Chapter 1

STON: a Smalltalk Object Notation

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

- First version: February 14, 2012
- Last update: May 5, 2012

1.1

JSON seems to have hit a sweet spot: it 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.

However, JSON knows only about lists and maps. There is no concept of object types or classes. This means that it is not easy to encode arbitrary objects, and some of the possible solutions are quite verbose (Encoding the type or class as a property and/or adding an indirection to encode the object's contents). STON extends JSON by adding a primitive value, class tags for object values and references as we will see next.

1.2 Getting started

STON is now hosted on SmalltalkHub. To load STON you can execute:

```
Gofer new
   smalltalkhubUser: 'SvenVanCaekenberghe' project: 'STON';
   package: 'ConfigurationOfSton';
   load

ConfigurationOfSton project load: #stable
```

The following repository contains the recent code:

```
MCHttpRepository
   location: 'http://smalltalkhub.com/mc/SvenVanCaekenberghe/STON/main'
   user: ''
   password: ''
```

1.3 Examples

If you have ever seen JSON http://www.json.org - Javascript Object Notation, you will be instantly familiar with Smaltalk Object Notation. It uses similar primitive values, with the addition of a symbol type. Some details are slightly different though.

For object values an optional class tag precedes a representation that is either a list of values or a map of keys and values. Exactly how each class is represented is up to the implementation, although this document suggests some conventions for well known classes.

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
```

```
  }
```

A Rectangle

Here is an example of a Rectangle and its STON representation:

```
Rectangle center: 10@10 extent: 100@50.

Rectangle {
  #origin : Point [ −40, −15 ],
  #corner : Point [ 60, 35 ]
  }
```

an HTTP Response

Here is a more complex example, the result of doing an HTTP request using the Zinc HTTP Components framework, a ZnResponse object:

Book Todo: how do we get this object from Zinc?

```
ZnEasy get: 'http://zn.stfx.eu/zn/small.html'.

ZnResponse {
   #headers : ZnHeaders {
      #headers : ZnMultiValueDictionary {
         'Date' : 'Fri, 04 May 2012 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'
    }
}
```

> **Book Todo:** it is not clear to me when we should use key as string or as symbols

As we will see { } defined dictionaries and [ ] lists.

About STON features

STON extends JSON by adding symbols as a primitive value, class tags for object values and references. Adding a symbol (globally unique string) primitive type is a very useful addition: because symbols help to represent constant values in a compact and fast yet readable way, and because symbols allow simpler and more readable map keys.

Allowing shared and circular object structures is also necessary simply because these exist in reality and because they allow for naturally efficient object graphs.

Additionally, the current STON implementation is backwards compatible with standard JSON.

1.4 Smalltalk Object Notation

Let us look now at the notation.

Values

STON encodes a value, which is either a primitive value or an object value. The undefined object nil and a reference to an already encountered object are considered values as well.

Primitive Values

The primitive types are numbers, strings, symbols and boolean.

- 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.
- Strings are enclosed using single quotes. A backslash is used as an escape mechanism. Some unreadable characters have there own escape code, like in JSON. A general Unicode escape mechanism using four hexadecimal digits is used to encode any character. STON conventionally encodes all non-printable non-ASCII characters.
- Symbols are introduced by a sharp sign. 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.
- Booleans consist of the constants true and false.

Book Todo: add examples

Object Values

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

Lists

Lists are delimited by [and]. Items are separated by comma ,. For example the following expression is a list with two numbers -40 and -15.

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

Maps

Maps are delimited by { and }. Keys and values are separated by a colon : and items are separated by a comma (,). For example the following mymetype object is a map and its parameter is also a map or dictionary.

