STON: an Object Notation for Pharo

Sven van Caekenberghe

March 12, 2024

Copyright 2017 by Sven van Caekenberghe.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
http://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Getting started with STON

STON (for *Smalltalk Object Notation*) is a lightweight, text-based, and human-readable data-interchange format. STON is developed by Sven Van Caekenberghe. STON can be used to serialize domain level objects, either for persistency or for network transport. As its name suggests, it is based on JSON. It adds symbols as a primitive value, and class tags for object values and references. Implementations for Pharo Smalltalk, Squeak and Gemstone Smalltalk are available.

1.1 Introduction

JSON is very simple, yet just powerful enough to represent some of the most common data structures across many different languages. JSON is very readable and relatively easy to type. If you have ever seen JSON Javascript Object Notationhttp://www.json.org, you will be instantly familiar with STON as it uses similar primitive values, with the addition of a symbol type. Some details are slightly different though.

Some of these differences are due to the fact that JSON knows only about lists and maps, which means that there is no concept of object types or classes. As a result it is not easy to encode arbitrary objects, and some of the possible solutions are quite verbose. For example, the type or class is encoded as a property and/or an indirection to encode the object's contents is added. To address this, STON extends JSON by adding a primitive value, and 'class' tags for object values and references, as we will see next.

1.2 STON Features and Limitations

STON offers three main features:

- Symbols: STON extends JSON by adding symbols as a primitive value, and class tags for object values and references. Adding a symbol (a globally unique string) primitive type is a very useful addition. This is because symbols help to represent constant values in a

  readable way that is compact and fast, and because symbols allow for simpler and more readable map keys.
- Circular structures: Allowing shared and circular object structures is also ueseful simply because these structures are widely used and because they allow for naturally efficient object graphs.
- JSON backward compatible: Additionally, the current STON implementation is backward compatible with standard JSON.

Limitations of STON are that in its current form it cannot serialize a number of objects that are more system or implementation than domain oriented, such as Blocks and classes. STON is also less efficient than a binary encoding such as Fuel.

1.3 Loading STON

A reference implementation for STON was implemented in Pharo and works in versions 1.3, 1.4, 2.0, 3.0 and 4.0. The project contains a full complement of unit tests.

STON is hosted on SmalltalkHub. To load STON, execute the following code snippet:

```
Metacello new
   baseline: 'Ston';
   repository: 'github://svenvc/ston/repository';
   load.
```

1.4 If you want to depend on it

Add the following code to your Metacello baseline or configuration

“‘ spec baseline: ’Ston’ with: [spec repository: ’github://svenvc/ston/repository’] “‘

A Gemstone (http://gemtalksystems.com/products/) port implemented by Dale Henrichs is available at https://github.com/dalehenrich/ston .

CHAPTER 2

Serializing and Materializing Objects

We now show how to serialize and materialize objects, starting with a simple rectangle and then continuing with more complex objects.

Serializing a Rectangle

To generate a STON representation for an object, STON provides two messages `toString:` and `toStringPretty:`. The first message generates a compact version and the second displays the serialized version in a more readable way. For example:

```
STON toString: (Rectangle origin: 10@10 corner: 100@50)
   --> 'Rectangle{#origin:Point[10,10],#corner:Point[100,50]}'
```

```
STON toStringPretty: (Rectangle origin: 10@10 corner: 100@50)
   -->
   'Rectangle {
      #origin : Point [ 10, 10 ],
      #corner : Point [ 100, 50 ]
   }'
```

What is shown above follows the default representation scheme for objects. Each class can define its own custom representation, as discussed in section2.11.

