# Ecstasy: XVM and the XTC Language

Version: DRAFT-20160826

*Software is music: Elegance in simplicity; harmony in vision; rhythm in motion.*

# Introduction

Imagine a system of execution. The ingredients are well known at a conceptual level: Instructions, conditionals, control flow, state, access, modification. Processors, virtual machines, languages: Each a tool to support the description of a desired outcome. Processes, threads, fibers, tasks, jobs: Each a picture of execution.

We create systems of executions as abstractions that exist between the definite world of digital processing and the indefinite and limitless world of programming. In the immortal words of Fred Brooks writing in *The Mythical Man Month*:

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

This document describes a system of execution, the XVM[[1]](#footnote-1). It exists in the abstract as defined by this document, and much like similar technologies that have preceded it, it is expected that compliant implementations could be implemented in software, with execution efficiency provided by translation to the execution language of the underlying machine, whether it be hardware or software.

It is this simultaneous duality of purpose – to provide both a natural and immersive abstraction, and to maximize the potential efficiency of execution – that creates a dramatic tension in design. Most computer languages and their associated execution machinery are examples of the necessary trade-offs made to resolve this tension; very few examples exist without evident compromise. At one end of the spectrum are the hardware-centric models, starting with machine code, then assembly code, and progressing into the realm of increasing abstraction with languages such as FORTRAN, COBOL, and C; these programming languages allow the expression of intention using constructs that translate very directly to the mechanics of the underlying machine, thus simplifying the potential for optimization of execution in terms of both time and space. As one proceeds along the language spectrum, abstractions emerge that are less and less a reflection of the underlying machinery, and more and more the attempt to realize concepts of ideal algorithmic and structural composition. These abstraction-centric models include object-oriented languages such as Smalltalk, Java, and C#, and functional languages such as Haskell, Erlang, and Clojure; these programming languages provide powerful abstractions that allow designs and concepts to translate very directly to the mechanics of the language, optimizing for some combination of simplicity, brevity, readability, consistency, conceptual purity, predictability of execution, code re-usability, portability, speed of development, and so on.

There is nothing more crucial in understanding a system of execution than understanding how the system chooses to resolve this natural tension. The two questions that must always be answered are these: (i) How does the system propose to map its capabilities to the hardware instruction set, the hardware memory model, other hardware capabilities, and the operating system(s) within which it will execute; and (ii) how does the system propose to represent its capabilities to the software developer?

To that end, the following arguments and observations are made, in an attempt to capture both the purpose and the rationale behind the decisions that are made to resolve this fundamental tension in this particular system of execution.

## On Hierarchical Organization

Many a software developer has referenced this saying:

*When the only tool that you have is a hammer, every problem begins to look like a nail.*

That is not to imply that a hammer is not useful. If there is one conceptual hammer that – more so than any other – has repeatedly proven its merit for managing – and *hiding* – complexity, that would be the concept of *hierarchy*. File systems are hierarchies. B\*Trees and binary trees and Patricia tries and parse trees are hierarchies. Most documents are internally organized as hierarchies, including the common HTML, XML, and JSON formats. Most graphical user interfaces are modeled as hierarchies. Many programming languages leverage hierarchy to provide nesting, information hiding, scoping, and identity. How is it that such a simple concept can be so universally useful?

First of all, hierarchical organization enables very simple *navigation*. What this means is that at any arbitrary point – called a *node* – in a hierarchy, there is a well-known set of operations that are possible, such as navigating from the current node to its *parent* node, and navigating from the current node to any of its *child* nodes. If a node does not have a parent, then it is the *root* node, and if a node does not have any child nodes, then it is a *leaf* node.

Child nodes are *contained* within their parent node. Each child is uniquely identifiable by its parent, for example by a name or some other unique attribute. A hierarchy is *recursive*; at any point in the hierarchy, from that point down is itself a hierarchy. Since a hierarchy has a single root and is recursive, each node in the hierarchy is uniquely identifiable in the hierarchy by combining the identities of each successive node starting with the root and proceeding down to the node; this identity is the *absolute path* to that node. It is possible to navigate between any two nodes in the same hierarchy by combining zero or more child-to-parent navigations with zero or more uniquely identifiable parent-to-child navigations; the sequence of these steps is a *relative path* between two nodes.

