Data Types

The "data types" in VAMP are divided into 2 categories:

- `Type` primary data-type
- `Kind` secondary data-type

These are global meta-attributes and is automatically defined & updated as entities are created & modified. The data-types also have complimentary globals which are used for comparrison. We say "data-types" here because, that is what it is known as; however, "data" is a very broad term. All information imaginable that can be stored on a disk -or transmitted via any medium is simply called: "data" and so is the code you type in programming as well.

Eventually the data changes as it is interpreted, but we need a way to distinguish what belongs where. VAMP has a `Data` "data-type" which simply means anything that can be stored as a bunch of characters, from text through to binary data: anything not parsed (yet) is simply just plain: `Data`. Before you start sweating - have a look below - after the examples that follow; you will see that the `Data` primary type has many secondary types.

From here on out we will simply refer to primary "data-type" as: `Type` and secondary "data-type" as `Kind`. So, we can say: "the Type of this", or "the Kind of that". Remember that VAMP is extensible, so you can extend the types & kinds to suit your specific needs, either in your project or in your VAMP base code; -the latter will be available for all your projects.

The concepts (global nodes, functions & constants) you see below are covered in the "concepts" doc, so keep reading ;)

Type

The types are the corner stones in VAMP. Some of these type names are not only static constants, but also functional nodes. However strange and wonderful they are, they do a lot of work for you so you can focus on the project and not spend so much time in persuading the compuer to do what you want. This is especially true with the "kinds" as you will see in a bit.

These are the types:

```
Void        ::    ()                                   ::    undefined
Spin        ::    +, ?, -                              ::    polarity
Nume        ::    1, 0.9, #123.kg                      ::    numbers
Data        ::    'abc', "123", `*.*`                  ::    text or binary
List        ::    []                                   ::    list of contents
Node        ::    {}                                   ::    functional object
Time        ::    Time(`2016-03-03`)                   ::    date, time
Fail        ::    (!?/*)                               ::    holds an error
```

Spin

The word "spin" was decided on this data-type because of its possible use-cases and what it could mean during conversation -either with coleges -or in your own contemplation of how to do things logically.

This type can be used as "boolean", -or "polarity", -or a "state mechanism", -or "flavor", -or "spin-off type" of anything, -depending on how it is used in your logic.

It is based on the theory and application of "Qubits" (quantum bits).
As you may know, traditional bits have only 2 possible states: 1 or 0 (`true` or `false`). Qubit values are calculated from "probability" in the form of "spin up" & "spin down" as the polarity of an atom rapidly changes.
This means it could be in 1 of 3 states: "positive", "negative", or "both". The latter (both) could be explained as "undecided", -or "not enough pull on either side to make any conclusion".
This concept is adopted in VAMP as "polarity" and it is part of the meta-aspects, named: "Spin".

The meta-aspect: "Spin" has 1 of 3 possible states and its initial value is determined by the `type` of the entity it belongs to and the entity's value:

- `+` : `True`, numbers more than `0`
- `?` : `None`, numbers equal to `0`, anything else
- `-` : `Fals`, numbers less than `0`

However you decide to use it, being able to set -or change -or view the polarity of something in 3 states is very useful, especially in data-type declarations, and expressions as you will see later on.

Fail

The `Fail` type is a meta-object reference that is only created and returned upon error. VAMP does its best to prevent errors; even so, sometimes you may code something horrendous in the early hours of the morning. In such cases VAMP fails "gracefully". Instead of having the entire process break; the `Fail` type flows along as far as possible until it reaches the output eventually through your own process flow.
A `Fail` value is logically "boolean false" and you can easily check if a value `type` is "Fail" like this:
`(!foo)` -OR- `IsFail(foo)` -OR- `foo.IsType(Fail)`.

Kind

The "kinds" are dependent on the "types" and they extend & compliment the types in extraordinary ways. These easily cut a lot of hours - when you take debugging in consideration of how many times you have to create & modify functions & expressions to identify some piece of data for you.

```
NoneVoid    ::    undefined

PosiSpin    ::    +           (boolean true)
IffySpin    ::    ?           (logically null)
NegaSpin    ::    -           (boolean false)

BareNume    ::    0           (zero)
IntgNume    ::    1           (integers)
FracNume    ::    0.1         (has a fraction)
UnitNume    ::    #FFF        (measured)

BareData    ::    ""
SpinData    ::    "True"      "None"      "Fals"
NumeData    ::    "123"       "0.4"       "#2.5.hz"
TextData    ::    "abc"       "foo bar"   some text
ListData    ::    "a,b,c"     "[1,2,3]"
NodeData    ::    "foo:1"     "{a:b}"     "(){}"      "foo(){}"
HardData    ::    (binary)
BiosData    ::    "/pth/dir"  "http://example.com"

BareList    ::    []          (empty array)
FlatList    ::    [a,b,c]     (no items are lists or nodes)
TreeList    ::    [a,[bleh]]  (some items are lists or nodes)

BareNode    ::    {}          (empty node)
FlatNode    ::    {a:1,b:2}   (no items are lists or nodes)
TreeNode    ::    {a:[c]}     (some items are lists or nodes)
FuncNode    ::    ()[]        {}()[]      foo{}()[]
BiosNode    ::    Sock(#WS:127.0.0.1:9000')[]

NumeTime    ::    Time()
DateTime    ::    Time(%YMD)
EmitTime    ::    Tick(every: #1:Sec)[]
```

Some entities may be more than one kind at a time. The `.Kind` meta-aspect will only be 1 though, and this would be the "most likely" (simply: the one that comes first).
You can check for more than one Type -or Kind at a time. The `isType()` & `isKind()` functions can also accept expression calls, so instead of repeating things you can do this instead:

```
:: check ALL
----------------------------------
foo.IsKind(FuncNode & TreeNode) :: same as (foo.isKind(FuncNode) & foo.isKind(TreeNode))
----------------------------------



:: check ANY
----------------------------------
foo.IsType(Spin | Nume)         :: same as (foo.isType(Spin) | foo.isType(Nume)
----------------------------------
```

Here are a couple of examples for clarity:

```
numb: 123
frag: 123.0
loaf: #0.7:kg

tnum: '123'
ptxt: 'abc'
path: '/some/place'

elst: []
nlst: [1,2,3]
tlst:
[
   ['a']
]

---------------------------------------

numb.Type                ::     Nume
tnum.Type                ::     Data
elst.Type                ::     List

---------------------------------------

numb.Kind                ::     IntgNume
frag.Kind                ::     FracNume
loaf.Kind                ::     UnitNume

tnum.Kind                ::     NumeData
ptxt.Kind                ::     TextData
path.Kind                ::     BiosData

elst.Kind                ::     BareList
nlst.Kind                ::     FlatList
tlst.Kind                ::     TreeList

---------------------------------------

(numb.Type = Nume)       ::     True
(tnum.Type = Data)       ::     True
(elst.Type = List)       ::     True

(numb.Kind = IntgNume)   ::     True
(tnum.Kind = NumeData)   ::     True
(elst.Kind = BareList)   ::     True

---------------------------------------

numb.IsType(Nume)        ::     True
tnum.IsType(Data)        ::     True
elst.IsType(List)        ::     True

numb.IsKind(IntgNume)    ::     True
tnum.IsKind(NumeData)    ::     True
elst.IsKind(BareList)    ::     True

---------------------------------------
```