API reference for **`Offsider-0.9.1`**

The framework

overview

The `Offsider` framework is implemented as a collection of executables, which together provide a novel and powerful paradigm for building systems, from simple to extremely complex.

The framework allows the programmer to build systems which respond as though they were objects. Such a system is called an **offsider**. In fact, an offsider **is** an object, and you interact with it by sending it messages.

The framework contains a single high-level executable, `offsider`, which provides the mechanism for sending messages to offsiders. (It can even *send* a message without an offsider being there to receive the message. This is how offsiders are created in the first place.)

By default, a newly created offsider will respond to a large number of methods. These are all of the methods that are provided by the framework itself. It is these methods that are documented below.

Once an offsider has been created, the programmer is free to override the methods that it responds to, or create new methods, without modifying any of the methods provided by the framework.

To understand the offsider framework, you need to understand the following concepts:

The base directory

Each offsider is fully contained within a single directory within the file system. This directory is known as the *base directory*. This directory specifies the offsider completely and uniquely. Without it, the framework does not know where to send the message.

The infrastructure of an offsider

When we talk of an offsider's *infrastructure*, we are simply referring to the contents of the base directory. The infrastructure completely determines the functionality, and the state, of that offsider. (Ignoring, of course, the default functionality provided by the framework).

A named executable

A named executable is a convenience provided for the programmer. It is just a wrapper for the `offsider` executable. It simplifies the task of sending a message to a particular offsider by providing a unique executable that will send messages to that offsider, and no other.

For example, if we have an offsider with a base directory `/path/to/Foo`, and it has a named executable `~/bin/foo`, then the following are equivalent:

```
    offsider -b /path/to/Foo message
```

```
foo message
```

The named executable is easier to type, and easier to remember.

Normally the named executable is specified and created at the same time that the offsider is created.

For example:

```
offsider create foo /path/to/Foo
```

Once you have a named executable, you don't ever have to know or care where the base directory is. A common way to create a new offsider is to specify only the named executable, and let the framework come up with the base directory:

```
offsider create foo
```

Next, we describe the executable `offsider`, and its alias `this`.

offsider

`offsider` will send a message to an offsider.

The syntax is:

```
offsider [ -b baseDirectory ] [ message ]
```

where

`baseDirectory` is the base directory of the offsider to receive the message.

If the *baseDirectory* is not specified, then `offsider` will look to see if the environment variable `$BASEDIRECTORY` has been set. If it has, then it will get the base directory from there.

If no base directory is specified at all (using either of these methods), then the message will still be sent, but it will have no target. Whether this results in an error is completely up to the method being envoked. For example, the method `create` does not require an offsider to receive the message, and will not generate an error. Many other methods do require an offsider to receive the message, and will generate an error.

`offsider` also has some debugging modes that allow you to see what methods are being called, and also to profile the execution to see where the time is being spent. Refer to the man page for more detail (`man offsider`).

this

`this` is just an alternative name for `offsider`. Nothing in the framework distinguishes between these two names. It is just that `this` makes more sense when used within a method definition.

Typical usage is:

```
this message
```

which is equivalent to:

```
offsider -b $BASEDIRECTORY message
```

provided `$BASEDIRECTORY` is set. Within a method definition, it is always safe to assume that `$BASEDIRECTORY` is set, since the framework ensures that this has been done.

Usage

This API documentation documents methods. The method is invoked by sending a message to an offsider.

For illustration purposes, suppose we have an offsider with base directory `/path/to/Foo`, and named executable `foo`. We want to envoke a method `baz` with arguments `a b c`.

The message to be sent to the offsider is:

```
baz a b c
```

From the command line, we can send the message using:

```
offsider -b /path/to/Foo baz a b c
```

or

```
foo baz a b c
```

From within a method envoked on this offsider, we can use:

```
this baz a b c
```

In the documentation that follows, we would document the syntax for this method as follows:

syntax:

```
baz arguments
```

Is an offsider passive or active?

Throughout this documentation, I will talk about offsiders as though they were dynamic, active entities. So, for example I will say things like *you send a message* ***to*** *an offsider* and *the offsider* ***responds*** *to the message*.

This is an easy way to conceptualise what is happening at the high level, but it is really a false picture of what is happening behind the scenes.

In fact, the offsider is a purely passive thing. It is nothing more than a repository for data and executable code. The active component is actually the offsider framework, and in particular the `offsider` executable.

The `offsider` executable is what takes the message, parses it and takes appropriate action. It does this *on behalf of* the offsider, by *referring to* the offsider, by taking note of what executables and data are *contained within* the offsider. It delegates as appropriate to executables that implement the required functionality (methods) on behalf of the offsider.

Keep this in mind as you read through the documentation.

Some wrappers that you might find useful:

`here`

`here` will send a message to the current working directory, as though it were an offsider.

Syntax:

```
here message
```

If the current working directory contains a subdirectory `Here/`, the message will be sent to that directory instead.

my

`my` will send a message to the user's home directory, as though it were an offsider.

Syntax:

```
my message
```

If the user's home directory contains a subdirectory `My/`, the message will be sent to that directory instead.

Types in the Offsider framework

Types were introduced in version 0.9.0 . Before that, the various *types* (**keys**, **methods**, **attachments** etc) were handled each in their own particular way by methods written specifically for that type.

The **types** framework is a set of methods that can be used with any type, including types created by the user.

The advantages of this new approach is that we now have a consistent generic API that works exactly the same across all types. In addition, the user can easily add new types as required.

In addition, a couple of wrapper methods (`in` and `to`), plus a new set of syntactic sugar, provides different ways of coding the actions on type members. This means you can choose the syntax that seems most intuitive and readable for any given context.

Concepts

type

A **type** is a collection of data or executables stored within the offsider infrastructure. Each type is a separate collection.

Implementation detail: The type is implemented as a subdirectory of the offsider's base directory. The Type framework also provides mechanisms for placing directories and offsiders into such a collection, so the Type framework is in effect just an abstraction of normal file and directory operations.

Note especially that, unlike most application programming languages, the types are specific to each offsider. They are not defined at the framework level. (Although there are conventions that are used by the default framework. In particular `methods` and to a lesser extent `keys` play an important role in the logic of the Offsider framework.)

full type name

The **full name of the type** is the name of the directory that holds the collection.