These basic attributes of a hierarchy combine in amazingly powerful ways. For example, since each node is itself the beginning of a hierarchy of any size, it is possible to refer to that entire sub-hierarchy simply by referring to that one particular node; this effectively *hides* the recursive complexity contained – or *nested* – within that node. As a result, it is possible to add, copy, move, or remove a hierarchy of any size simply by adding, copying, moving, or removing a single node.

Using a hierarchy, it is incredibly simple to construct the concept of *scope*. For example, a scope could include only a specific node, or it could include a specific node and all of its child nodes recursively to its *descendent* leaf nodes, or it could include a specific node and its *ancestor* nodes to the root node, or any other combination of inclusion and exclusion that could be described in an unambiguous manner.

These concepts are incredibly simple, yet at the same time incredibly powerful, and are leveraged liberally throughout the XVM, from managing and hiding complexity for developers, to managing memory in an execution context.

## On Predictability vs. Performance

In the course of researching language design, one preterdominant concern emerges: That of execution performance. While many different concerns are evaluated and balanced against each other in a typical design, and while the goal of performance is often explicitly ranked in a position of lesser importance, in reality there is no one goal more considered, discussed, and pursued. Regardless of the importance assigned to performance as a language goal, performance to the language designer is a flame to a moth.

Perhaps it is simply best to admit, up front, that no language can survive – let alone succeed – without amazing feats of performance. Yet performance as a goal tends to conflict with other goals, particularly with respect to manageability, serviceability, and the quality of the abstractions that are presented to the developer.

Beginning programmers often ask: “Is language *A* faster than *B*?” After all, no one wants to be using a slow language, any more than someone would want to buy a slow automobile or a slow computer. Speed is a thrill, and speed in software execution holds no less attraction than a souped-up hot rod with a throaty growl and a body-rumbling ride.

The corollary to that question is “Why is language *B* slower than *A*?” The answers to this question tend to be very illuminating. Take any two languages that compile to and execute as native machine code, and compare their performance for a given task. Despite running on the same hardware, and using the same hardware instruction set, one may be dramatically slower than the other, by a factor of 10%, or 100% (half as fast), or even 1000% (an order of magnitude slower). How is such a difference possible?

The answer lies in translation, specifically in the automated translation of idioms. A language such as C selected idioms that closely represented the underlying machine instructions of the day, which allowed programmers to write simple programs that could be almost *transliterated* from C to machine code. In other words, the language chose as its abstractions the same set of abstractions that the CPU designers were using, or abstractions that were at most one translation removed from the machine’s abstractions. This allowed for very simple compilers, and made it extremely simple to support localized optimization.

A localized optimization is an optimization in the compiled code that can be made using only information about code that is *most local to* the code that is being optimized; in other words, information outside of the scope of the small amount of code being optimized is not even considered. Many optimizations for the C language, for example, are extremely local, such that they can be performed without any information about code outside of a particular expression or statement; it is hard to imagine more localized optimizations.

However, there is a trade-off implicit in achieving such simple and direct optimizations: The abstractions provided by the language are constrained by the abstractions provided by the CPU. As one could rightfully surmise, hardware abstractions tend not to be very abstract, and the abstractions provided by hardware instruction sets tend to be only slightly better. In its early days, the C language was jokingly referred to as “assembly with macros”, because as a language, it was only slightly higher level than assembly itself.

Computing efficiency is often stated in terms of a tradeoff between time (CPU) and space (memory); one can often utilize one in order to optimize for the other, subject to the law of diminishing returns. Unfortunately, there is no single simple metric that captures computing efficiency, but if the trade-off between time and space appeared as a graph with time on one axis and space on the other, it might resemble the shape of the **TODO** Mandelbrot curve *y=1/x*, which most closely approaches the origin at (1,1). If there were a single computing efficiency measurement for a programming language, it could arguably be represented by the closest that this trade-off curve approaches the origin (0,0), which distance could be considered the minimum weighted resource cost. Since efficiency is the inverse of cost, the efficiency measure in our example would be *1/√2*.

