BEFE - Grey Paper

BEFE Lifecycle

A detailed description of the BEFE Lifecycle and associated Methods.

Author:	Bruce Ferris
Date:	08 October, 2016
Status:	Draft

Table of Contents

BEFE Lifecycle

The purpose of this paper is to describe the BEFE **Lifecycle** model and **Methods** used to implement it in the **BEFE Primitive Model** and **BEFE Value Model**.

> **Note:** Currently the methods are restricted to, and described using, C++ procedure calling sequences. In the future we expect this to expand to a reasonably generic and abstract specification – meaning we will take C++ calling sequence specifications out of the description and replace them with **BEFE Calling Sequence** specifications once that portion of the **BEFE Model** is complete enough to support that endeavour.

Warning: The entire set of *Methods* described here in this document are currently in a state of fluctuation because the **BEFE Lifecycle** is still under development.

So, don't be completely surprised when the list of *Methods* and terminology changes significantly over time – and it will!!!

Overview

The **BEFE** *Lifecycle* is both a model and a set of methods implemented on most **BEFE** *Values* and **BEFE** *Types*. In the future we expect this to be expanded beyond the **BEFE** *Value* and **BEFE** *Type* models to include the **BEFE Object Model** and beyond.

The purpose of the *Lifecycle* methods is to provide a language independent set of standard processes and states instances of *Primitive Types*, *Types*, *Values*, and (in the future) *Objects* go through during their lifetime, whether they are transient, persistent, shared, transported over a local or global network, and even as their specifications and implementations inevitably evolve through time.

As such, the list of **BEFE** *Lifecycle Methods* is not fixed or mandatory. But, they can be roughly categorised as follows...

- Primary Methods
- Secondary Methods
- Ancillary Methods

The long term intent of these methods is to provide the semi-formal means why which we can generate code when and where possible to minimise the amount of coding on your part – that is the long term goal but it is more of a coding principle at this point, so it currently means a bit MORE work for you not LESS.

> **Note:** It is important to point out right here at the start that not all *Types* and *Values* in the **BEFE** *Model* support and implement ALL the methods in the **BEFE** *Lifecycle*. Instead, each *Value/Type* only implements the methods that make sense for the *Value/Type* being implemented.
>
> Take, for example, a **BEFE** *Primitive Type* like `Int`... This is currently implemented in C++ as a type `signed int`.

> Since C++ doesn't provide the means for specifying a "Constructor" for the type `signed int` (C++ is notoriously and rigidly handicapped in that respect), we clearly can't implement a **BEFE** Lifecycle `StartUp` method on the **BEFE** Primitive Type `Int` or describe sub-typing of the `Int` directly using C++ examples, can we?

Each of the BEFE *Lifecycle* methods are described separately in detail below but we'll provide a quick list of the methods in each category to start off with...

> **Note:** In the following tables and subsequent descriptions in this *Grey Paper*, we don't use "real C++" to explain the methods, only "pseudo C++" – meaning we sometimes miss out specifying implementation details like `const`, since that's sort of misleading, and we introduce illegal syntax like <`T`> to indicate "whatever type we're talking about goes HERE", and other non-standard pseudo syntax.
>
> Just figured we'd point that out before it becomes too confusing to developers soaked in a lifetime of thinking C++!!!

*Why the **BEFE** Lifecycle?*

The reason we've gone to the bother of introducing the **BEFE** *Lifecycle* everywhere we can is simple...

Economy of Software Development

Let us clarify just what we mean by that phrase... By *economy* we mean the same notion database designers are thinking when they talk about "normal forms". Some people get their kicks out of inventing important sounding phrases like that, but what all it boils down to is *economy.*

It turns out that if you have a *model* defined AND your software design conforms to that *model,* then it's a short jump to being able to "generate code" instead of "writing it".

Now, that's a bigger discussion than we have space for here and *Code Generation,* as a concept, has seen lots of bad press because there's so many bad implementations out there – But we figure that if you expand the scope of *Code Generation* to include machine instructions as part of "the code" then a lot of the bad press magically disappears – especially since, in that sense, that's what compilers are supposed to to: *Generate Code.*

This whole approach is why we consider **BEFE** to be an umbrella that embraces both *declarative* and *imperative* programming styles, both of which have their time and place. So, with **BEFE** you have the choice... you can *declare* things or you can *impair* things (sorry about the cheap pun!)

As the **BEFE `Model`** expands and matures, we expect to remove the start up dependency on C++ and start providing the same features in other languages like Java, Python, and more. This will enable you to design your software in **BEFE** and not have to throw it away when talking to other programming languages or contexts.

BEFE Lifecycle and Inheritance

The **`BEFE`** Lifecycle model intentionally avoids dealing with the concept of "inheritance". This is to separate the two concepts so each can be treated by themselves. Inheritance, in BEFE, is considered as an "economy of specification" than it is a "fundamental property" of a model.

In the BEFE Lifecycle, we focus only on the context of a specific Type and leave any "supertype" as separate issue. This approach is at odds with that taken by typical "Object Oriented" languages such as C++ and Java. In those language's "models", a supertype's Constructor is called BEFORE a sub-types's Constructor and its supertype's Destructor is called AFTER the sub-type's Destructor.

