Selflang cheat sheet

Object syntax

An *object* consists from (possibly empty) set of *slots* and (optionally) *code*.

Slots behave as key-value lookup table, translating messages to objects. *Code* is a sequence of *expressions* (*message sends* and *literals*) separated by dots, evaluated in order.

Syntax is straightforward:

```
(| slot1. slot2 | 'code' printLine. 'str')
```

for object with *slots* and *code*, or

```
(| | 'this is str') or ('this is str')
```

for object with *code*, but no *slots*, or

```
(| slot1. slot2. slotN |)
```

for object with *slots*, but no *code*, or

```
(| |) or ()
```

for empty object.

Comments

Comments use double quotes and are ignored by parser.

```
("empty object same as () or (| |)")
```

Slot assignments

Slots can be assigned at the time of definition, either read only using = operator:

```
(| slot = 1 |)
```

Or rewritable using <- operator. This operator actually adds two slots - `slot` for the data and `slot:` with assignment method object.

```
(| slot <- 1 |)
```

Messages

Messages are evaluated from left to right. Object is on the left, messages on the right. Messages can be without arguments:

```
obj message
```

(to the *obj* send a *message*), or with arguments:

```
obj message: argument
```

(to the *obj* send a *message* with *argument* as value).

Special slots

`self` slot

Every object has automatically created slot named `self`, pointing to object itself. Unlike other languages, `self` may be omitted, in *message sends*, so

```
self message
```

is same thing as

```
message
```

Annotation slot

Additional informations may be provided in the *annotation slot*:

```
(| {} = 'Annotation string.' |)
```

or

```
(| { 'Annotation string.' slot. another. } |)
```

`parent` slots

Every object may also contain one or more parent slots, which are slots whose names end in a star (*). Lookups unresolved in the receiving object are delegated to these parent slots. This is used to implement inheritance (often with a single parent slot named `parent*`) and also mixins, and namespaces.

Method slots

Method slots is the storage for the code in objects.

Unary slots

Code *slots* without arguments are called *unary*.

```
(| first = (printLine) |)
```

This *method slot* can be invoked by sending *unary* message:

```
obj first
```

Binary slots

Slots with one argument is called *binary*:

```
(| first: = (| :arg | arg printLine) |)
```

Which has shorter equivalent:

```
(| first: arg = (arg printLine) |)
```

Invocation is possible using the *binary message*:

```
obj first: 1
```

Binary slots are also used as operators (one or more characters from `!@#$%^&*-+=~/?<>,;|'\` set).

Keyword slots

It is also possible to create *keyword* slots with multiple arguments:

```
(| first: x Second: y = (x + y printLine.) |)
```

Notice the upper-case *S* in `Second`. Invocation is done via *keyword message*:

```
obj first: 3 Second: 5
```

Priorities

Constant definition > Unary > Binary > Keyword messages. Constant = self | number | string | object.

Block objects *(closures)*

Block objects are Self closures, which means, that `self` slot is not present. Unresolved lookups are delegated via `parent*` to namespace surrounding the *block object* at the time of creation. Syntax is same as with objects, but square brackets are used:

```
[| slot1. slot2 | 'code']
```

Another difference is support of *non-local return statement* using ˆ preceding the returned value. This returns the value not just from the *closure*, but also from the enclosing namespace.

```
(|
   x = 6.
   non_zero = (x > 5 ifTrue: [ ˆ 'ok' ]
                     False: [ ˆ 'nope' ])
|)
```

ˆ doesn't just return from the closure in `[]`, but also from the surrounding *method object*. `non_zero` message thus returns `'ok'`.

Block messages

Block objects take by default message `value`, which evaluates the code and returns result of last statement.

If the *block object* takes (multiple) parameters, they may be supplied using `value: With:` message pattern:

```
block_obj value: x With: y .. With: z
```

Notice the upper-case first letters in `With:` message. This tells the Self, that it's still part of one message with multiple arguments. In case of blocks, **unrequested arguments are ignored**, so you may use unlimited number of `With:`.

Namespaces

Traits

Objects with shared behaviour, which is used by pointing `parent*` slots to them. This is analogical to *sub-classing* in other languages.