TODO

As languages have been designed with the purpose of supporting higher level abstractions for developers to compose solutions from, this hypothetical measure of computing efficiency has tended to drop. TODO time (CPU) and space (memory) necessary to execute programs built in these languages increased dramatically. To understand why, it is important to think of abstractions as a two-sided coin: On one side, we see the benefit of the abstraction, which allows a programmer to work with ever-larger building blocks, while the other side of the coin represents the cost of the abstraction, which is called the *contract*.

TODO building out of atoms analogy?

A programming contract represents a set of non-negotiable responsibilities. To produce usable abstractions, one cannot avoid the growth in the complexity of the corresponding contracts. Fundamentally, this is caused by the encapsulation of complexity (i.e. the abstraction), whose cost is the hiding of internal detail and the predictability of composed behavior (i.e. the contract).

It is the recursive composition of behavior that creates challenges for optimization. While low level optimizations are focused on the creation of more efficient low level code, higher level optimizations rely on explicit knowledge of what portions of a behavior’s contract can be safely ignored as a result of the various effects of those portions being discarded or ignored, and thus the optimizations are able to eliminate the costs of entire aspects of carefully defined behavioral contracts. Examples include the inlining of potentially dynamically-dispatched invocations by determining that the potential for dynamic dispatch is precluded, and the elimination of selective memory fences in multi-threaded programs as a result of escape analysis.

TODO why do Quality abstractions perform poorly

## On God, Turtles, Baloons, and Sandboxes

Wikipedia defines a software sandbox as follows[[2]](#footnote-2):

In computer security, a sandbox is a security mechanism for separating running programs. It is often used to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users or websites, without risking harm to the host machine or operating system. A sandbox typically provides a tightly controlled set of resources for guest programs to run in, such as scratch space on disk and memory. Network access, the ability to inspect the host system or read from input devices are usually disallowed or heavily restricted.

In the sense of providing a highly controlled environment, sandboxes may be seen as a specific example of virtualization. Sandboxing is frequently used to test unverified programs that may contain a virus or other malicious code, without allowing the software to harm the host device.

In the physical world, in which children play with sand, there are two common styles of sandbox. The first is constructed from four equally sized planks, each stood on its long edge to form a square box, and then filled with sand. The second style is typified by a large green plastic turtle, whose “turtle shell” is the removable top that keeps the rain out, and whose “body” which is the hollow bowl that keeps the sand in. Both styles hold sand and allow a child to dig tunnels and build sand-castles, but there is one major difference: When a child tunnels too deeply in the wooden-sided sandbox, the tunnel burrows past the sand and into the soil beneath, while the tunnel depth in the turtle sandbox is strictly limited by the plastic bowl.

Software sandboxes tend to mirror these physical types, in that *the dirt often lies beneath*. In other words, the sandbox attempts to protect the resources of the system, but a determined programmer will eventually be able to find a way through. The only way that a language runtime as a sandbox can ensure the protection of the underlying resources of a system is for the sandbox itself to completely lack the ability to access those resources. The purpose of the sandbox is to defend against privilege escalation:

Privilege escalation is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. The result is that an application with more privileges than intended by the application developer or system administrator can perform unauthorized actions[[3]](#footnote-3).

As a language runtime designer, it is not sufficient to simply distrust the application code itself; one must distrust the entire graph of code that is reachable by the application code, including all third party libraries, including the language's own published runtime libraries, and including any internal libraries that come with the runtime that are accessible. Or, put another way, if there is a possible attack vector that is reachable, it will eventually be exploited. To truly seal the bottom of the sandbox, it is necessary to disallow resource access *through* the sandbox altogether, and to enforce that limit via transitive closure.