Materializing a Rectangle

Once you have the textual representation of an object you can obtain the encoded objects using the `STONReader` class as follows:

```
(STONReader on: ( 'Rectangle {
   #origin : Point [ -40, -15 ],
   #corner : Point [ 60, 35 ]
   }') readStream) next
   -->  (-40@ -15) corner: (60@35)
```

Alternatively, you can also use the STON facade as follows

```
(STON reader on: ( 'Rectangle {
   #origin : Point [ -40, -15 ],
   #corner : Point [ 60, 35 ]
   }') readStream) next
   -->  (-40@ -15) corner: (60@35)
```

2.1 Serialization of Maps, Lists and Class Tags

This example shows how more complex data structures are represented in STON. Maps are represented by curly braces { and }, with keys and values separated by a colon `:` and items are separated by a comma `,`. Lists are delimited by [and] and their items are separated by a comma. Class tags are represented by `ClassName [ ... ]` or `ClassName {...}`.

Next is an example of what pretty printed STON for a simple object looks like. Even without further explanation, the semantics should be clear.

```
TestDomainObject {
   #created : DateAndTime [ '2012-02-14T16:40:15+01:00' ],
   #modified : DateAndTime [ '2012-02-14T16:40:18+01:00' ],
   #integer : 39581,
   #float : 73.84789359463944,
   #description : 'This is a test',
   #color : #green,
   #tags : [
      #two,
      #beta,
      #medium
   ],
   #bytes : ByteArray [ 'afabfdf61d030f43eb67960c0ae9f39f' ],
   #boolean : false
 }
```

2.2 A Large Example: an HTTP Response

Here is a more complex example: a ZnResponse object. It is the result of serializing the result of the following HTTP request (using Zinc, see Chapters **??** and **??**). It also shows that curly braces are for dictionaries and square brackets are for lists.

```
ZnResponse {
   #headers : ZnHeaders {
      #headers : ZnMultiValueDictionary {
         'Date' : 'Sat, 21 Mar 2015 20:09:23 GMT',
         'Modification-Date' : 'Thu, 10 Feb 2011 08:32:30 GMT',
         'Content-Length' : '113',
         'Server' : 'Zinc HTTP Components 1.0',
         'Vary' : 'Accept-Encoding',
         'Connection' : 'close',
         'Content-Type' : 'text/html;charset=utf-8'
      }
   },
   #entity : ZnStringEntity {
      #contentType : ZnMimeType {
         #main : 'text',
         #sub : 'html',
         #parameters : {
            'charset' : 'utf-8'
         }
      },
      #contentLength : 113,
      #string :
'<html>\n<head><title>Small</title></head>\n<body><h1>Small</h1>
<p>This is a small HTML document</p></body>\n</html>\n',
      #encoder : ZnUTF8Encoder { }
   },
   #statusLine : ZnStatusLine {
      #version : 'HTTP/1.1',
      #code : 200,
      #reason : 'OK'
   }
}
```

Note that when encoding regular objects, STON uses Symbols as keys. For Dictionaries, you can use Symbols, Strings and Numbers as keys.

2.3 How Values are Encoded

We will now go into detail on how the notation encodes Smalltalk values. Values are either a primitive value or an object value. Note that the undefined object`nil` and a reference to an already encountered object are considered values as well.

2.4 Primitive Values

The kinds of values which are considered as primitives are numbers, strings, symbols, booleans and `nil`. We talk about each of these next, and we show an example of their encoding.

2.5 Numbers

Numbers are either integers or floats.

- Integers can be of infinite precision.
- Floats can be simple fractions or use the full scientific base 10 exponent notation.

```
(STON reader on: '123' readStream) next.
   --> 123
```

```
(STON reader on: '-10e6' readStream) next.
   --> -10000000
```

Strings

Strings are enclosed using single quotes and backslash is used as the escape character. A general Unicode escape mechanism using four hexadecimal digits can be used to encode any character. Some unreadable characters have their own escape code, like in JSON. STON conventionally encodes all non-printable non-ASCII characters.

```
(STON reader on: '''a simple string''' readStream) next.
   --> 'a simple string'
```

```
(STON reader on: '''\u00E9l\u00E8ves Fran\u00E7aises''' readStream)
    next.
   --> 'élèves Françaises'
```

```
(STON reader on: '''a newline \n and \t a tab''' readStream) next.
   -->
'a newline
 and     a tab'
```