What this means is that...??????????????????????????????

BEFE Lifecycle State Summary

The following table summarises the BEFE Lifecycle States as they apply to both Values and Primitives...

xxx

		Required for all...	
State Name	**Description**	**Values**	**Primitives**
`Null`	Used to determine of the value is Null	Yes	Yes*
`Empty`	Used for Sequencing	No	No
`Consumable`	Used for Copy semantics	No	No
`Mutable`	Used to restrict mutability	No	No
`Queryable`	Used to restrict visibility	No	No
`Addressable`	Used to determine direct addressability	No	No
* See note below			

Note: As a rule, the `Null` state is required for everything, end of story – almost... But, the old adage "for every rule there's an exception" applies in one case: The `Bit` Type.

This is because a single "bit" can only have the values `0` and `1`, at least in today's computers, so there's "no room at the inn" to house a `null` value UNLESS it is implemented internally as multiple bits (sort of like a Boolean can be implemented in more than one bit).

Still, the `SetNull` method (see below) will, in the case of a single bit `Bit`, will set the value to zero instead of a proper `null` value while the `IsNull` method will always return the answer `false`. – this unfortunate rule breaking case is not only counter-intuitive but clearly unavoidable.

BEFE Lifecycle

It is important to mention at this point that these States are NOT defined by BEFE to be implemented in as compile-time/runtime ??????????????????????????????

BEFE Lifecycle Method Summary

The following table summarises the BEFE Lifecycle methods as they apply to both Values and Primitives...

xxx

		Required for all...	
Method Name	**Description**	**Values**	**Primitives**
`StartUp`	Initialise from an indeterminate State to Null	Yes	No
`ShutDown`	Release resources, set to indeterminate State	Yes	No
`Reset`	Release resources, set to Null State	No	No
`Clear`	Synonymous with `SetEmpty` if present	No	No
`MoveFrom`	Move from another instance	Yes	No
`CopyFrom`	Copy from another instance	Yes	No
`IsNull`	Answer "Is this Null?"	Yes	Yes
`SetNull`	Set to Null State	Yes	Yes
`IsEmpty`	Answer "Is this empty?"	No	No
`SetEmpty`	Set to Empty State	No	No
`IsConsumable`	Answer "Is this consumable?"	No	No
`Consumable`	Set to Consumable state (if applicable)	No	No
`NotConsumable`	Clear Consumable state (if applicable)	No	No
`IsMutable`	Answer "Is this mutable?"	No	No
`Mutable`	Set to Mutable State	No	No
`Immutable`	Clear Mutable State	No	No

xxx

BEFE and Code Generation

It turns out that a substantial portion of the current **BEFE** implementation has been "manually written" by me. BUT... The end results of each manual rewriting of "the same old code" are so repetitive and have so few differences and so many things in common that it becomes clear there's great benefit to generating a lot of the code automatically in the very near future.

So, that's the plan.

> **Note:** If you happen to disagree with the whole *Code Generation* idea and approach, that's your loss... Because that's where we're heading, and doing to with a perfectly clear conscience.
>
> But if you're still unconvinced, take a closer look at C++ *Templates* and Java *Generics*... They are both just specifications for a language specific *Code Generator*, so have another think about it because both those reasonably acceptable tools are there for the same reason we're promoting – **Economy**!!!

Another *Economy Of Specification* technique that most people don't think of as such, is the entire *Object Oriented*, or *OO* approach. OO has similar intents to most other software design modelling methodologies. Try to put aside all that *Polymorphism* and *Multiple Inheritance* gobbledygook for a moment and think about *Economy* instead...

Inheritance of specification is simply *Economy* – you specify it once, and the specification gets "inherited" down the type hierarchy – no rocket science involved. Well, not much anyway!

So, that's the whole point and justification for the **BEFE** *Lifecycle*, and that's all we're trying to enable here... **Economy of Software Development**.

BEFE Lifecycle: States

Throughout the BEFE Lifecycle we refer to several states which are, conceptually, simple Boolean flags. These states are...

- **Null**
- **Empty**
- **Consumable**
- **Mutable**

These states are NOT exclusive except for one fairly obvious case... If an instance's state is **Null** then it should also be **Empty**, otherwise something has gone wrong somewhere – and it's highly likely that the "somewhere" is in code typed in by YOU, if not – many apologies and we'll fix it if you let us know!!!

It is NOT the responsibility of the **`BEFE` Lifecycle** model to enforce a specific way of implementing these states. A common example of that is the **Empty** state, which can quite commonly be a constant `False` if the value is a scalar and isn't **Null**, or can be determined by examining the size of any variable length or external data associated with the instance.

But, it IS the job of the developer to determine a sane way to implement the states without them stepping all over themselves in the process.

Note: To be open and honest about it... Nothing in the BEFE **Lifecycle** even demands that any of these states are implemented in a given *Class* so, you should only think about them if you want your code to partake in the full benefits of the **`BEFE` Lifecycle**.