But what good is a language that lacks the ability to work with disks, file systems, networks, and network services? Such a language would be fairly worthless. Ecstasy addresses this requirement by using *dependency injection*, which is a form of *inversion of control*. To comprehend this, it is important to imagine the container functionality not as a sandbox, but as a balloon, and our own universe as the primordial example.

Like an inflated balloon, the universe defines both a boundary and a set of contents. The boundary is defined not so much by a location, but rather by its impermeability – much like the bottom of the green plastic turtle sandbox. In other words, the content of the universe is fixed, and nothing from within can escape, and nothing from without can enter. From the point of view of someone within our actual universe, such as you the reader, there is no boundary to the universe, and the universe is seemingly infinite.

However, from outside of the universe, the balloon barrier is quite observable, as is the creation and destruction of the balloon. Religiously speaking, one plays the part of God when inflating a balloon, with complete control over what goes through that one small – and controllable – opening of the balloon.

It is this opening through which dependency injection of resources can occur. When an application needs access to a file system, for example, it supplicates the future creator of its universe by enumerating its requirement as part of its application definition. As a result, an attempt to create a container for the application will require a file system resource to be provided.

And there are two ways in which such a resource can be obtained. First of all, the resource is defined by its interface, so any implementation of that interface, such as a *mock* file system or a fully emulated – yet completely fake! – file system would do. The second way that the resource can be obtained is for the code that is creating the container to have asked for it in the same manner – to declare a dependency on that resource, and in doing so, force its own unknown God to provide the filing system as an answer to prayer.

As the saying goes, it’s turtles all the way down. In this case, the outermost container to be created is the root of the container hierarchy, which means that if it requires a filing system, then the language runtime must inject something that provides the interface of a filing system, and that resource that is injected might even be a representation of the actual filing system available to the operating system process that is hosting the language runtime.

And here we have a seemingly obvious contradiction: What is the difference between a language that attempts to protect resources by hiding them at the bottom of a sandbox container, versus a language that provides access to those same resources by injecting them into a container? There are several differences, but let’s start with an obvious truth: Security in design is hard to get perfectly correct, and even harder to prove the correctness of, so it is important to understand that this is not a design that automatically guarantees security. Rather, this design seeks to guarantee that only one opening in the balloon – and anything that is injected through that opening – needs to be protected, and the reason is self-evident: Transitive closure. By having nothing naturally occurring in the language runtime that represents an external resource, there is simply no surface area within the language runtime – other than the injected dependencies themselves – that is attackable.

Secondly, the separation of interface and implementation means that the implementation of the resource is not visible within the container into which it is injected. (While this pre-introduces a number of language and runtime concepts, the isolated container implementation is known as a Secure Container, which only permits the surface area of the resource injection interface – not of the implementation – to be visible within the container, even by introspection, and which requires injected resources to be either immutable objects or explicitly thread-safe services.)

Third, there is no possibility of native code within an Ecstasy application; native functionality can only exist outside of the outermost container and thus outside of the language runtime itself, and can only be exposed within the language runtime via a resource injected into a container, subject to all of the limits already discussed.

While it is still possible to introduce security bugs via injection, the purpose of the design is to minimize the scope of potential security bugs to the design of the relatively small number of interfaces that will be supported for resource injection, and to the various injectable implementations of those interfaces.

## On Processor Performance

There exists no shortage of opinions on the topic of what aspects are the most important in a system of execution. One voice will claim that only performance matters, while another will suggest that it no longer matters at all. One voice will claim that achieving efficiencies in development is far more valuable, while another will insist that predictability and stability in execution is critical. Opinions morph with time, as the reality of the physical units of execution evolves and the conceptual units of design are ever the more realized in languages and libraries.