Symbols

Symbols are preceded by a #. Symbols consisting of a limited character set (letters, numbers, a dot, underscore, dash or forward slash) are written literally. Symbols containing characters outside this limited set are encoded like strings, enclosed in single quotes.

```
(STON reader on: '#foo' readStream) next.
   --> #foo
```

```
(STON reader on: '#''Foo-bar''' readStream) next.
   --> #'Foo-bar'
```

Booleans

Booleans consist of the constants `true` and `false`.

```
(STON reader on: 'true' readStream) next.
   --> true
```

The UndefinedObject

The undefined object is represented by the constant `nil`

```
(STON reader on: 'nil' readStream) next.
   --> nil
```

2.6 Object Values

Values that are not primitives can be three kinds of objects. The first kind is a collection of values: lists or maps, the second kind is a non-collection object, and the last kind is a reference to another value.

Like in JSON, STON uses two primitive composition mechanisms: lists and maps. Lists consist of an ordered collection of arbitrary objects. Maps consist of an unordered collection of key-value pairs. Keys can be strings, symbols or numbers, and values are arbitrary objects.

2.7 Lists

Lists are delimited by `[` and `]`. Items are separated by a comma `,`.

For example the following expression is a list with two numbers -40 and -15.

```
[ -40, -15 ]
```

The serialization of an array is represented by a list.

```
STON toString: #(20 30 40)
   --> '[20,30,40]'
```

```
STON toString:  { 1. 0. -1. true. false. nil }.
   --> '[1,0,-1,true,false,nil]'
```

Lists are also used to represent values of certain object instance variables, as discussed in section 2.11.

```
STON toString: 20@30
   --> 'Point[20,30]'
```

```
STON toString: Date today
   --> 'Date[''2015-03-21'']'
```

2.8 Maps

Maps are delimited by `{` and `}`. Keys and values are separated by a colon `:` and items are separated by a comma `,`. Dictionaries are serialized as maps, for example as below:

```
STON toStringPretty: (
   Dictionary new
      at: #blue
      put: 'bluish';
      at: #green
      put: 'greenish';
      yourself)
   -->
'{
   #green : ''greenish'',
   #blue : ''bluish''
}'
```

2.9 Objects

An object in STON has a class tag and a representation. A class tag starts with an alphabetic uppercase letter and contains alphanumeric characters only. A representation is either a list or a map. The next example shows an instance of the class `ZnMimeType`:

```
ZnMimeType {
   #main : 'text',
   #sub : 'html',
   #parameters : {
      'charset' : 'utf-8'
   }
}
```

This is a generic way to encode arbitrary objects. Non-collection classes are encoded using a map of their instance variables: instance variable name (a symbol) mapped to instance variable value. Collection classes are encoded using a list of their values.

For the list like collection subclass `Array`, the class tag is optional, given a list representation. The following pairs are thus equivalent:

```
[1, 2, 3]  =  Array [1, 2, 3]
```

Also, for the map like collection subclass `Dictionary` the class tag is optional, given a map representation:

```
{#a : 1, #b : 2} = Dictionary {#a : 1, #b : 2}
```

2.10 References

To support shared objects and cycles in the object graph, STON adds the concept of references to JSON. Each object value encountered during a depth first traversal of the graph is numbered from 1 up. If a object is encountered again, only a reference to its number is recorded. References consist of the `@` sign followed by a positive integer. When the data is materialized, references are resolved after reconstructing the object graph.

Here is an `OrderedCollection` that shares a `Point` object three times:

```
| pt ar |
pt := 10@20.
ar := { pt . pt . pt }.
STON toString: ar
   --> '[Point[10,20],@2,@2]'
```

A two element `Array` that refers to itself in its second element will look like this:

```
[ #foo, @1 ]
```

Note that strings are not treated as objects and are consequently never shared.

2.11 Custom Representations of Objects

In the current reference implementation in Pharo, a number of classes received a special, custom representation, often chosen for compactness and readability. We give a list of them here and then discuss on how to implement such a custom representation.