For example `keys` is the full name of the directory that holds all the offsider's keys.

abbreviated type name

You can often refer to a collection by providing **an abbreviated name**, for example `key` and `k` can usually be used as names for the type with full name `keys`.

The name can be used provided it provides no ambiguity. In otherwords if you have types `foobah` and `foozball`, then `foob` is valid for `foobah` and `fooz` is valid for `foozball`, but `foo` is not valid as a type name.

member

A type is a collection, a **member** is an element of such a collection.

A member can be ordinary data, an executable, or another collection, such as a directory or an offsider.

Implementation detail: The type is implemented as a directory, and each member is just a file or directory in that directory.

action

An **action** is something that is done to a member. For example `get` the value, or `rename` it.

Differences with earlier versions. Pitfalls

Deprecated methods

All of the existing methods (eg `getKey`, `pipeToAttachment`, `methods`) are still available, but they are deprecated. They will almost certainly not be available in version 1.0.

New syntactic sugar

Some methods are now replaced by an equivalent form of syntactic sugar. For example, the message `keys` used to be implemented as a method. The same message is now handled by syntactic sugar, which converts the message to `listType keys`. The end result is the same.

The need to create types explicitly

Previously, methods like `attachment`, `addChildDictionary` and so on, would automatically create the appropriate subdirectory if it did not already exist. This is no longer the case. You will now need to explicitly create the appropriate type, like so:

```
this addType attachments
this edit attachment foo
```

It is possible that the standard directories `attachments` and `offsiders` (previously named `subdictionaries`) will be created automatically when the offsider is first created. This will be finalised before version 0.10.0 .

Atomic updates

`append` and so on do not do atomic updates (unlike their deprecated equivalents). If you want an atomic update, you will need to create a temporary member, and then rename it once it is complete.

`has` doesn't recognise the same typename abbreviations as the other actions

`has` existed in earlier versions, and has not been modified. It does not (yet) use `isType` to work out the full name of the type. It simply tries adding a terminal `s`. This will be fixed.

Methods

addType

Create a new type.

syntax:

```
addType fullType
```

where *`fullType`* is the full name of the type.

forEach

Perform an action on each member of a type.

syntax:

```
forEach type action arguments
```

where *`type`* is the name of the type (possibly contracted), *`action`* is an action to perform on each member and *`arguments`* are the arguments to that action.

in

Perform a specified action to a type member.

syntax:

```
in type action member arguments
```

where *`type`* is the (abbreviated) name of the type, *`action`* is the specified action, *`member`* is the name of the member and *`arguments`* are the arguments for the action.

NOTE: This method is merely a wrapper for the various *`action`* methods. It is provided as an alternative syntax.

isAction

Determine if an action is recognised as a valid action.

syntax:

```
isAction action
```

Returns the name of the action if it is a valid action.

isMember

Look for a member amongst all types in this offsider.

syntax:

```
isMember member
```

Returns a list of all types that contain the named member.

isType

Determine if a type exists for this offsider.

syntax:

```
    isType type
```

where `type` is an abbreviated name for a type.

Returns the full name of the type if it matches uniquely.

listType

List all members of a type.

syntax:

```
    listType fullType
```

where `fullType` is the full name of the type.

pathTo

Return the path to a given type or member.

Syntax:

```
    pathTo type [ member ]
```

`type` is the abbreviated name of the type, and `member` is the name of a member

If `member` is not given, returns the filepath to the type otherwise returns the filepath to the member.

Unlike `has`, this will only check for the existence of type, not the existence of member.

removeType

Remove a type from this offsider. Also removes all the members for that type.

syntax:

```
    removeType type
```

where `type` is the full name of the type.

renameType

Rename a type for this offsider.

syntax:

```
    renameType name newName
```

where `name` is the full name of the type and `newName` is the full new name of the type.

Does nothing if the named type does not exist.

It is an error if there is already a type called *`newName`*

to

Perform a specified action to a type member.

Syntax:

```
to fullType/member action arguments
```

or

```
to type member action arguments
```

where *`fullType`* is the full name of the type, *`type`* is the (contracted) name of the type, *`member`* is the name of the member, *`action`* is the specified action, *`arguments`* are the arguments for the action.

NOTE: This method is merely a wrapper for the various *`action`* methods. It is provided as an alternative syntax.

typeGroup

Set or get the group of a type.

Syntax:

```
typeGroup fullType [ owner ]
```

Where *`fullType`* is the full name of the type, and *`group`* is a `UNIX` group who will own the type.

If *`group`* is not given, return the current group, else set the group for the type's directory.

typeOwner

Set or get the owner of a type.

Syntax:

```
typeOwner fullType [ owner ]
```

Where *`fullType`* is the full name of the type, and *`owner`* is a UNIX user who will own the type.

If *`owner`* is not given, return the current owner, else set the owner.

typePermissions

Get or set the permission for a type.

Syntax:

```
typePermissions fullType [ permissions ]
```

Where *`fullType`* is the full name of the type, and *`permissions`* is a permission setting, as understood by `UNIX chmod`.

If *`permissions`* is given, set the permissions of the type directory, otherwise returns the permissions, as reported by `UNIX ls -l`.

types

List all types for this offsider.

Syntax:

```
types
```

typeSummary

Output a summary of the specified type.

Syntax:

```
typeSummary type [ contents | contents+ ]
```

`type` is the fullname of the type. If not given, the arguments default to

```
keys contents
```

If the keyword `contents` is given, then the contents of each member is displayed after the member name, like so:

```
membername: contents
```

If the keyword `contents+` is given, then the contents of each member is displayed, and for each line after the first, the format

```
membername:+ line of content
```

is used.

If neither keyword is given, just the member names are displayed.

Actions

Each of the methods in this section implements an *action* which can be performed on a *member* of a *type*

addDirectory

Create a directory within a type.

Syntax:

```
addDirectory type name
```

where *`type`* is the (contracted) name of the type and *`name`* is the name of the new directory

addTypeOffsider

Create an offsider within a type.

Syntax:

```
addTypeOffsider type name [ template ]
```

where *`type`* is the (contracted) name of the type *`member`* is the name of the member

`template` is an existing offsider, (either the named executable or the base directory) If specified, the new offsider will be a clone of *`template`*.

append

Append to the value of type member from stdin or from a file.