A common example of NOT implementing any of these is if you implemented a "`Null` Object" somewhere, like Java's `null`, Python's `None`, and SQL's `NULL`. You wouldn't implement the `IsNull,` and `SetNull` methods on the `Null` Object, because all it could do was to return a **Null** answer – at least in theory...

If you DID implement the **Null** state on the `Null` Object then you would be begging the question "Is `Null Null?`" and that's well weird and needn't be asked to begin with. But asking "Is `X Null?`" makes sense, and that's what the `IsNull` method is for (discussed here in this paper) on whatever *Value* type `X` is.

Then there's the question "Does `X` equal `Null`", which causes different philosophical questions to arise and we'll cover that under the `CompareEquals` method later.

BEFE Lifecycle: Primary Methods

The following methods are pretty much implemented in all **`BEFE` Value** classes...

Method	Description
`Status  StartUp()`	Initialise an uninitialised instance
`Status  ShutDown()`	Finish with an initialised instance
`Status  Reset()`	Reset an instance
`Status  Clear()`	Clear an instance's contents
`Status  MoveFrom ( <T> const &that)`	Move contents from another instance
`Status  CopyFrom ( <T> const &that)`	Copy contents from another instance
`<T>    &Consumable()`	Set this instance as Consumable
`<T>    &NotConsumable()`	Set this instance as not Consumable
`Boolean IsConsumable()`	"Is this instance **Consumable**?"

Note: The `Consumable` and `NotConsumable` methods both return a `<T>` reference in C++ because those methods are quite commonly used when passing parameters or returning from a procedure.

BEFE Lifecycle: Secondary Methods

These **Secondary Methods** are implemented in a lot of **`BEFE` Value** implementations but not all of them.

For example, the concept of **Immutable** is commonly used for `String` and other *Values*, it is counter-intuitive or "out of place" for things like `File`, `Device`, or `Media` where a similar – but not exactly the same – concept is usually called **Read Only** instead and has a subtly different meaning.

Method	Description
`UInt    Length()`	Get the Length, in **Elements**, of an instance
`UInt    Size()`	Get the Length, in **Storage Units**, of an instance
`Boolean IsMutable()`	"Is this instance **Mutable**?"
`Status  Mutable()`	Set this instance to be **Mutable**
`Status  Immutable()`	Set this instance to be **Immutable**
`Status  CompareEquals( <T> const &that, Boolean answer )`	Compare instance with `that` instance - **Equality**
`Status  CompareRelative( <T> const &that, Int result )`	Compare instance with `that` instance - **Relative**

BEFE Lifecycle: Ancillary Methods

These **Ancillary Methods** are implemented in a currently “haphazard” way throughout BEFE because they don't quite fit into a semi-formalised model yet – but we expect them to at some point in the near future...

Method	Description
`String ToString()`	Convert to “printable” `String`
`Strings ToStrings()`	Convert to multiple line list of `Strings`
`Status WriteTo(Stream &stream)`	Write to a `Stream` (persistence)
`Status ReadFrom(Stream &stream)`	Read from a `Stream` (persistence)

BEFE Lifecycle: Procedures

Now that we have given an overview of the *Methods*, it is appropriate to mention that these *Lifecycle Methods* were designed to be general in nature and, as such, should also be available in the native language as **`BEFE` Namespace Procedures** as well as by *Instance Methods*.

> By the term **Namespace Procedures** we refer to *Procedures* that perform an action but are passed all the information needed through explicit *Parameters*, as opposed to *Methods* which are passed the Value/Object, sometimes called "`this`", implicitly by the language.

The most commonly used *Namespace Procedures* are the *Procedures* `IsNull` and `SetNull`, which are passed the operand as a *Parameter*. Which actual implementation of the procedure gets called is determined by the Language itself, usually by *Procedure Overloading* – which isn't a subject covered directly here.

This allows us to not be bothered with the method specific notations enforced by most languages and focus on the operation being requested instead. The C++ languages used "." or "->" syntax for this but other languages use other odd notations in their syntax.

So, using C++, the following two examples demonstrate the two different uses of the same procedure...

-
  ```
  String mystring;
  ...
  if (mystring.IsNull()) {…
  ```

-
  ```
  String mystring;
  ...
  if (IsNull(mystring)) {...
  ```

BOTH of these examples are, as far as we're concerned, just as useful depending on what you were thinking at the time you wrote them. But more importantly, BOTH are just as readable. So, we've tried to implement these "procedural equivalents" of commonly used methods wherever possible.

> **Note:** If you have a peek in some of the **`BEFE`** C++ headers, you'll find we've preceded these with the `BEFE_INLINE` macro. This is one of the few C++ macros we use in `BEFE` and it simply declares the procedure to be "`inline`" for efficiencies sake.
>
> BTW... This isn't the first and won't be the last place we'll gleefully point out that we absolutely **detest** C/C++ macros and, as far as we're concerned, Brian Kernighan and Dennis Ritchie placed a horrendous burden on mankind by introducing them in the first place – And, we highly suspect it's because they weren't touch typists, since that single personal deficiency causes untold bother in our business, almost like it's a disease of some kind!!!!