CHAPTER 3

Default Custom Representations

Time

`Time` is represented by a one element array with an ISO style `HH:MM:SS` string

```
STON toString: Time now
   --> 'Time[''17:06:41.489009'']'
```

Date

`Date` is represented as a one element array with an ISO style `YYYYMMDD` string

```
STON toString: Date today
   -->  'Date[''2015-03-21'']'
```

Date and Time

`DateAndTime`, `TimeStamp` is represented as a one element array with an ISO style `YYYY-MM-DDTHH:MM:SS.N+TZ.TZ` string

```
STON toString: DateAndTime now
   --> 'DateAndTime[''2015-03-21T17:46:01.751981-03:00'']'
```

Point

`Point` is represented as a two element array with the x and y values

```
STON toString: 100@200
   --> 'Point[100,200]'
```

ByteArray

`ByteArray` is represented as a one element array with a hex string

```
STON toString: #( 10 20 30) asByteArray
   --> 'ByteArray[''0a141e'']'
```

Character

`Character` is represented as a one element array with a one element string

```
STON toString: $a
   --> 'Character[''a'']'
```

Associations

Associations are represented as a pair separated by `:`.

```
STON toString: (42 -> #life)
   --> '42:#life'
```

Nesting is also possible `#foo : 1 : 2` means `#foo->(1->2)`.

Note that this custom representation does not change the way maps (either for dictionaries or for arbitrary objects) work. In practice, this means that there are now two closely related expressions:

```
STON fromString: '[ #foo:1, #bar:2 ]'.
   --> { #foo->1. #bar->2 }
```

```
STON fromString: '{ #foo:1, #bar:2 }'.
   --> a Dictionary(#bar->2 #foo->1 )
```

In the first case you get an Array of explicit Associations, in the second case you get a Dictionary (which uses Associations internally).

3.1 Creating a Custom Representation

The choice of using a default STON mapping for objects or to prefer a custom representation is up to you and your application. In the generic mapping instance variable names (as symbols) and their values become keys and values in a map. This is flexible: it won't break when instance variables are added or removed. It is however more verbose and exposes all internals of an object, including ephemeral ones. Custom representations are most useful to increase the readability of small, simple objects.

The key methods are instance method `stonOn:` and class or instance method `fromSton:`. The former produces a STON representation of the object and the latter creates a new object from a STON representation. If `fromSton:` is implemented at instance side, STON will first create an instance of the object before calling `fromSton:` e.g. as in `Point`. If implemented at class side, the creation of the instance is the responsibility of the `fromSton:` method, e.g. as in `ByteArray`.

During encoding, classes can output a one-line representation of themselves by sending either the message `#writeObject:listSingleton:` or the message `#writeObject:streamShortList:` to an instance of `STONWriter`. The first argument of the message should be `self` and the second argument a single element, or a collection of elements respectively.

Examples of this are below:

```
Date>>stonOn: stonWriter
   "Use an ISO style YYYYMMDD representation"
   stonWriter writeObject: self listSingleton: self yyyymmdd
```

```
Point>>stonOn: stonWriter
   stonWriter writeObject: self streamShortList: [ :array |
      array add: x; add: y ]
```

An instance of `STONWriter` also understands the `#writeObject:streamList:` and `#writeObject:streamMap:` messages, which generate a multi-line representation. Also, classes can use another external name by overriding `Object class>>stonName`.

STON offers a way to control which instance variables get written and the order in which they get written. This can be done by overwriting `Object class>>#stonAllInstVarNames` to return an array of symbols. Each symbol is the name of a variable and the order of the symbols determines write order. Also, having `Object>>#stonShouldWriteNilInstVars` return `true` causes instance variables to be written out when they are`nil` (the default is to omit them).

Lastly, postprocessing on instance variables for resolving references is realized by the `Object>>stonProcessSubObjects:` method. If custom postprocessing is required, this method should be overwritten.