Syntax:

```
append type member [ file ]
```

where `type` is the (contracted) name of the type, `member` is the name of the member and `file` is the name of a file

If `file` is not given, will append from `stdin`.

If the member doesn't already exist, it will be created.

copy

Set the value of type member from the contents of a file.

Syntax:

```
copy type member file
```

where `type` is the (contracted) name of the type, `member` is the name of the member and `file` is the name of a file.

If the member doesn't already exist, it will be created.

USAGE NOTE: The positions of the membername and filename may conflict with normal usage with existing `Unix cp`, and other copy paradigms, **especially** when you consider the various syntactic possiblities provided by the Offsider type framework. Just keep in mind that it is entirely consistent with all of the rest of the Offsider Type framework.

edit

Edit a type member using a text editor.

Syntax:

```
edit type member [ editor ]
```

where `type` is the (contracted) name of the type, `member` is the name of the member to edit and `editor` is the editor to use.

By default, the editor is `vi`, or the value of the `$EDITOR` environment variable.

If the member doesn't already exist, it will probably be created by the editor.

get

Get the value of type member.

Syntax:

```
get type member
```

where `type` is the (contracted) name of the type and `member` is the name of the member.

If the member doesn't exist, a null string will be returned.

group

Get or set the group for a member of a type.

Syntax:

```
group type member [ group ]
```

where `type` is the (contracted) name of the type, `member` is the name of the member and `group` is a `UNIX` user, who is to own the member.

If `group` is given, then sets the group for the member, otherwise, returns the current group

owner

Get or set the owner for a member of a type.

Syntax:

```
owner type member [ owner ]
```

where `type` is the (contracted) name of the type, `member` is the name of the member and `owner` is a `UNIX` user, who is to own the member.

If `owner` is given, then sets the owner for the member, otherwise, returns the current owner

permissions

Get or set the permissions for a member of a type.

Syntax:

```
permissions type member [ permission ]
```

where `type` is the (contracted) name of the type, `member` is the name of the member and `permission` is a permission specification, as understood by `UNIX chmod`.

If `permission` is given, then sets the permission for the member, otherwise, if the member exists, returns the permissions as per `UNIX ls -l`.

pipe

Set the value of type member from `stdin`

Syntax:

```
pipe type member
```

where `type` is the (contracted) name of the type and `member` is the name of the member.

If the member doesn't already exist, it will be created.

prepend

Prepend to the value of type member from stdin or from a file.

Syntax:

```
prepend type member [ file ]
```

where *`type`* is the (contracted) name of the type, *`member`* is the name of the member and *`file`* is the name of a file.

If *`file`* is not given, will prepend from `stdin`.

If the member doesn't already exist, it will be created.

remove

Remove a member.

Syntax:

```
    remove type member
```

where *`type`* is the (contracted) name of the type and *`member`* is the current name of the member.

rename

Rename a member.

Syntax:

```
    rename type member newName
```

where *`type`* is the (contracted) name of the type *`member`* is the current name of the member *`newName`* is the new name for the member

WARNING: Will over-write any existing member with *`newName`*

run

Execute a type member, if possible.

Syntax:

```
    run type member arguments
```

where *`type`* is the (contracted) name of the type, *`member`* is the name of the member and *`arguments`* are the arguments for the executable.

If the member doesn't exist, or is not executable, nothing happens.

send

Send a message to a type member.

syntax:

```
    send type member message
```

where *`type`* is the (contracted) name of the type, *`member`* is the name of the member and *`message`* is the message to send, and may contain blanks.

Will not send the message if the member is not a directory.

set

Set the value of type member.

Syntax:

```
set type member value
```

where *type* is the (contracted) name of the type, *member* is the name of the member and *value* is the value to set, and may contain blanks.

If the member doesn't already exist, it will be created.

setAppend

Append to the value of type member from the command line arguments.

syntax:

```
setAppend type member value
```

Where *type* is the (contracted) name of the type, *member* is the name of the member, and *value* is the value to append to the current value. may contain blanks.

If the member doesn't already exist, it will be created.

x

Set the permission for a member to make it executable.

syntax:

```
x type member
```

where *type* is the (contracted) name of the type *member* is the name of the member

The permission used is `a+x`, as understood by `UNIX chmod`.

xcopy

Set the value of type member from the contents of a file. Make the resulting file executable.

Syntax:

```
xcopy type member file
```

where *type* is the (contracted) name of the type, *member* is the name of the member and *file* is the name of a file.

If the member doesn't already exist, it will be created.

USAGE NOTE: The positions of the membername and filename may conflict with normal usage with existing `Unix cp`, and other copy paradigms, **especially** when you consider the various syntactic possiblities provided by the Offsider type framework. Just keep in mind that it is entirely consistent with all of the rest of the Offsider Type framework.

xedit

Edit a type member using a text editor. Make the resulting file executable.

Syntax:

```
xedit type member [ editor ]
```

where `type` is the (contracted) name of the type, `member` is the name of the member to edit and `editor` is the editor to use.

By default, the editor is `vi`, or the value of the `$EDITOR` environment variable.

If the member doesn't already exist, it will probably be created by the editor.

xpipe

Set the value of type member from `stdin`. Make the resulting file executable.

Syntax:

```
xpipe type member
```

where `type` is the (contracted) name of the type and `member` is the name of the member

If the member doesn't already exist, it will be created.

Syntactic sugar for the Types framework

A variety of syntactic sugar has been implemented to go with new Type framework.

This provides a number of alternative ways to express the fact that you want to perform an *action* on a *member* of a *type*.

The methods documented in this section implement the syntactic sugar.

Each has the syntax:

```
syntacticSugarMethod message
```

where `message` is the incoming message, which is being parsed for syntactic sugar.

Each method will output a string in the form:

```
methodName arguments
```

provided the incoming message is recognised by `syntacticSugarMethod`.