BEFE C++ Calling Sequences

The majority of BEFE Methods return a scalar value of type Status. This is a 32-bit integer that, if Zero, means "`No Error`". If the value is non-zero, it contains an enumerated value indicating "`Error <nnn> occurred`".

These `Status` values are currently visible in the C++ header file `Error.h` and the `BEFE_Home` directory file `BEFE_ErrorMessages.txt`.

As BEFE progresses, we fully expect to implement a full set of calling sequences such as...

- Positional
- Named
- Functional
- Tuples (In, Out, InOut)
- et cetera

A full description of these will be available, at some point, in a separate Grey Paper devoted to the subject.

Warning: While this isn't the place to go into a discussion about *Calling Sequences*, it IS the place to point out that BEFE is currently only in the position to a) talk about C++ calling sequences, and b) typically return only `Status` values unless absolutely necessary for readability and usability in the code.

BEFE C++ `Status` result Ordering

This may seem like an out of place subject but we figure this short description goes right after mentioning Calling Sequences...

As mentioned above, most BEFE Lifecycle methods return a single Status type result to indicate their success or failure.

However, we feel it is important to establish an early standard semantic description of exactly what we can expect this Status value to mean...

The problem is a fairly simple one if you ask "What does error `<nnn>` mean?" because it raises several immediate issues and questions when you start writing code that calls things like that...

1. Does it mean "error `<nnn>`" is THE ONLY thing that happened stopped it from working?
2. Does it mean "error `<nnn>`" is THE FIRST error it got when processing?
3. Does that mean "`<nnn>`" is really an ERROR?
4. What state did this method/function/procedure leave my input-output and output parameters in when it returned a non-zero `<nnn>`?

Oddly, I've never seen Microsoft or anybody else that uses that style of calling sequence actually answer those questions in detail. Make you wonder, doesn't it? I can't even begin to count the number of times I've had to inspect the state of my parameters after calling somebody else's library just to make sure, or prove, that THEY did or didn't screw things up.

Well, we can't answer that for YOUR code but we can at least lay down some recommended guidelines which we ourselves try to follow in our code...

Answer to Question 1: No, other "bad stuff" could have happened but this is the one that we're reporting because we figure it will be the most helpful. Not only isn't that necessarily the ONLY error that occurred, another one may have been reported and we turned it into that because we figured it would be a lot more helpful than what the original error was.

Answer to Question 2: No, it's not necessarily the FIRST one, see answer 1 above. But, in the case of functions/methods which are clearly a set of calls to other methods (like many of the **`BEFE` Lifecycle** methods), we highly recommend returning the Status of the FIRST error encountered but continue on your way if your method is responsible for performing a well defined and exact operation (like many of the BEFE Lifecycle methods).

Answer to Question 3: No, we've declared Status to be a signed 32-bit value, so negative "errors" can be considered "warnings" although, as of writing, we haven't really used this a lot.

Answer to Question 4: That's an extremely important question commonly never answered... We highly recommend NEVER destroying parameters documented or declared as *Input*. We also highly recommend that you should ALWAYS keep your return results in local variables and then, if everything goes to plan without any errors, move them all in to the caller's *Output* parameters just before your function/procedure/method returns. We try to adhere to this principle throughout **`BEFE`** and not just in the **`BEFE` Lifecycle** methods.

But where all this goes if "warning" are implement we're not really sure what to say yet, sorry!!!

Methods: StartUp and ShutDown

The `StartUp` procedure is responsible for initialising a new *Value* instance from uninitialised memory or whatever **Storage Location** (see below) it exists in.

For *Value* instances whose S*torage Location* is in processor memory, no matter how long or short the instance's lifetime is, this method serves the same purpose as the C++ constructor method.

For *Value* instances that whose S*torage Location* is somewhere out of the processor's "direct reach" (like on disk, on the network somewhere, out there in "the cloud", et cetera), this method serves a subtly different purpose than the C++ constructor methods does.

> **Note:** For now, and until we finish the semi-formal **`BEFE`** *Persistence Model*, we will side-step this distinction because we haven't yet completely firmed up all the details of the BEFE approach to "persistence" – but still, we wanted to point out that we're heading in an unconventional direction here at the outset by taking the concept of *Storage Location* into account in the **`BEFE`** *Value Model*.
>
> We strongly suspect this part of many Object Models (including Java's) was approached from a completely wrong, or at least naive, perspective and that we've all paid the price, over and over again, for those designer's lack of imagination and foresight.

Meanwhile, back to `StartUp`...

When `StartUp` is called, it is expected to put the Value instance in a **Null** state meaning that the Value has no memory or other resources allocated to it other than the fixed size storage location it has been allotted.

In addition, the **Null** state means it not only doesn't have any "contents", but it has no idea of its "identity" either. This implies a couple of things...

- The *Value* represented by the instance is not referenced anywhere else or, if it is, this instance certainly does not know where those references are and can't answer to it properly
- The *Value* represented by the instance has NO IDENTITY

The whole point of stating the above is that once `StartUp` is called, then subsequent or "wrapping" methods must supply the instance with any contents or identity.