3.2 Usage

This section lists some code examples on how to use the current implementation and its API. The class `STON` acts as a class facade API to read/write to/from streams/strings while hiding the actual parser or writer classes. It is a central access point, but it is very thin: using the reader or writer directly is perfectly fine, and offers some more options as well.

3.3 Simple Reading and Writing

Parsing is the simplest operation, use either the `fromString:` or `fromStream:` method, like this:

```
STON fromString: 'Rectangle { #origin : Point [ -40, -15 ],
   #corner : Point [ 60, 35 ]}'.
```

```
'/Users/sven/Desktop/foo.ston' asReference
   fileStreamDo: [ :stream | STON fromStream: stream ].
```

Invoking the reader (parser) directly goes like this:

```
(STON reader on:
   'Rectangle{#origin:Point[0,0],#corner:Point[1440,846]}'
      readStream) next.
```

Writing has two variants: the regular compact representation or the pretty printed one. The methods to use are `toString:` and `toStringPretty:` or `put:onStream:` and `put:onStreamPretty:`, like this:

```
STON toString: World bounds.
STON toStringPretty: World bounds.
'/Users/sven/Desktop/bounds.ston' asReference
```

```
   fileStreamDo: [ :str |
      STON put: World bounds onStream: str ].
'/Users/sven/Desktop/bounds.ston' asReference
   fileStreamDo: [ :str |
      STON put: World bounds onStreamPretty: str ].
```

3.4 Supporting Comments

Like JSON, STON does not allow comments of any kind in its format. However, STON offers the possibility to handle comments using a special stream named `STONCStyleCommentsSkipStream`. The following snippets illustrate two ways to use this stream:

```
STON fromStream: (STONCStyleCommentsSkipStream on:
    'Point[/* this is X*/ 1, /* this is Y*/ 2] // Nice huh ?'
    readStream).
   --> 1@2
```

```
STON fromStringWithComments: '// Here is how you create a point:
Point[
   // this is X
   1,
   // this is Y
   2 ]
// Nice huh ?'.
   --> 1@2
```

This helper class is useable in other contexts too, like for NeoJSON. The advantage is that it does not change the STON (or JSON) syntax itself, it just adds some functionality on top.

3.5 Configuring the Writer

The writer can be created explicitly as follows:

```
String streamContents: [ :stream |
   (STON writer on: stream)  nextPut: World bounds ].
```

When created, the reference policy of the writer can be set. The default for STON is to track object references and generate references when needed. Other options are to signal an error on shared references by sending the writer `referencePolicy: #error`, or to ignore them (`referencePolicy: #ignore`) with the risk of going into an infinite loop. An example of the error reference policy is below:

```
| pt ar |
pt := 10@20.
ar := { pt . pt . pt }.
String streamContents: [ :stream |
   (STON writer on: stream)
      referencePolicy: #error;
      nextPut: ar]
   --> STONWriterError: 'Shared reference detected'
```

3.6 Compatibility with JSON

The current STON implementation has a very large degree of JSON compatibility. Valid JSON input is almost always valid STON. The only exceptions are the string delimiters (single quotes for STON, double quotes for JSON) and `nil` versus`null`. The STON parser accepts both variants for full compatibility.

The STON writer has a `jsonMode` option so that generated output conforms to standard JSON. That means the use of single quotes as string delimiters, `null` instead of `nil`, and the treatment of symbols as strings. When using JSON mode the reference policy should be set to `#error` or `#ignore` for full JSON compatibility. Also, as JSON does not understand non-primitive values outside of arrays or dictionaries, it is necessary to convert data structures to an`Array` or `Dictionary` first. Attempting to write non primitive instances that are not arrays or dictionaries will throw an error.

Next is an example of how to use the STON writer to generate JSON output.

```
| bounds json |
bounds := World bounds.
json := Dictionary
   with: #origin -> (
      Dictionary
         with: #x -> bounds origin x
         with: #y -> bounds origin y)
   with: #corner -> (
      Dictionary
         with: #x -> bounds corner x
         with: #y -> bounds corner y).
String streamContents: [ :stream |
   (STON writer on: stream)
      prettyPrint: true;
      jsonMode: true;
      referencePolicy: #error;
      nextPut: json ].
```

3.7 Handling CR, LF inside Strings

STON also supports the conversion or not of CR, LF, or CRLF characters inside strings and symbols as one chosen canonical newLine.