Normally, you would not use any of these methods explicitly. They are all called by the `parseMessage` method, which is envoked by the `offsider` executable.

getSetSugar

Syntactic sugar for keys and attachments only

Message:

To return the value of a key or attachment:

```
member
```

To set the value of a key only:

```
member: value
```

Returns:

```
get keys member
```

```
    get attachments member
    set keys member value
```

.. as appropriate

NOTE: will probably assume the member is a key, even if the key doesn't exist.

listTypeSugar

Syntactic sugar for `listType`

Message:

```
    fullType
```

where *`fullType`* is the full name of the type.

returns

```
    listType fullType
```

memberActionSugar

Syntactic sugar for a member and action

Message:

```
    member action args
```

where *`member`* is the name of a member, *`action`* is the action to perform and *`args`* are the arguments for that action. The type is not specified.

If any type contains a member with name *`member`*, then the type will be set to be the **first** type that it is found in. **So be careful!**

returns

```
    action fullType member args
```

offsiderSugar

Syntactic sugar for a member name, where the member is a directory.

Message:

```
    member message
```

where *member* is the name of a member, which is a directory. The type is not specified.

If any type contains a member with name *`member`*, and the member is a directory, then the type will be set to be the **first** type that it is found in. **So be careful!**

returns

```
    send fullType member message
```

typeSugar

Syntactic sugar for a type.

Message:

```
type member action arguments
```

where `type` is the (contracted) name of a type, `member` is the name of a member in that type, `action` is the action to perform on that member and `arguments` are the arguments for that action.

returns

```
action fullType member arguments
```

Keys

A **key** is data that can be associated with an offsider. It is one way of specifying an offsider's *state*.

Each key has a name, and each key contains a value.

There is specific syntactic sugar to make it easier to get and set the value for a key:

Get the value for a key:

```
this keyName
```

Set the value for a key:

```
this keyName: key value
```

Almost identical to the concept of a key is the concept of an *attachment*. See the section on attachments for more detail.

In practice you would use a key for storing data that consists of one or two lines of printable text, and an attachment for anything else.

All of the methods documented here for Keys are deprecated as of version 0.9.0. Use methods described in the section on Types instead.

appendToKey

Append data to a key using the contents of a named file, or `stdin`.

Syntax:

```
appendToKey key filename
```

or

```
stream | appendToKey key
```

where `key` is the name of the key.

`filename` is the file from which the extra data is to be copied.

`stream` represents a stream, for example `cat file`, or some other process that can generate data for the key.

editKey

Edit a key using a text editor.

Syntax:

```
editKey name [ editor ]
```

where *name* is the name of the key to edit, and *editor* is the editor to use.

By default, the editor is `vi`, or the value of the `$EDITOR` environment variable.

If the key doesn't already exist, it will be created.

getKey

Return the value for a key.

If the key doesn't exist, return an empty string.

Syntax:

```
getKey key
```

hasKey

Determine whether the offsider has a specified key.

Syntax:

```
hasKey key
```

If the key exists, return the full path to the key. Otherwise, return an empty string, and raise an error condition.

keyPath

Return the path that a key would have whether or not it actually exists.

syntax:

```
keyPath key
```

where *key* is the name of the key.

keys

Return a list of all the keys in the offsider.

Syntax:

```
keys
```

pairs

Return a list of all the keys in the offsider, together with their values.

For each key, returns

```
keyName: keyValue
```

Notice that this output is the same format as the syntactic sugar for setting the value of a key.

pipeKey

Set the value for a key from `stdin`.

If the key doesn't exist, create it.

syntax:

```
    stream | pipeKey key
```

where *`key`* is the name of the key, and

`stream` is a process that generates the value for the key.

removeKey

Remove a key.

syntax:

```
    removeKey key
```

If the key does not exist, then an error condition is raised.

setKey

Set the value for a key.

If the key doesn't exist, creates it.

syntax:

```
    setKey key value
```

where *`key`* is the name of the key.

`value` is the value for the key and can contain spaces.

Methods

A method is executable code that an offsider is capable of executing. You send a message to the offsider, the offsider responds by executing a method.

The methods documented in this section relate to listing and modifying the methods for a particular offsider.

Clarification:

An offsider can execute methods that belong specifically to itself, and others that are external to it, but which it can access. Unless otherwise specified, all of the methods documented in this section work on methods that are owned specifically by the offsider.

Note:

Many of these methods will return paths to executables. This is dependent on the specific implementation of the offsider framework, and is not something that would neccessarily be available in an alternative implementation. Therefore, those methods should be regarded as low-level tools that effectively break the information-hiding principle of object-oriented programming.

The offsider framework will maintain the current implementation at least until Version 2.0.

All of the methods documented here for Methods are deprecated as of version 0.9.0. Use methods described in the section on Types instead.

allMethods

List all methods that this offsider responds to

usage:

```
allMethods [ fullpath ]
```

If the keyword `fullpath` is used, then the full path to each method's executable is returned, otherwise just the method names are returned.

Hint: You can pipe the result into `column` to get the list formatted into columns so it is easier to view on a terminal.

catMethod

Return the contents of an offsider's method.

syntax:

```
catMethod name
```

Where *`name`* is the name of the method.

Will only work if the method belongs specifically to the offsider. Use `hasMethod` to determine this.

Very useful for viewing the content of methods that are implemented as a script.

editMethod

Edit a method using a text editor.

Syntax:

```
editMethod name [ editor ]
```

here *`name`* is the name of the method to edit, and *`editor`* is the editor to use.

By default, the editor is `vi`, or the value of the `$EDITOR` environment variable.

If the method doesn't already exist, it will be created.

`editMethod` is useful for editing methods that are implemented as scripts.

externalMethods

Provide a list of all methods that the offsider understands, which are not specifically owned by the offsider.

syntax:

```
externalMethods [ fullPath ]
```

If the keyword `fullpath` is used, then the full path to each method's executable is returned, otherwise just the method names are returned.

hasMethod

Determine if the offsider *contains* a specific method. (This is not the same as determining whether an offsider *recognises* a specific method - see `isMethod` in the section on *messages*)

Syntax:

```
hasMethod method
```

where *`method`* is the name of the method.

If the offsider contains the method, then returns the full path to the executable. Otherwise, returns an empty string.

method

Create or modify an offsider method by copying a file, or `stdin`.

Syntax:

```
method name file
```

or

```
stream | method name
```

where *`name`* is the name of the method, and *`filename`* is the file from which the method executable is to be copied.

`stream` represents a stream, for example `cat` *`file`*, or some other process that can generate the contents of an executable.