> **Note:** That is the subtle difference between the `Reset` and `Clear` calls we describe a bit later... The `Reset` method calls `ShutDown` immediately followed by `StartUp`, placing the instance in a **Null** state and losing any identity where the `Clear` method, retains the instance's identity but discards the contents. More on that later...

As mentioned above, StartUp does, in many cases, sort of behave like a C++ constructor method but we've defined the precise behaviour of StartUp more clearly than C++ defines constructors. Notice, we don't have a BEFE "copy constructor" but we have StartUp followed by CopyFrom. So, we can do the same actions as expected from standard C++ methods but with more clearly defined semantics. A good step in the right direction we think!!!

The `ShutDown` method serves to release the entire Value instance's contents AND identity, thereby freeing up any resources used by the instance.

So, in essence, the `ShutDown` method serves pretty much the same purpose as a C++ destructor method except that it returns a Status value letting you know if it worked or not AND it can be called whenever you wish without "hiding around the corner" like the C++ destructor does.

We do not demand that your Value's `ShutDown` "clear the storage" it leaves behind (that's sort of up to you and the Storage Device) but we DO demand that `ShutDown` releases all resources associated with the instance because it is always the last method to be called in a Value instance's lifetime.

> **Note:** Just like the C++ destructor, it is NOT the job of `ShutDown` to release the physical storage the instance is sitting in. In C++ that's left up to `operator delete`. In **BEFE** that's left up to YOU, the *Storage Device*, and the **BEFE** context your implementation is designed to run in (e.g. C++, Python, Java, et cetera).

A typical `StartUp` method would simply initialise all *Primitive Type* members to a fixed (preferable **Null**) value and invoke the `StartUp` method on all *Value Type* members – if you remember earlier we mentioned that *Primitive Types* don't have a `StartUp` method so the initial value has to be assigned "manually".

Once `StartUp` has performed these actions, it has satisfied it's purpose in life... To initialise an uninitialised *Value* instance to a consistent **Null** state.

• • •

A typical `ShutDown` method would simply invoke the `ShutDown` method for each of the *Value Type* members in the class.

A couple of final things about `StartUp` and `ShutDown` should be pointed out at this point...

- **BEFE** `StartUp` and `ShutDown` methods are NOT required by **BEFE** to initialise/free the instance members in any specific order. This is because, as far as **BEFE** is concerned, member order is inconsequential and may even be non-existent or implemented differently on different target platforms to "get things just right". So, if a specific order IS implemented that's up to whoever wrote or generated the implementation.

- Since the requirements of the `StartUp` and `ShutDown` methods are extremely terse, and well defined, and straight forward, is it any wonder we started thinking about "code generation" to keep from having to write EVERY `StartUp` and `ShutDown` for all our class implementations? Interestingly, the same thing holds for the rest of the **BEFE Lifecycle** methods in typical class implementations!!!!

Methods: Reset and Clear

The `Reset` and `Clear` *Methods* perform similar but different functions... The `Reset` method is responsible for putting the *Value* instance in the **Null** state while the `Clear` method is responsible for putting the *Value* instance in the **Empty** state.

In both the **Empty** and the **Null** states, the instance has no *Contents.* But there's two natural notions of "emptiness" and the distinction limited to *contents*, it has to do with **Identity** as well...

In the **Empty** state, the instance DOES retain its *Identity.* If the implementation of the instance's *Identity* requires resources (such as memory, disk space, et cetera) then there still external resources allocated to the instance.

Take, for example, an instance of a typical `File` class... A `File` identity may consist of an `Id` member perhaps, and a `Name` member implemented as a `String`. In the **Empty** state, the `Id` member would contain a non-**Null** value and the `Name` member could still contain a full file name and path. So, the `Name` member might require externally allocated memory.

That is different from the **Null** state, where ALL external resources are released – or have not been allocated yet.

So, in our theoretical `File` class, when an instance is in the **Null** state it would have `Id` set to **Null** and `Name` set to **Null** resulting i no external resources are needed. So, the only memory required for an in-memory instance of our `File` class would be the fixed number of bytes needed to contain the **Null** version of `Id` and `Name` which, in our case, are negligible.

• • •

A typical `Reset` method would simply invoke the class's `StartUp` method followed by the `ShutDown` method and return the combined `Status` result of the two calls if there were any errors.

> **Note:** See earlier where we discussed our recommended standard `Status` return result order.

A typical `Clear` method would first save any *Identity* members in local storage. It would then call the `Reset` method, or alternately call `Clear` on each of the `Value` members. And finally, it would restore the *Identity* members from local storage. This retains the instance's identity but puts it into the **Empty** state.

Methods: `MoveFrom` and `CopyFrom`

Now this is where it starts to get fun and **`BEFE`** distinguishes itself from other data models. Please read this section of methods and the next section of methods together (if possible) because the two sets of methods interact a lot...

Basically, most data models seem to be stuck in the "copy world" and haven't got it sorted out in their heads yet.

> **Note:** Granted, C++11 (the most recent C++ ISO standard, as of 12 August 2011 and not implemented everywhere) got mostly there with their "*Rvalue references and move constructors*" but, as one has learned to expect, they got it all wrong... They put it into the constructor for God's sake!!! You'd think those guys who love, and live, to invent arcane notation would have simply added a "move" syntax like "`a <- b`" or something along those lines. Sheesh!!!

