- `members` — type: `t`; initarg: `:members`; reader: `schemata::alist-members`; writer: `(setf schemata::alist-members)`
- `required-keys` — type: (or boolean list); initarg: `:required-keys`; reader: `schemata::required-keys`; writer: `(setf schemata::required-keys)`

If T (default), all keys are considered required. If a list, only those listed are considered required.

- `optional-keys` — type: (or boolean list); initarg: `:optional-keys`; reader: `schemata::optional-keys`; writer: `(setf schemata::optional-keys)`

If T, then all keys are considered optional. If a list, then the keys listed are considered optional.

- `allow-other-keys` — type: `t`; initarg: `:allow-other-keys`; reader: `schemata::allow-other-keys-p`; writer: `(setf schemata::allow-other-keys-p)`■

Whether other keys than the specified are allowed in the data being checked.

SCHEMATA:SCHEMA-CLASS [Class]

Metaclass for schema objects

Class precedence list: `schema-class`, `standard-class`, `class`, `specializer`, `metaobject`, `standard-object`, `t`

Slots:

- `schema-name` — type: (or null string symbol); initarg: `:schema-name`; reader: `schemata::schema-name`; writer: (setf `schemata::schema-name`)

SCHEMATA:SCHEMA-OBJECT [Class]

Class precedence list: `schema-object`, `standard-object`, `t`

SCHEMATA:TYPE-SCHEMA [Class]

Schema for a Common Lisp type.

Syntax: (schema type)

Examples:

(schema string) (schema integer)

Class precedence list: `type-schema`, `schema`, `standard-object`, `t`

Slots:

- `type` — type: `t`; initarg: `:type`; reader: `schemata:schema-type`; writer: (setf `schemata:schema-type`)

SCHEMATA:ATTRIBUTE-PROPERTIES [Class]

Class precedence list: `attribute-properties`, `standard-object`, `t`

Slots:

- `required` — type: boolean; initarg: `:required`; reader: `schemata:attribute-required-p`; writer: (setf `schemata:attribute-required-p`)
- `required-message` — type: (or string null); initarg: `:required-message`; reader: `schemata::attribute-required-message`; writer: (setf `schemata::attribute-required-message`)
- `default` — type: `t`; initarg: `:default`; reader: `schemata::attribute-default`; writer: (setf `schemata::attribute-default`)
- `validator` — type: (or null trivial-types:function-designator); initarg: `:validator`; reader: `schemata:attribute-validator`; writer: (setf `schemata:attribute-validator`)
- `add-validator` — type: (or null trivial-types:function-designator); initarg: `:add-validator`; reader: `schemata:attribute-add-validator`; writer: (setf `schemata:attribute-add-validator`)
- `parser` — type: (or null trivial-types:function-designator); initarg: `:parser`; reader: `schemata:attribute-parser`; writer: (setf `schemata:attribute-parser`)
- `formatter` — type: (or null trivial-types:function-designator); initarg: `:formatter`; reader: `schemata:attribute-formatter`; writer: (setf `schemata:attribute-formatter`)
- `external-name` — type: (or string null); initarg: `:external-name`; reader: `schemata:attribute-external-name`; writer: (setf `schemata:attribute-external-name`)

- `serializer` — type: `t`; `initarg: :serializer`; `reader: schemata::attribute-serializer`; `writer: (setf schemata::attribute-serializer)`
- `unserializer` — type: (or null trivial-types:function-designator); `initarg: :unserializer`; `reader: schemata::attribute-unserializer`; `writer: (setf schemata::attribute-unserializer)`

`SCHEMATA:VALIDATION-ERROR` [Class]

Class precedence list: `validation-error`, `error`, `serious-condition`, `condition`, `t`

`SCHEMATA:ATTRIBUTE` [Class]

Class precedence list: `attribute`, `schema`, `attribute-properties`, `standard-object`, `t`

Slots:

- `name` — type: `symbol`; `initarg: :name`; `reader: schemata:attribute-name`; `writer: (setf schemata:attribute-name)`
- `type` — type: `schemata:schema`; `initarg: :type`; `reader: schemata:attribute-type`; `writer: (setf schemata:attribute-type)`
- `accessor` — type: (or null `symbol`); `initarg: :accessor`; `reader: schemata:attribute-accessor`; `writer: (setf schemata:attribute-accessor)`
- `writer` — type: (or null trivial-types:function-designator); `initarg: :writer`
- `reader` — type: (or null trivial-types:function-designator); `initarg: :reader`
- `slot` — type: (or null `symbol`); `initarg: :slot`; `reader: schemata::attribute-slot`; `writer: (setf schemata::attribute-slot)`

7 Index

(Index is nonexistent)

*

BASE64-ENCODE 4

SCHEMATA:ATTRIBUTE-ACCESSOR 9
 SCHEMATA:ATTRIBUTE-ADD-VALIDATOR 9
 SCHEMATA:ATTRIBUTE-EXTERNAL-NAME 9
 SCHEMATA:ATTRIBUTE-FORMATTER 9
 SCHEMATA:ATTRIBUTE-NAME 9
 SCHEMATA:ATTRIBUTE-OPTIONAL-P 10
 SCHEMATA:ATTRIBUTE-PARSER 9
 SCHEMATA:ATTRIBUTE-READER 10
 SCHEMATA:ATTRIBUTE-REQUIRED-P 9
 SCHEMATA:ATTRIBUTE-TYPE 9
 SCHEMATA:ATTRIBUTE-TYPE-NAME 10
 SCHEMATA:ATTRIBUTE-VALIDATOR 9
 SCHEMATA:ATTRIBUTE-WRITER 10
 SCHEMATA:DEF-SCHEMA-CLASS 9
 SCHEMATA:DEFSHEMA 9
 SCHEMATA:FIND-OBJECT-ATTRIBUTE 10
 SCHEMATA:FIND-SCHEMA 10

S

SCHEMATA:*BASE64-ENCODE* 4

SCHEMATA:GENERATE-SCHEMA-FROM-CLASS 10
 SCHEMATA:OBJECT-ATTRIBUTES 9
 SCHEMATA:OBJECT-CLASS 9
 SCHEMATA:OBJECT-NAME 9
 SCHEMATA:PARSE-WITH-SCHEMA 9
 SCHEMATA:PATCH-WITH-SCHEMA 10
 SCHEMATA:POPULATE-WITH-SCHEMA 10
 SCHEMATA:SCHEMA 9
 SCHEMATA:SCHEMA-CLASS-SCHEMA 10
 SCHEMATA:SCHEMA-DOCUMENTATION 9
 SCHEMATA:SCHEMA-GENERATOR 9
 SCHEMATA:SCHEMA-SPEC 10
 SCHEMATA:SCHEMA-TYPE 9
 SCHEMATA:SERIALIZE-WITH-SCHEMA 10
 SCHEMATA:UNSERIALIZE-WITH-SCHEMA 9
 SCHEMATA:VALIDATE-WITH-SCHEMA 10
 SCHEMATA:VALIDATION-ERROR 10