The resulting method will be made executable using `chmod`.

methodPath

Print the paths to the offsider's methods, or a given method.

Syntax:

```
methodPath [ name ]
```

where *`name`* is the name of a method, and is optional.

If *`name`* is given, will return the full path to that executable. Otherwise will return the path to the directory that contains all the offsider's executable methods.

NOTE: Will return a path even if the named method does not exist.

methods

Return a list of all the methods in the offsider.

usage:

```
    methods [ fullpath ]
```

If the keyword `fullpath` is used, then the full path to each method's executable is returned, otherwise just the method names are returned.

removeMethod

Remove an offsider's method.

Syntax:

```
    method name
```

where *`name`* is the name of the method to remove.

Attachments

An **attachment** is data that can be associated with an offsider. Functionally, an attachment is entirely equivalent to a **key**. The only difference is that typically the values of keys are given when producing a summary of an offsider (for example, using `asText`), whereas for an attachment, only the name is given.

Therefor you can use keys and attachments more or less interchangably if you don't care about presentation.

In practice you would use a key for storing data that consists of one or two lines of printable text, and an attachment for anything else.

All of the methods documented here for Attachments are deprecated as of version 0.9.0. Use methods described in the section on Types instead.

appendToAttachment

append data to an attachment using the contents of a named file, or `stdin`.

syntax:

```
    appendToAttachment name filename
```

or:

```
    stream | appendToAttachment name
```

attachment

Create or overwrite an attachment, using the contents of a named file, or `stdin`.

syntax:

```
    attachment name filename
```

or

```
    stream | attachment name
```

Creates the attachment atomically, meaning that there is no possibility of accessing a partially written attachment.

attachments

List the names of all attachments for this offsider

syntax:

```
attachments
```

editAttachment

Edit an attachment using a text editor

syntax:

<pre>editAttachment <i>name</i> [<i>editor</i>]</pre>

By default, the editor is `vi`, or the value of `$EDITOR`.

getAttachment

Get the contents of an attachment

syntax:

<pre>getAttachment <i>name</i></pre>

hasAttachment

Determine whether an attachment exists

Syntax:

<pre>hasAttachment <i>name</i></pre>

Returns the name of the attachment if it exists, otherwise, returns a null string and an error condition

removeAttachment

Remove an attachment.

syntax:

<pre>removeAttachment <i>name</i></pre>

Creation

This section documents methods that are used to create an offsider.

Warning: It is likely that there will be changes to this API following a review prior to version 1.0

clone

Create a new offsider by copying everything from this one.

The new offsider is a complete deep copy of the current one, although the metadata in the var/ subdirectory is generated from scratch.

Syntax:

```
clone [ name [ baseDirectory ] ]
```

where *`name`* is the name for the named executable, and *`baseDirectory`* is the base directory for the new offsider.

To specify the base directory, where no named executable is to be created, use:

```
clone --noname baseDirectory
```

If no base directory is given, one is generated.

If either the named executable or the base directory already exists, then an error is generated and nothing is created.

Returns the base directory of the newly created offsider.

create

Create a new offsider, and optionally a named executable.

Syntax:

```
create [ name [ baseDirectory ] ]
```

To specify the base directory, where no named executable is to be created, use:

```
create --noname baseDirectory
```

If *`name`* is specified, create a named executable with the given name, otherwise do not create a named executable. If the name contains a path, then create the named executable at that location, otherwise use a standard location (typically `~/bin`).

If *`baseDirectory`* is specified, then build the offsider infrastructure there, otherwise create the offsider at a standard location (typically under `~/.offsiders`).

If either the base directory, or the named executable already exists, does nothing and generates an error.

Returns the base directory of the newly created offsider.

createStandard

Create a new offsider in a standard location. Also, create a named executable at a standard location.

syntax:

```
createStandard name
```

If you are `root`, the base directory is `/usr/local/share/Offsider/offsiders/`*`name`* otherwise it is `~/.offsiders/`*`name`*

If you are `root`, the named executable is `/usr/local/bin/`*`name`* else, it is `~/bin/`*`name`*

If either the base directory, or the named executable already exists, does nothing and generates an error.

Returns the base directory

destroyCompletely

Remove all trace of an offsider, including the named executable.

Syntax:

```
destroyCompletely
```

This method will definitely remove the base directory and all the infrastructure contained therein. It will remove the named executable if it was created at the same time as the offsider, and has not been moved to another location since then. If you created extra named executables manually, it will probably not be aware of them.

makeExecutable

Create the contents of an executable which can send messages to an offsider.

The executable is a script, and the source code is written to `stdout`.

Syntax:

```
makeExecutable [ baseDirectory ]
```

`baseDirectory` is the base directory for an offsider. The executable will send messages to the offsider at that base directory.

If *`baseDirectory`* is not given, it defaults to the base directory for this offsider.

Usage:

```
this makeExecutable [ baseDirectory ] > scriptName
chmod +x scriptName
```

makeExecutableForChild

Create the contents of an executable which can send messages to a child offsider.

The executable is a script, and the source code is written to `stdout`.

Syntax:

```
makeExecutableForChild relativePath
```

`relativePath` is a relative path from this offsider's base directory to the base directory for the child offsider. The executable will send messages to the offsider at that base directory.

Example usage:

We have an offsider `foo` at `/path/to/Foo`. We also have an offsider at `/path/to/Foo/childDictionaries/Baz`. We want to create an executable for the child offsider.

Create the named executable:

```
foo makeExecutableForChild childDictionaries/Baz > ~/bin/baz
chmod +x ~/bin/baz
```

Envoke a method `bah` on `baz`

```
baz bah
```

From within the method `bah`, send a message to `foo` (the parent of `baz`):

```
this parentDictionary message
```

Messages

All interaction with an offsider is by means of **messages**. A message is a string of text which is sent to the offsider. This is normally done by providing the message as the arguments to an executable, for example

```
offsider -b baseDirectory message
this message
namedExecutable message
```

In every case, the message is parsed by the offsider in order to determine which ***method*** needs to be run, and what arguments need to be sent to that method.

All of the methods documented in this section have something to do with the process of parsing a message and determining what action to take.

Remember, it is the object itself that parses the message, not some language engine.