Anyway, the BEFE Lifecycle lets you handle the difference between "Copy" and "Move" quite easily.

We've already introduced the **Consumable** state. But, let's start afresh in our description because, sure, it's a runtime setting and NOT a compile time feature but let's just wait until we finish with our **`BEFE`** *Model* and get our *code generation* going, it'll all be there, just like you would expect.

We've invented two methods: `MoveFrom` and `CopyFrom` and what they're meant for is pretty self-explanatory...

The `MoveFrom` method's job is to move all instance data from a source instance to a destination instance and, once successful, to invoke `StartUp` on the source instance. Simple.

The `CopyFrom` method's job is to copy all instance data ("deeply") from a source instance to a destination instance and, once successful, to leave the original source instance intact.

Yes, it's THAT straightforward!!!

> **Note:** In C++ we had to decide between either having a separate procedure to do this or to make these two actual methods. We chose the latter, so `this` becomes the destination instance and the parameter `that` becomes the source instance.
> Sure, we could have named them `MoveTo` and `CopyTo` the other way around but that goes against the normal C++ and Java way of doing copy, which we might as well stick to now. Who knows? Once we get generating the code, there's no reason we can't just add `CopyTo` and `MoveTo` into the **`BEFE` Lifecycle** and just generate them. We'll see.

Now, here's where the "magic" starts... This runtime "*consumability*" flag (we've called it `isConsumable` where we've implemented it) should be checked by `CopyFrom` and, if set, CopyFrom passes the job on to `MoveFrom` instead.

So, basically, `CopyFrom` defers to MoveFrom IF AND ONLY IF `isConsumable` is set.

And, the plot thickens here... Neither CopyFrom or MoveFrom EVER reset the isConsumable flag. That's the job of the methods discussed below.

But then the question arises of "How do I implement a C++ `operator =` in **`BEFE`**?".

The answer to that question is provided by the BEFE separation of imperative and declarative ideas...

Given the declaration of StartUp, ShutDown, MoveFrom, CopyFrom and the Consumability methods described in the next section, the answer is clearly of the imperative kind, meaning it's a "how to" instead of a "what" question.

So, a C++ `operator =` is easy... It simply calls `CopyFrom` followed by `NotConsumable` and because it's C++ and the language prefers exceptions to Status returns (yuck!) we throw away the return result from both those methods. End of story – BUT IT WORKS JUST GREAT!!!

The final issue to be addressed is whether or not EVERY class needs to have an `isConsumable` if they want to implement `MoveFrom` functionality.

Your class doesn't have to have it's own `isConsumable` flag IF AND ONLY IF it has at least one properly working *Value* member whose **Null** and **Consumable** states accurately represent your own instance's state. If your class only has *Primitive Type* members, there's hardly any need to distinguish between `MoveFrom` and `CopyFrom`.

So, with *Value* members the question arises as whether to use "their `isConsumable`", implement our own, or what. What BEFE tends to do (and we don't always do it consistently across the board) is to use one of the Value member's `IsConsumable` method to determine if we're in the **Consumable** state and, if we are, behave accordingly. Clearly, this means our `Consumable` and `NotConsumable` methods have to defer the action to the chosen *Value* member's methods.

Warning: The "gotcha" here is that if you decide to defer the **Consumable** state to one or your *Value* members, you most choose that member carefully because if you ever set it to the **Null** state, its `isConsumable` will be cleared, meaning your ENTIRE instance not consumable. Probably a bad idea all around. So, be careful which *Value* member you chose OR just bite the bullet and implement your own `isConsumable` to do it properly.

The other option, which we haven't figure out exactly why you would want to do it this way, is to use ALL the value member's `IsConsumable` methods, in parallel or series or whatever, and defer to ALL of their `Consumable` and `NotConsumable` methods. But we can't figure out where or when that would be a sane option BUT it IS an option.

• • •

A typical `MoveFrom` method first calls the `Reset` method to clear out any currently allocated resources,, then physically copy the entire bytes of the fixed part of the `that` parameter to `this`, and calls `that.StartUp` to put it into the **Null** state.

A typical `CopyFrom` method first checks to see if it is in the **Consumable** state and, if so, simply calls `MoveFrom` and returns. If it wasn't in the **Consumable** state, it first copies all the contents of the *Primitive Type* members of `that` into the associated member of `this`. Then it calls each *Value Type* member's `MoveFrom` method passing it a reference to the associated member of the `that` parameter.

Methods: `IsConsumable`, `Consumable`, and `NotConsumable`

By this point you probably already have a good idea what these methods are all about but we still need to specify their exact behaviour...

The `Consumable` method is responsible for setting the *Value* instance in the **Consumable** state. It should do this REGARDLESS of instance's **Null**, **Empty**, or **Immutable** (discussed later) states.

Typically, `Consumable` either sets an `isConsumable` member to be 1, or defers the call to one or more of its *Value* members by calling its `Consumable` method instead.