Mixins

Mixins are small, parentless bundles of behavior designed to be *mixed into* some other object. Approximate analogy to other languages would be *interface with (partial) implementation*.

Globals

Prototype objects and *oddballs* (unique singletons) like `true`, `false`, `nil`..

Resends

Resends allows you to delegate the messages to the parent branches of OOP tree. *Resend messages* are equivalent of *super* calls in other languages.

Syntactically, *resends* are implemented using `resend.` prefix for resent messages:

```
resend.unary
resend.+ 1
resend.keyword: 1 Another: 2
```

You may also use *directed resends*, targeting specific parent:

```
intParent.+ 1
otherParent.keyword: 1 Another: 2
```

Mirrors

Mirrors provide Self with introspection capabilities.

Mirror can be created by sending `reflect: x` message to any `defaultBehavior` instance. The message will return *dictionary-like mirror* object.

```
defaultBehavior reflect: 1
```

Mirror presents you with sort of introspection layer usable for structural changes in *reflectee* (original object `x`) by manipulating the *dictionary-like mirror* object.

This may be used for examination, addition or removal of the *slots* of the *reflectee*.

Collections

Various containers for data are implemented as *key: val* storages. Even lists use this convention (elements are used both as *key* and *value*).

Self offers a rich variations of *Sets*, *Dictionaries* and *Trees*:

```
traits collection
├ traits abstractSetOrDictionary
│ ├ traits abstractSet
│ │ ├ traits universalSet
│ │ │ └ universalSet
│ │ └ traits hashTableSet
│ │   ├ identitySet parent
│ │   ├ customizableSet parent
│ │   ├ reflectiveIdentitySet parent
│ │   ├ traits set
│ │   └ traits sharedSet
│ └ traits abstractDictionary
│   ├ traits orderedDictionary
│   ├ traits universalDictionary
│   │ └ universalDictionary
│   └ traits hashTableDictionary
│     ├ identityDictionary parent
│     ├ customizableDictionary parent
│     ├ reflectiveIdentityDictionary parent
│     ├ traits dictionary
│     └ traits sharedDictionary
└ traits tree
  ├ traits emptyTrees abstract
  │ ├ traits emptyTrees bag
  │ └ traits emptyTrees set
  └ traits treeNodes abstract
    ├ traits treeNodes bag
    └ traits treeNodes set
```

Sets behave like mathematical sets - unordered unique collection of values. *Dictionaries* work as *key: val* storages and are implemented using hashmaps.

Trees are different implementations of *dictionaries* using *unbalanced binary trees*.

Note: If the elements are added in sorted order, *trees* may degenerate into lists, which may result in really bad performance.

There is also variety of *Lists*, *Vectors*, *Strings* and *Queues*:

```
traits collection
├ traits path
├ traits sharedQueue
├ traits priorityQueue
├ traits list
│ ├ traits orderedSet
│ └ sortedList parent
│   └ sortedListSet parent
└ traits indexable
  └ traits mutableIndexable
    ├ traits sequence
    │ └ traits sortedSequence
    ├ traits vector
    │ └ vector
    └ traits byteVector
      ├ byteVector
      ├ traits int32or64
      │ ├ traits int64
      │ └ traits int32
      └ traits string
        ├ traits immutableString
        │ └ traits canonicalString
        └ traits mutableString
```

Collections have rich message protocol, allowing various operations. Most important are:

Message	Description
`at:`	Get item at position / key.
`at: Put:`	Put item to position / key.
`add:`	Add item (to the end in ordered) collections.
`addAll:`	Add all items to (end of) collection.
`do:`	Iterate over collections.

Collector

Collector is special kind of object created using `&` operator. *Collector* is not a collection, but can be converted to one.

Main reason to use it is the `&` operator, which may simplify the syntax required to create such collection.

```
(1 & '+' & 2) asList
```

Point

Another kind of container often used with *collections* is `point` and naturally the `rectangle` made of two points.

Control sequences

As usual in *Smalltalk-like* languages, *control sequences* are implemented using message sends combined with block.