[**Version 0.9**] Introduced a comprehensive set of new syntactic sugar to handle the new Types framework. Those methods, and the syntactic sugar they implement, are documented in the section on **Types**.

customSugar

This method is not implemented in the framework. However, if you implement this method for an offsider, then `parseMessage` will run this method **after** other checks for syntactic sugar.

The syntax must be:

```
customSugar message
```

The method must return a string in the form:

```
knownMethod arguments
```

where *`knownMethod`* is the name of a method known to that offsider, and *`arguments`* are the arguments to be passed to that method.

If the message is not recognised by `overrideSugar` then it must return an empty string, so that `offsider` can continue its processing.

WARNING: If you implement this method for an offsider:

Only use syntax like `this` *`knownMethod arguments`*, rather than the more general `this` *`message`*, to avoid infinite recursion. Be even more careful if `isMethod` has been over-ridden.

isMethod

Determine whether this offsider recognises the named method.

If so, return the full path to the executable that implements the method. Otherwise, an empty string is returned.

syntax:

```
isMethod methodName
```

By default, looks for executables in the following order:

`baseDirectory/methods/methodName`

`offsider.methodName` (as per the `$PATH` environment variable)

`Dictionary.methodName` (**Deprecated**. Will be removed in version 1.0)

This method can be overridden for any offsider, to change the way in which methods are found.

keySugar

This method is deprecated as of version 0.9.0, and is no longer used by `offsider` or `parseMessage`.

Parse a message and convert it to a canonical form of

```
method arguments
```

Specifically, looks for messages in the form:

`key` (which converts to) `getKey` *`key`*

`key``:` *`value`* (which converts to) `setKey` *`key value`*

Syntax:

```
keySugar message
```

If the message is in the appropriate form, then returns the canonical form:

```
method arguments
```

else, returns an empty string.

If `getKey` is implied for a key that doesn't exist, then an empty string is returned (so we can substitute `messageNotImplemented` or similar.)

If `getKey` is implied, but more arguments are supplied, an Error is raised.

It can be seen that this method has a high probability of deciding that the message is in the correct form, because it doesn't mind if the key doesn't already exist.

This method is called by `parseMethod`, **only** if the message was not already in the form *method arguments* (using the test `isMethod` *`method`*).

WARNING: If you extend or over-ride this method for an offsider:

Only use syntax like `this` *`knownMethod arguments`*, rather than the more general `this` *`message`*, to avoid infinite recursion. Be even more careful if `isMethod` has been over-ridden.

messageNotUnderstood

What to do when a message is not understood by this offsider.

The standard response (implemented here) is to do nothing.

Do not rely on this method to capture typos. There is a lot of scope for the default syntactic sugar to swallow messages, even if the message doesn't actually match anything. This is especially true of `getSetSugar`.

See `offsider`, and methods `parseMessage` and `isMethod`

noMessage

What to do when an empty message is sent to an offsider.

The default action is to return the base directory for the offsider.

Note. `noCommand` is a deprecated name for this method.

NOP

No action is taken.

This method cannot be over-ridden. The response to this message is hard-coded into the `offsider` executable.

overrideSugar

This method is not implemented in the framework. However, if you implement this method for an offsider, then `parseMessage` will run this method **before** other checks for syntactic sugar.

The syntax must be:

```
overrideSugar message
```

The method must return a string in the form:

```
knownMethod arguments
```

where *`knownMethod`* is the name of a method known to that offsider, and *`arguments`* are the arguments to be passed to that method.

If the message is not recognised by `overrideSugar` then it must return an empty string, so that `parseMessage` can continue checking.

WARNING: If you implement this method for an offsider:

Only use syntax like `this` *`knownMethod arguments`*, rather than the more general `this` *`message`*, to avoid infinite recursion. Be even more careful if `isMethod` has been over-ridden.

parseMessage

Parse a message and convert it to a canonical form of

```
method arguments
```

Includes the possibility of syntactic sugar.

syntax:

```
parseMessage [ notMethod ] message
```

where *`message`* is the incoming message.

If the keyword `notMethod` is specified, then will not check to determine whether the first token is a method name.

returns a message in the form

```
method arguments
```

NOTE: The `offsider` executable calls this method with the `notMethod` keyword specified, **after** it has already checked to see if the first word is a known method name. This means that you can't override `parseMessage` to hide the names of known methods.

WARNING: If you extend or over-ride this method for an offsider:

Only use syntax like `this` *`knownMethod arguments`*, rather than the more general `this` *`message`*, to avoid infinite recursion. Be even more careful if `isMethod` has been over-ridden.

rawFind

This method is a special-purpose method, and is used by `offsider` to get a *foot-hold* into the operation of a particular offsider.

By default, `isMethod` is a wrapper to `rawFind`, so they have the same functionality.

`rawFind` cannot be overridden, any attempt to do so will fail.

An application programmer would not normally have a reason to use `rawFind`. Use `isMethod` in preference to `rawFind` unless you have studied the source-code to `offsider` in detail and understand exactly what you are doing (and why). Do not be fooled by the similarity in functionality. The two methods have very different reasons for existing.

Note. This method is absolutely critical to the correct functioning of the Offsider framework.

sugar

This method is deprecated as of version 0.9.0, and is no longer used by `offsider` or `parseMessage`.

Parse a message and convert it to a canonical form of

```
method arguments
```

This is called by `parseMethod`, **only** if the message was not already in the form *`method arguments`* (as determined by `isMethod` *`method`*.)

Specifically, looks for syntactic sugar **other than** the normal *`key`* and *`key`*`:` *`value`* stuff.

syntax:

```
sugar message
```

If the message is a recognised form of syntactic sugar, then returns a message in the form

```
method arguments
```

else, returns an empty string.

The default Offsider behaviour is to do nothing. Syntactic sugar to do with getting and setting keys is handled seperately by the `keySugar` method.

WARNING: If you extend or over-ride this method:

Only use syntax like `this` *`knownMethod arguments`*, rather than the more general `this` *`message`*, to avoid infinite recursion. Be even more careful if `isMethod` has been over-ridden.

which

This method acts like the normal `UNIX which` command. It was neccessary to implement it as a method due to intractible differences in the `which` command across different variants of `UNIX`.

Syntax:

```
    which executable
```

Where *`executable`* is the name of an executable.

`which` will search through the directories listed in the `$PATH` environment variable, looking for an executable named *executable*.