The `NotConsumable` method is responsible for setting the *Value* instance in the Not Consumable state. Once again, it should do this REGARDLESS of the instance's **Null**, **Empty**, or **Immutable** states.

Typically, `NotConsumable` either sets an `isConsumable` member to be 0, or defers the call to one or more of its Value member's by calling its `NotConsumable` method instead.

That's all there is to say about these methods.

> **Note:** An interesting bit of trivia you may want to know about **BEFE**... Even with all the bother of describing `MoveFrom`, `CopyFrom`, and all the `Consumable` stuff, there's only three places in all of **BEFE** that actually implement the `isConsumable` flag... in the `BufferLow` class (an internal and low level implementation of the `Buffer` class), and in the `Integer` and `Dictionary` classes.
> And the last two implementations are both going to be rejigged soon so, when they are, the `isConsumable` flag will only exist in the `BufferLow` class.
> I guess that puts the whole **Consumability** subject into a clearer perspective!

Methods: `IsNull`, `SetNull`, `IsEmpty`, and `SetEmpty`

The purpose of these methods are to manage the `Null` and `Empty` states.

The `IsNull` and `IsEmpty` methods simply return an indicator `True` if the instance is in that state.

The `SetNull` method is responsible for releasing all resources, if any, and placing the instance in the **Null** state if it is not already in that state and the `SetEmpty` method is responsible for placing the instance in the **Empty** state.

The `SetNull` and `SetEmpty` methods are very similar, if not identical in meaning, to the `Reset` and `Clear` methods already discussed. In fact, it's quite common for `Reset` to simply call `SetNull` and `Clear` to call `SetEmpty`.

We have, however, separated these methods to allow the you to apply any subtle and clever semantic differences you may wish to design into your *Value* class – but make sure you stick to the specifications here or you won't be playing the same game as **BEFE**.

> **Note:** It is quite common to implement both **Null** and **Empty** using the same internal flag or value, especially for scalar numeric *Values* like `Age`.
>
> In the case of an `Age`, the concept of "*contents*" doesn't seem to fit very well, so the designer has to look around for a more meaningful notion of **Empty** and **Clear**.
>
> At the same time, **Null** may easily be interpreted to mean "*the Default Value*". So, when "*Clearing* `Age`", it makes absolute sense to set the value to `Null`.

Methods: `IsMutable`, `Mutable`, and `NotMutable`

These methods have to do with managing a Value instance's Mutable state.

The Mutable state determines if the caller is allowed to "change the value" of an instance.

We have put this into the BEFE Lifecycle because, although you don't have to implement it on all your classes, we have found that having Mutability be an attribute of the Type causes unnecessary limits to be place on things in the "real world".

For example **`BEFE`**, unlike Java and Python, doesn't demand that ALL `String` instances are immutable because, sometimes you want them to be and sometimes you don't – let's be a bit fair about it... Sure, you may want a certain sub-class of `String` to be immutable for performance reasons but ALL `Strings`? No, that's way too limiting and we, the developers, have to constantly jump through hoops to pay for that unforgiving restriction. And, certainly in the case of a String, a single bit per instance isn't too much to pay to let them be mutable, is it?

> **Note:** Sure, the OTHER price to pay is to write a lot of code up front to ensure that cases where a `String` should be **immutable** are enforced, and that both cases, **mutable** and **immutable**, are runtime efficient but that's a one off price we've already paid here at BEFE so that YOU don't have to pay it day in and day out by having to code around it.
>
> And, although we thank you for the applause, we actually had great fun doing it so yet another reason to "JUST DO IT"!

The Mutable and NotMutable methods are responsible for switching an instance's mutable state and both return a reference to the Value instance so it can be passed along easily in expressions or as parameters.

> Note: These two methods are `Mutable` and `NotMutable` and, at first, appear to be a slightly different naming convention than we used for the `SetNull` and `SetEmpty` methods.
>
> This is because the mutability state has no other means of setting it like those states do – for example, you can set an instance to a non-`Null` value to clear the **Null** state or populate it with some contents to get rid of the **Clear** state.
>
> With with the **mutable** state there's no clear cut and obvious way to change it with a value. Problem solved by simple naming convention magic!

Methods: `Length` and `Size`

The `Length` and `Size` methods may at first appear to tell you the same thing but they don't...

The `Length` method tells you "how many entries" there are in the value's contents, so it's more of a "logical size", whereas `Size` is a "physical size".

For scalars `Length` typically returns the value 1. For arrays, sequences, maps and the like, `Length` returns you the number of entries in the array, sequence, map or whatever.

The `Size` method is responsible for tell the caller how many **Storage Units** are used by the instance implementation, typically in bytes. We can hardly overstress this... `Size` is in **Storage Units**. We'll discuss that shortly, but first...

So, don't confuse Size's return value with bytes because it is formally in **Storage Units** NOT *bytes*.

So, a `String` containing the value "`abc◄─►def`" would have a `Length` of `9` and a `Size` of `15`. This is because the string is 9 Characters long, but is represented in UTF-8 by 16 bytes each each of the Unicode characters ('◄', '─', and '►') are represented in UTF-8 by three bytes each.