The message `STONReader>>convertNewLines: aBoolean` and the message `STONReader>>newLine: aCharacter` read and convert CR, LF, or CRLF inside strings and symbols as one chosen canonical newLine. When true, any newline CR, LF or CRLF read unescaped inside strings or symbols will be converted to the newline convention chosen, see `newLine:`. The default is false, not doing any convertions.

In the following example, any CR, LF or CRLF seen while reading Strings will all be converted to the same EOL, CRLF.

```
(STON reader on: ..)
   newLine: String crlf;
   convertNewLines: true;
   next.
```

The message `STONWriter>>keepNewLines: aBoolean` works as follows: If true, any newline CR, LF or CRLF inside strings or symbols will not be escaped but will instead be converted to the newline convention chosen, see `newLine:`. The default is false, where CR, LF or CRLF will be enscaped unchanged.

```
(STON writer on: ...)
   newLine: String crlf;
   keepNewLines: true;
   nextPut: ...
```

Any CR, LF or CRLF inside any String will no longer be written as `\r`, `\n` or `\r\n` but all as CRLF, a normal EOL.

3.8 About Storing Block Closures

This is a recurring question. BlockClosures are way too general and powerful to be serialised. That is why serialising BlockClosures is not supported in STON. The code inside a block can refer to and even affect state outside the block. Furthermore the return operator is quite special as it returns from some outer context.

A subset of BlockClosures are those that are clean. These do not close over other variables, nor do they contain a return. By using their source code representation, it is possible to serialise/-materialise them.

You can try this by adding the following methods:

```
BlockClosure >> stonOn: stonWriter
 self isClean
   ifTrue: [ stonWriter writeObject: self listSingleton: self
    printString ]
   ifFalse: [ stonWriter error: 'Only clean blocks can be
    serialized' ]
```

```
BlockClosure >>stonContainSubObjects
 ^ false
```

```
BlockClosure clas >> fromSton: stonReader
 ^ self compilerClass new
     source: stonReader parseListSingleton;
     evaluate
```

With these additions you can do the following:

```
STON fromString: (STON toString: [ :x :y | x + y ]).
```

Note that the actual class name depends on the Pharo version (BlockClosure in Pharo 7, Full-BlockClosure in Pharo 9 and maybe soon CleanBlockClosure - Marcus is working on that last one and that would be very cool because it would say exactly what it it).

3.9 Conclusion

STON is a practical and simple text-based object serializer based on JSON (see also Chapter **??**). We have shown how to use it, how values are encoded and how to define a custom representation for a given class.

CHAPTER 4

Appendix: BNF

```
value
   primitive-value
   object-value
   reference
   nil
primitive-value
   number
   true
   false
   symbol
   string
object-value
   object
   map
   list
object
   classname map
   classname list
reference
   @ int-index-previous-object-value
map
   {}
   { members }
members
   pair
   pair , members
pair
   string : value
   symbol : value
   number : value
```

```
list
   []
   [ elements ]
elements
   value
   value , elements
string
   ''
   ' chars '
chars
   char
   char chars
char
   any-printable-ASCII-character-except-'-"-or-\
   \'
   \"
   \\
   \/
   \b
   \f
   \n
   \r
   \t
   \u four-hex-digits
symbol
   # chars-limited
   # ' chars '
chars-limited
   char-limited
   char-limited chars-limited
char-limited
   a-z A-Z 0-9 - _ . /
classname
   uppercase-alpha-char alphanumeric-char
number
   int
   int frac
   int exp
   int frac exp
int
   digit
   digit1-9 digits
   - digit
   - digit1-9 digits
frac
   . digits
exp
   e digits
digits
   digit
```

```
    digit digits
e
    e
    e+
    e-
    E
    E+
    E-
```