If found, it will return the full path to that executable.

If not found, it will return an empty string. (Not all versions of `UNIX which` do this).

Code Reusage

Code reusage is an important concept in object oriented programming, and is often listed as one of the key strengths of the object oriented paradigm.

In an object-oriented application language, code reusage is typically (but not always!) implemented by means of classes and inheritence.

This is not really an option in the offsider framework, due to the persistent and exposed nature of the offsider's infrastructure. (It is possible to implement classes and inheritence, but they turn out to be very problematic, and create as many difficulties as they solve).

Instead, the offsider framework provides quite different ways to reuse code. They are more in line with existing `Unix` practices.

Basically, the methods documented in this section provide various ways to **copy** code into an offsider from different sources.

Warning: The methods documented in this section may well change between now and version 1.0, since we do not feel this is the best possible API for code-reusage in the offsider framework.

allFrom

Copy keys, methods or other information from another offsider.

`allFrom` will overwrite existing files, whereas `moreFrom` will not.

syntax:

```
    allFrom offsider [ type [ recursive ] ]
```

`offsider` is either the *base directory* of the offsider, or its *named executable*.

`type` specifies the type of information to copy, eg `methods`, `keys`, etc.

`type` is in fact the name of a subdirectory within the source offsider.

`type` defaults to `methods`.

if `all` is specified for *`type`*, will copy everything except `var/`.

if `recursive` is given as a keyword, then the copy is recursive.

installMemberDirTo

Copy files from one directory to another.

syntax:

```
installMemberDirTo sourceDirectory targetDirectory
```

`sourceDirectory` contains files which are the keys to be copied.

`targetDirectory` is the directory that the keys are put (normally `baseDirectory/keys`)

installMembersTo

copy data from a specification file into files in a specified offsider

syntax:

```
installMembersTo sourceFile targetDirectory
```

`sourceFile` contains lines in the form

```
key: value...
```

(may also contain empty lines, and comment lines starting with `#`, both of which are ignored)

`targetDirectory` is the directory that the keys are put (normally `baseDirectory/keys`)

installMethodsTo

Copy executables from one directory to another.

syntax:

```
installMethodsTo sourceDirectory targetDirectory
```

`sourceDirectory` is the directory that contains the methods to install

`targetDirectory` is the directory to install into (normally `baseDirectory/methods`)

moreFrom

Modify the offsider by copying methods and keys from another one.

Will only copy methods or keys that do not already exist. (Use `allFrom` if you want to overwrite existing methods or keys)

syntax:

```
moreFrom offsider
```

`offsider` is either the *base directory* of an offsider, or its *named executable*.

upgradeFromSource

Upgrade an offsider from its source directory. The offsider must be set up before-hand, by setting appropriate keys.

Syntax:

```
    upgradeFromSource
```

The source directory is specified in the offsider key `sourceDirectory`.

The upgrade is performed by the method `upgradeMe`

The arguments for `upgradeMe` is specified in the offsider key `sourceUpgradeList`.

upgradeMe

Upgrade an offsider's methods and keys from information in the current directory.

Methods are taken from a subdirectory `methods/`

Keys are taken from a file `pairs` and a subdirectory `keys/`

syntax:

```
    upgradeMe [ directoryList ]
```

directoryList is a list of directories and keywords, the same as for method `upgradeMeFrom`

If not given, upgrade from pairs, keys and methods, using `upgradeMeDefault`.

Before copying any files, run `./pre.upgrade` if it exists. After copying all files, run `./post.upgrade` if it exists.

upgradeMeDefault

Upgrade an offsider's methods and keys from information in the current directory.

Methods are taken from a subdirectory `methods/`

Keys are taken from a file `pairs` and a subdirectory `keys/`

Syntax:

```
    upgradeMeDefault
```

This method is a wrapper for the methods `installMembersTo`, `installMemberDirTo` and `installMethodsTo`. Refer to them for more detail.

upgradeMeFrom

Copy code or data from the current directory into the offsider, by listing the subdirectories to copy.

syntax:

```
    upgradeMeFrom directoryList
```

where *`directoryList`* is a list of subdirectories and optional keywords.

For each directory in the list, the contents of that subdirectory is copied from the current directory to the offsider infrastructure.

The following keywords are also recognised:

`executable` - from now on, ensure that all files copied are made executable.

`regular` - from now on, don't force files copied to be executable (this is the default).

`pairs` - look for a file named `pairs`. Use it to create files in the `keys/` directory, according to the information in that file.

This method is a wrapper for the methods `installMembersTo`, `installMethodsTo` and `installMemberDirTo`. Refer to them for more detail.

Child Offsiders

An offsider can have one or more children, which are themselves offsiders. These are called **Child Offsiders**. For historical reasons, these are also called *Child Dictionaries*.

Typically, you would send a message to a child offsider by sending the message through its parent.

For example, if offsider `Foo` has a child offsider `Baz`, then you would send a message to `Baz` by going:

```
Foo Baz message for Baz
```

Child offsiders are implemented in such a way that it is completely contained within the parent offsider's infrastructure. In addition, the parent can be moved within the filesystem, or archived using `tar` or similar, without breaking the parent/child relationship or the ability to send a message to the child through the parent.

NOTE: Most of the following methods will be deprecated soon. The following methods all assume that the *Type* is called `subdictionaries`. In future, this is likely to change to `offsiders`, and many of these methods will be deprecated for the various *actions* defined in the Type framework.

addChildDictionary

Create a new offsider, and implement it as a child of the current one.

Syntax:

```
addChildDictionary childName [ offsider ]
```

If *`offsider`* is given, (as either a *baseDirectory*, or a *named executable*), then the child is a clone of that offsider. Otherwise the offsider is a default offsider, (effectively empty).

The new offsider infrastructure is placed within the current offsider infrastructure. This allows the current offsider to be moved or `tar`red without breaking any links.

childDictionaries

List the names of all child dictionaries for this offsider.

Syntax:

```
childDictionaries
```

hasChildDictionary

Determine whether a child offsider exists

Syntax:

```
hasChildDictionary name
```

Returns the absolute path to the child offsider, if it exists. otherwise, returns a null string and an error condition

parentDictionary

Assuming this is a child offsider, send a message to the parent offsider.

syntax:

```
parentDictionary [ message ]
```

WARNING: No checks are done to determine whether this truly is the child of an offsider.