What you DO need to be aware of is that Size returns you the physical size in **Used** *Storage Units* not the storage of theinstance itself OR the total bytes of the memory buffer the bytes are held in. This is a subtle thing that may sound confusing but you'll get used to quickly because the value returned by Size represents the "physical contents" instead of some freek value based on memory allocation or heap management reasons, et cetera.

> **Note:** Typically, in hardened C and C++ developer's minds, they are likely to equate the `BEFE Size` method with C's `sizeof()` function and get confused with the result.
>
> But, it actually IS the same as sizeof() even though it's not immediately obvious... sizeof() doesn't tell you how big the block of memory is that a struct/class is allocated in, does it?

Believe me, I myself had a difficult time working it out in my own head while I was implementing all this stuff so I figured it would be best to warn you about the wrong path one's head can lead them at times.

Some classes in the BEFE implementation expose a PhySize method to return the physical size but that turns into a Long which expresses the Physical Size in bits, which 64 bits is well enough to capture for anything you're likely to come across in your lifetime.

• • •

Now, a few more sharp words about the distinction between bytes and **Storage Units** because, even though it doesn't come directly into play at this point in time, we'll make a bold assertion/declaration:

> A **Class** is an implementation of a **Type** for a given type of **Storage Mechanism**(s) and on a given **Platform**(s).

That may seem like a weird assertion but let's play with it for a while...

BEFE Lifecycle

Let's say you designed a `BitArray` class, and an instance of it had 13 bits in the array. Then the `Length` method would return 13 and `Size` would return 2.

But that's only if it the `BitArray` Class is designed for fairly small in-memory arrays of bits. Suddenly, bringing persistence and storage into the equation starts raising some ugly issues and starts causing problems...

If you're going to design a "disk based" `BitArray` implementation, you'll have to start thinking about addressing the `BitArray` within a `File`, or perhaps as a file system sector (what Microsoft calls a "cluster").

So, you're now in the position to decide, if your BitArray implementation is going to handle "really big" ones, how you're going to address them on disk, et cetera.

Let's say you decide that you can afford to start each persistent `BitArray` at a "disk block boundary", where your disk blocks are 4096 bytes each. That means that with a base block address of 32 bits, you've now got 16 terabytes to play with (because the 32 bits are now a block address not a byte address). So, somewhere in that file, and on a 4096 byte boundary somewhere, your specific `BitArray` starts. Cool, that seems well worth the limitation of putting them each on a block boundary.

Great so far, BUT the question of "what is the `Size` of the `BitArray`?" needs to be answered, even if it still only has 13 bits in it.

In that case `Length` would still return 13, but `Size` would return 1 because it's "one block big" even though it only contains 13 bits.

This means your `BitArray` has used 1 *Storage Unit* of space, even though you may have pre-allocated a bunch of contiguous blocks in the file so you could quickly append to your bit array – that's where, once again, the `PhySize` method and others might come in but those aren't part of the **`BEFE` Lifecycle**, at least not yet because we haven't finished our **Persistence Model** at the time of writing

Anyway, that's why with `Size` we distinguish between bytes and **Storage Units**. But it sure can mess with your mind at times.

Methods: `CompareEquals` and `CompareRelative`

These two methods are present in the BEFE Lifecycle to support instances "comparison" which includes, but is not limited to the following two main questions...

The `CompareEquals` method, if implemented, is responsible for answering the simple question "Are the contents of `X` the same as the contents of `Y`?".

The `CompareRelative` method, if implemented, is responsible for not actually answering a question, but performing an operation "Compare the relative order of `X` and `Y`".

We say this is "more complex" but actually, it's just a different question than the other one and we're not going to get into the deep semantics or philosophy of it right here and now.

Needless to say, sometimes you want one and sometimes you want the other. Just be aware that the `CompareRelative` method is potentially more costly.

> **Note:** The `String` class has added several similar methods to the soup by introducing the concept of **case sensitivity** into the comparison question. But those aren't officially part of the **`BEFE` Lifecycle** proper, so you can look at the **BEFE – Class: String** *Grey Paper* to understand how `String` comparison works.

Methods: `ToString` and `ToStrings`

Enough BEFE Classes have implemented either or Both of these methods that we decided they should be thrown into the BEFE Lifecycle because they're so handy.

The `ToString` method, if implemented, is responsible for turning the instance into some kind of one line displayable form and returns a `String` instance.

> **Note:** This returns a non-`Status` value but we figure that's okay because `ToString` is not really an operational or "active" part of an instance's **Lifecycle**, it is more like a "handy convenience" with no particular effect on the instance's operation. However, should we change our minds we'll let you know.

The `ToStrings` method, if implemented, is responsible for turning the instance into some kind of multi-line displayable form and appending "lines" to a passed Strings parameter in which each entry represents a single line in some the multi-line output.

> **Note:** It's not a good idea to put a '`\n`' at the end of the lines populated in the `Strings` parameter because they're already defined to be lines so will have another '`\n`' output after each one when they are displayed.

Methods: `WriteTo` and `ReadFrom`

*** This page is only here as a PLACEHOLDER until we've finished the **`BEFE` Persistence Model**