```
ZnMimeType {
   #main : 'text',
   #sub : 'html',
   #parameters : {
```

```
    'charset' : 'utf-8'
  }
}
```

Class Tag

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 previous example show a class tag for the class ZnMimeType.

There are generic ways to encode arbitrary objects. Non-collection classes are encoded using a map of their instance variables, instance variable name symbol mapped to instance variable value. Collection classes are encoded using a list of their values. Some classes have their own custom representation, often chosen for compactness and readability.

For one selected list like collection subclass, currently Array, the class tag is optional, given a list representation.

For one selected map like collection subclass, currently Dictionary, the class tag is optional, given a map representation. The following pairs are thus equivalent:

```
[1, 2, 3] = Array [1, 2, 3]
{#a : 1, #b : 2} = Dictionary {#a : 1, #b : 2}
```

During encoding, some classes can ask for a short list representation. This will instruct the pretty printer to print their encoding on one line.

Book Todo: explain previous two lines

1.5 References

To support shared objects and cycles in the object graph, STON adds the concept of references. Each object value encountered during depth first traversal is numbered from 1 up. If a object is encountered again, only a reference to its number is recorded. References consist of the at-sign followed by a positive integer. After reconstructing an object graph, references are resolved. Here is an OrderedCollection with three times the same, shared Point object:

```
OrderedCollection [ Point [1, 2], @2, @2 ]
```

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

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

Strings are not treated as objects and are consequently never shared.

Some Conventional Representations

In the current working reference implementation in Pharo, a number of classes received a special, custom representation. These are:

- Time a one element array with an ISO style HH:MM:SS string
- Date a one element array with an ISO style YYYYMMDD string
- DateAndTime, TimeStamp a one element array with an ISO style YYYY-MM-DDTHH:MM:SS.N+TZ.TZ string
- Point a two element array with the x and y values
- ByteArray a one element array with a hex string
- Character a one element array with a one element string

Whether or not you choose to use the default STON mapping for Objects, where instance variables symbol names and their values become keys and values in a map, or whether you prefer a custom representation is up to you and your application.

The generic mapping is flexible: it won't break when instance variables are added or removed. It is 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.

1.6 Implementation

To validate the Smalltalk Object Notation a reference implementation was implemented in [Pharo Smalltalk](http://www.pharo.st). It is available on SmalltalkHub.

It works in Pharo Smalltalk versions 1.3, 1.4 and 2.0 as well as in [Squeak](http://www.squeak.org). The project contains a full complement of unit tests.

The key methods are instance method stonOn: and class or instance method fromSton:. Classes can use another external name using the class method stonName.

Additionally, you have to implement the instance method stonProcessSubObjects: to allow subobjects to be processed, to resolve forward references to real objects. Furthermore, the class method stonName allows for class name aliasing. That is, the external and internal class names do not have to be identical.

Dale Henrichs did a port to [Gemstone Smalltalk](http://www.gemstone.com/products/gemstone) that is available here: https://github.com/dalehenrich/ston. He is working on ports to Amber.

1.7 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 OK too, and offers some more options as well.

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 ].
```

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: [ :stream |
        STON put: World bounds onStream: stream ].

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

Invoking the reader (parser) directly goes like this:

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

The writer can be used in a similar way:

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

Next is an example of how to use the STON writer to generate JSON output. Note that it is necessary to convert to Arrays and/or Dictionaries first.

```
| 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 ].
```

1.8 Compatibility with JSON

The current STON implementation has a very large degree of JSON compatibility. Valid JSON input is almost valid STON. The only exceptions are the string delimiters (single quotes for STON, double quotes for JSON) and nil vs. 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, the treatment of symbols as strings.

Any non primitive instances that are not arrays or dictionaries will throw an error. Another option sets the referencePolicy. The default for STON is to track object references and generate reference when needed. Other options are to signal an error on shared references or circles, or to ignore them with the risk of going into an infinite loop.

1.9 Limitations

STON in its current form 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. As a new format, STON has yet to prove itself in practice.

1.10 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-
```

1.11 Conclusion

STON is a practical and simple text-based object serializer. It is handy to exchange information.