Context

About VAMP context

Everything in your entire VAMP project structure is based on: "concepts", "contexts", "aspects" and "values".
At "project root" level, everything directly inside it are referred to as "concepts" - which exist in the "context" of your "project root".

The "aspects" of every "context" -or "concept" has "values" which define its attributes, relationships, methods, and contents. The collective for "aspects" is simply referred to as `meta`, represented by the `$` character wich means "self".

Contexts can be defined anywhere in your project tree by folders & files (or symbolic links), as each of these need to be named something.
Contexts can also be defined inside VAMP files using the "context operators".
As you may have guessed by now that you cannot simply access any "path" (folder or file) on the local system (where VAMP is installed), because your VAMP runtime (project) can only access its own contents; however, if you need to access something outside of your project tree, then you can use "symlinks" to do so. These links can be defined either "implicitly" or "explicitly". The reasons for this is both for security and structure simplity, but we'll get to these later.

It is important to note in these docs about references to "implicit" & "explicit". Vamp is designed to make your life easier, so if things get done "implicitly" it simply means: "automatically done the way it it is implied"; where "explicitly" means: "do the followng exactly".

The difference between "implicit" and "explicit" is:

- "implicit" is less coding, but may run a bit slower at runtime (in some cases)
- "explicit" may run a bit faster at runtime, but also may require a bit more coding

With the above in mind, when using the following operators after a named (or unnamed) entity, it usually means this:

- the `:` operator defines -or re-defines an entity (or result) explicitly;
- the `{}` and `[]` and `()` operators defines -or `{modifies}` -or `[extends]` -or `(calls)` an entity (or result) implicitly.

The rest of this doc will explain.

Aspects

Here is an example of how to define some "thing" and give it some properties:

```
ride {color:'blue', roof:'none', status:'notorious'}
```

From the example above, the curly braces turned the word "ride" into a "named entity".
This means that it got registered as a "named thing" within the context of wherever the code above resides. It automatically gained a meta-property: "Name" and its Name is: "ride".
It has 3 data-properties: "color", "roof" & "status".

You can access these properties by using the "select" operator `.` (dot) like this:

```
ride.$          ::    {color:'blue', roof:'none', status:'notorious'}
ride.Name       ::    ride
ride.roof       ::    none
```

Contents

To assign contents to an entity, we can either define it with the "content operators", or with the "Gist" aspect like this:

```
ride [ 'some text contents' ]       :: implicitly

ride {Gist:'some text contents'}    :: explicitly
```

You can also define an entity with aspects and contents with the `{}[]` operator sequence like this:

```
ride {color:'blue', roof:'gone'}
[
   'seats'
]
```

In the example above, the `{}` denotes the aspects of "ride" and the `[]` denotes the "contents" of "ride".

Functions & Methods

A function (or "called on method") is defined with the `()[]` operator sequence like this:

```
ride: ()
[
   'vrooom!'
]
```

In the example above, the `:` states "is" `()` denotes the arguments of function:"ride" and the `[]` denotes the "contents" (context) of function: "ride".
When calling `ride()` it will yield: `vrooom!`.

A "callee" (function or method) can return a yield (echo) explicitly, like this:

```
ride:()
[
   $Echo('vrooom!')

   'blah'
]
```

In the example above, "vrooom!" is its yield, and "blah" is skipped.

Functions & methods in VAMP are both synchronous AND asynchronous -depending on how they are called.
The `()()` context sequence-operators simply means: "call the return value"; however, the `()[]` operator-sequence means: "attach this context as call-back to this call-instance".
The following example will explain:

```
ride()          .:    vrooom!

ride()
[
   $Echo        .:    vrooom!
   $Name        .:    ride
]
```

To explain the above example, you need to know how VAMP handles function (-or method) -calls:

- the "call handler" is initiated with an instance of itself when a call is made to it
- if no call-back context is defined for this instance then it returns its yield (echo)
- if a call-back context is defined it sends its yield (echo) to this context when all is done

Notice in the example above that there is no `:` in front of the `()`, which means to "call" (-NOT "define").

Definition

VAMP offers different ways of defining, modifying & extending things. This is by design as some things are better to define one way and modify in another way, depending on the dynamics of your logic.

The following example shows how to create an entity and modify it after:

```
ride: {}

ride.roof: 'no'
ride.Gist: (ride.roof + ' seats')
ride.Call: ()
[
   'vrooom!'
]
```

The example above shows a rather "long winded" way of creating & modifying an entity, but it shows how things are usually done in other languages; however, it works in VAMP also.

In VAMP, the above can be done explicitly without repeating yourself, like this:

```
ride:
{
   roof: 'no'
   Gist: ($roof + ' seats')
   Call: ()
   [
      'vrooom!'
   ]
}
```

In the example above, the value of "Gist" is assigned in an interesting way.
Here's how it works:

- when an entity is being defined, its defined aspects are availale to the next statement within the same context
- the expression that follows the `:` inherits the current context so `$` (self) refers to "ride"
- `$` has an aspect named: "roof" which has a text value: "no".

So, the value of "Gist" in the example above is "no seats"; hence, the contents of "ride" is "no seats".
Erpessions will be covered in detail later, but the above should help getting a firm grasp on how things operate in context.

Calls & expressions

A "call" is an "expression" that operates on the context of what is "called upon".
In VAMP, anything can be "called" "implicitly" or "explicitly" (with arguments), even if "the thing" does not exist as a named thing.

When something is called, it does 1 of 4 things, depending on how this thing being called is defined:

1. if it does not exist then the "echo" would be "Void" (nothing)
2. else-if it has a "Call" aspect (discussed below) then that is `called` accordingly
3. else-if it has "contents" then its contents are `searched` accordingly
4. else its aspects are `searched` accordingly

When an entity is called and it does not have a "Call" aspect, it implies that you are making a "data-lookup" and in such cases, the following apply:

- if the lookup is done on aspects, then "meta-aspects" are skipped
- lookups always yield a list, even if nothing is found -an empty list is returned
- the lookup result list contain pointers to the "real entities", so modifications can be made implicitly or explicitly

Here are some examples of how to "call" things "implicitly"; the comment on the right shows what is returned:

```
ride:
{color:'blue', roof:'none', status:'notorious'}
-------------------------------------------------------------------------

ride()                      ::    [color:'blue', roof:'none', status:'notorious']
ride(`moo`)                 ::    []
ride(`color`)               ::    [color:blue]
ride(`moo` | `color`)       ::    [color:blue]
ride(`roof`, `status`)      ::    ['roof:none', 'status:notorious']
```

Let's go through the example above line by line:

- `ride()` no arguments are given, so a copy of "ride" is returned
- `ride('moo')` "ride" has no aspect named "moo", so: "nothing" is returned
- `ride('color')` the value of aspect "color" in "ride"
- `ride('moo' | 'color')` value of "moo" -or value of "color" in "ride"
- `ride('roof', 'status')` value of "roof" -and value of "status" in "ride"

The example and explanation above shows how things are called "implicitly", meaning: what is "implied" by the call - depending on what it is called upon.

To call things "explicitly" - it works exactly the same, but in this case you specify exactly what to "call upon", like this:

```
ride.$(`roof`, `status`)      ::    ['roof:none', 'status:notorious']
ride.Gist()                   ::    []
```

In the example here (above):

1. the `$` is used to select the "aspects" of "ride", so the call is done on the aspects explitly;
2. the "content" (or "Gist aspect") of "ride" is selected, so the call is done on the contents explicitly -which in this case is "nothing" because our "ride" has no contents.