Miscellaneous

This section documents various miscellaneous methods that don't fit into any other section.

asText

Return a summary of an offsider. Includes the names and values of keys, and the names of all methods and attachments.

syntax:

```
asText [ all ]
```

If the keyword `all` is used, the summary will include the names of all methods that the offsider recognises, and the contents of the offsider's meta-data.

baseDirectory

Returns the base directory as a fully resolved absolute path.

Syntax:

```
baseDirectory
```

cli

Provide a simple command line interface, to allow the user to send a series of messages to the offsider.

Syntax:

```
cli [ option list ]
```

option list can contain any of the following, in any order:

`prompt:` *`string`* make *`string`* the prompt

`prompt` turn prompts on (this is the default, so it's not needed)

`noprompt` turn prompts off

`echo` echo each command before it is executed

`noecho` do not echo commands. (this is the default, so it's not needed)

Note: There does not seem to be any way to force the prompt to contain spaces.

Usage (within a shell-script):

```
this cli noprompt noecho <<EOF
message1
message2
message3
EOF
```

Error

Generate an error condition.

- prints an error message to `stderr`

- prints an error number to `stdout`

- uses the same error number as an exit code, so exits with an error code.

Syntax:

```
Error [ no-tree ] messageKey [ parameters ]
```

where *`messageKey`* is a string that identifies the error message, and *`parameters`* are extra parameters that are inserted into the error message, according to the specifications for that message.

Usage:

```
exit `this Error [ no-tree ] messageKey [ parameters ]`
```

By default, prints the entire process calling tree. If `no-tree` is specified, the calling tree is suppressed, but the error message is still printed.

has

Determine whether the offsider has a particular type of member, with a particular name.

Syntax:

```
has type name
```

`type` is the type of member, for example `key`, `method` or `attachment`.

`name` is the name of the member.

Returns the full path to that member, if it exists. Returns an empty string if the member does not exist.

For example:

Test if the offsider has a key named `foo`:

```
this has key foo
```

Test if the offsider has a method named `bah`:

```
this has method bah
```

Note: Because of the way offsiders are implemented, you can also specify the plural form for *`type`*:

```
this has keys foo
```

You cannot, however, specify more than one name.

Explanation: Each type is stored in a subdirectory named for that type, so for example all keys are stored in a subdirectory called `keys/`.

This is why the plural form is also recognised. In fact, this method first searches for a subdirectory having the same name as the *`type`* argument, and if it doesn't find it, it adds a terminal `s` to the *`type`* argument and searches for a subdirectory with that name.

This method is an attempt to replace the various methods `hasKey`, `hasMethod` and so on with a single, inclusive method. It also means that if you create new member types for custom offsiders, then this method will still work on those new member types.

Id

Returns the unique identifier for this offsider.

Syntax:

```
Id
```

This identifier is generated when the offsider is created, and uniquely identifies this offsider, even if it is a clone of another offsider.

Warning: A known bug means that child offsiders of a cloned offsider will inherite the same Id as the corresponding children of the original offsider.

LICENCE

Returns licencing information.

The **Offsider** software is released under the terms of the GNU General Public License (GPL), version 3 or later.

metadata

Return the offsider's metadata

Syntax:

```
metadata
```

An offsider's *metadata* consists of the contents of the files in the `var/` subdirectory of the offsider's base directory.

not

Reverse the (logical) sense of a message.

Syntax:

```
not message
```

Like the shell, an offsider uses an empty string for false. Any other string is considered true.

This method first sends *message* to the current offsider, using

```
this message
```

If an empty string results, it returns a non-empty string.

If a non-empty string results, it returns an empty string and exits with a non-zero return code.

Profiler

Write a message to a logfile, including a timestamp.

Syntax:

```
Profiler [ message ]
```

where `message` is some text.

The location of the logfile is determined by the environment variable `$OFFSIDERPROFILETO`. If not set, it defaults to `/usr/local/share/Offsider/Profiler.out`

Note: The executable that implements this method (`offsider.Profiler`) is used by `offsider` to do automated profiling. `offsider` has its own convention that enables the logfile to be analysed by a special-purpose offsider called `Profiler`.

You are free to use this method, but please be aware that it may be in conflict with the normal profiling operations.

`offsider` uses the environment `$OFFSIDERPROFILE` to determine whether to perform profiling. If non-empty then `offsider` will produce profiling information.

Note: The Offsider framework comes with a dummy place-holder method, and an optional `Profiler` package which needs to be installed separately.

To get true profiling, you need to install the optional package.

The `Profiler` offsider is documented separately. It has methods for analysing the logfile produced by `offsider`.

pstree

Show the process tree, from the current process down to `init`.

Syntax:

```
pstree
```

shell

Execute shell commands within the context of the offsider.

syntax:

```
shell [ shell command and arguments ]
```

This method first makes the base directory the current working directory, then executes the shell command with the arguments.

Be careful with wild-cards, escape characters, and other characters that have special meaning to the shell. It is unlikely that they will survive the offsider processing stages, no matter how you quote them.

Use of this method is discouraged, because it provides a temptation to break the object-oriented principle of information hiding.

If you find that you are using this method repeatedly to do the same task, you should consider writing a method to wrap the usage of `shell`. At least that will provide a more object-oriented interface for the same functionality.

You should also look again at the methods provided by the framework. Maybe there is already a method that does what you want.

You should also review why you are resorting to the `shell` method. Are you thinking in the offsider paradigm or the UNIX shell paradigm? In other words, are you thinking of the offsider as an *object*, or as a *directory*?

toBaseDirectory

Takes either a named executable or the base directory for an offsider.

Returns the base directory.

Syntax:

```
    toBaseDirectory [ namedExecutable ]
```

or

```
    toBaseDirectory [ baseDirectory ]
```

This is a utility method. Sometimes it is convenient to specify an offsider by giving the named executable, at other times the base directory is more convenient. This method allows either form to be used.

version

Returns a string that identifies the specific version of the installed Offsider framework software.

Syntax:

```
    version
```

Be aware that this method might be over-ridden for specific offsiders, especially if you have downloaded special-purpose offsiders as separate packages.

The executable `offsider.version` will return the version of the Offsider framework.