Conditionals

If condition works by using messages defined in `boolean` (or `boolean traits`) objects:

Message	Description
`obj ifTrue: b`	Execute `b` if the *obj* is `true`.
`obj ifFalse: b`	Execute `b` if the *obj* is `false`.

`obj ifTrue: b1`
`False: b2`
	b1 is executed, when *obj* evaluates to `true`. If not, (optional) *b2* block is used.

`obj ifFalse: b1`
`True: b2`
	Opposite of previous.

Loops

Looping is implemented by sending various loop messages to *block objects*:

```
[ ... ] loop
```

Loop over the *block* indefinitely.

```
[ proceed ] whileTrue: [ ... ]
```

Loop while `proceed` is `true`.

```
[ quit ] whileFalse: [ ... ]
```

Loop while `quit` is `false`.

```
[ ... ] untilTrue: [ quit ]
```

Loop until `quit` is `true`. Loop at least once.

```
[ ... ] untilFalse: [ proceed ]
```

Loop until `proceed` is `false`. Loop at least once.

```
[| :exit | ... cond ifTrue: exit ... ] loopExit
```

Loop until the `exit` parameter is not evaluated.

```
[
   | :exit |
   ...
   cond ifTrue: [ exit value: expr ]
] loopExitValue
```

Loop until the `exit` parameter is not evaluated. Allows to return the `value` with the exit from the loop.

integerIteration loops

There is also loops defined as message sends to integers:

```
numObj do: block
```

Do the `block` *numObj* times.

```
numObj to: end Do: block
```

Do the `block` each time counting from *numObj* to `end`. For example `5 to: 8 Do: [| :i | i print]` will print `5678`.

Following messages are all variations of this message:

```
numObj to: By: Do:
numObj to: ByNegative: Do:
numObj to: ByPositive: Do:
numObj upTo: Do:
numObj upTo: By: Do:
numObj downTo: Do:
numObj downTo: By: Do:
```

Useful bits

copy message

Often, you will need to create new instance of some object. What is in other languages implemented by calling `new`, or other processes calling the object builders is in prototype languages usually done by copying the object into new instance.

In Self, you may do this simply by sending the `copy` message:

```
obj copy
```

Interpreter parameters

When you run the Self interpreter on the *commandline*, it expects that you specify the path to the *World snapshot*. That may be supplied using `-s` parameter.

Self *worlds* are images for the virtual machine. You may use the official, but there is nothing that prevents you to create your own, with or without Graphical User Interface.

Self's REPL

When you run your world, it is easy to forget, that Self's interpreter also provides you with Read Eval Print Loop in the terminal. This allows you to send messages like if you were in the Shell in GUI. For example, if no windows opens, or the *desktop* crashes, you may open it by sending:

```
desktop open
```

`CTRL+c` will bring the scheduler, where you may kill processes.

You may also debug processes using `debugger attach: N`.

Additional informations

Download

Newest Self may be downloaded from the official web:

- `http://selflanguage.org`

Source code may be obtained from the GitHub:

- `https://github.com/russellallen/self`

Manual and tutorial

There is good manual called Self handbook:

- Self Handbook (online)
- Self Handbook (pdf)
- Self Handbook (ePub)

This *Cheat sheet* was created with high inspiration from this book.

If one of the links doesn't work, you may always build it for yourself from sources (`/docs/handbook`) using Sphinx (`make html` / `pdf` / `epub`).

There is also slightly outdated Self tutorial (`/docs/tutorial`) called Prototype-Based Application Construction Using SELF 4.0.

Other writings

There is blog:

- `http://blog.selflanguage.org`

and also a lot of academic papers:

- `http://bibliography.selflanguage.org`

Community

There is not-quite dead IRC channel `#self-lang` at `irc://freenode.org:6667`.

There is also forum at:

- `http://forum.selflanguage.org/`

If the forum doesn't work, don't worry, it is just frontend for mail conference self-interest-subscribe@yahoogroups.com.

Disclaimer

I did this as my notes while learning Self. That means, that there will be typos and bugs. If you find any, please send pull request.

Self may seem dead, but it is not. There is not much active people, but that only means, that you can make difference.