Nonetheless the state of the art today bears the hallmark of a path followed far beyond its logical conclusion. In 1977, John Backus raised an early warning in his ACM Turing Award lecture:

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

While programming advances have largely digested and expelled the explicit concerns of store-addressing and word-at-a-time thinking, these advances have been repetitively accreted onto a burial mound whose foundation remains a von Neumann architecture. Perhaps the success of that underlying architecture is the result of natural selection, or perhaps we have only inertia to blame. In any case, the evolution of concurrent multi-processing and distributed systems has stretched the von Neumann architecture past its effective limits. Specifically, it appears that the recent growth in the extent of the now automatically-managed store has occurred at a pace well beyond the increase in performance of the heart of the von Neumann machine: the processor. Whether this imbalance can be rectified by further technological accretion or by the adoption of a fundamentally new execution architecture is yet to be seen, but regardless: The inevitable and predictable increase in performance that has become the opiate of an industry has taken a sabbatical, and may have accepted an early retirement altogether.

There has existed a loose historic alignment in the growth of processor performance, memory capacity, memory throughput, durable storage capacity, durable storage throughput and network throughput. This relatively consistent growth has allowed a general model of assumptions to be perpetuated throughout hardware architectures, operating systems, programming languages and the various resulting systems of execution. Now we find that model to be threatened by the failed assumption that processor performance will increase at a rapid and relatively predictable rate.

To maintain the façade of progress, explicit hardware parallelism has emerged as the dominant trend in increasing processor throughput. Symmetric Multi-Processing (SMP) has a relatively long history in multi-CPU systems, but adoption of those systems was hobbled both by high prices and a lack of general software support. The advent of the World Wide Web propelled multi-CPU systems into the mainstream for servers, but it is the recent, seemingly instantaneous and near-universal commoditization of multi-core CPUs that has finalized the dramatic shift from a focus on processor performance to a focus on processor parallelism. Further compounding the adoption of multi-CPU and multi-core systems are various technologies for Concurrent Multi-Threading (CMT), which enables a single CPU core to execute multiple threads concurrently. In aggregate, since the turn of the millennium, parallelism has increased from one to dozens of concurrently executing threads in an entry-level server, while the performance of an individual processing unit has only increased by only a few times. Looking forward, processor performance is now expected to improve only incrementally, while the level of parallelism appears to be doubling with each new processor generation.

Since overall processing throughput has continued to increase at a dramatic pace not dissimilar from its historic trend, this shift from performance to parallelism could be safely ignored but for one problem: The von Neumann architecture is bound to a individual processing unit, and thus has nearly halted its forward progress in terms of the throughput of a single thread of execution. This means that for the first time in software history, existing programs do not run significantly faster on newer generations of hardware unless they were built to explicitly take advantage of thread parallelism, which is to say unless they were built assuming their execution would occur on multiple von Neumann machines in parallel. Since the art of programming is expressed almost entirely in imperative terms, and since the imperative nature of programming languages is based on the von Neumann architecture, we have managed to accumulate generations of programs and programmers that are hard-wired to a model that has at least temporarily halted its forward progress.

As a result, computing devices are providing increases in processing throughput that can only be consumed by parallelism. It is obvious that this mandates support for parallelism in any new system of execution, but there is a far less obvious implication of critical importance. Parallelism increases throughput only to the extent that coordination is not required by (among) the threads of execution, and coordination is required only for mutable resources that have the potential to be shared across multiple threads of execution. In common modern systems of execution such as Java and C#, explicit parallelism is provided by threads of execution, each representing the state of a single von Neumann machine, but those machines collectively share a single store. Compounding the coordination overhead for the store is the prevalence of automatic management of the store, referred to as Garbage Collection (GC), which unavoidably requires some level of coordination. While GC algorithms have advanced dramatically in terms of parallelism, the remaining non-parallelized (and possibly non-parallelizable) portion of GC is executed as if by a single thread, which is to say that all threads that share the store are halted for that portion of the GC execution. The unavoidable conclusion is that growth in the shared store without a corresponding increase in processor performance will lead to unavoidable and growing pauses in the execution of the parallelized von Neumann machines.

A series of advances in GC algorithms have thus far masked this inevitable consequence, but the advances are already showing diminishing returns, while the upward pressure on the size of the store has not abated and the dramatic progress of processor performance has not resumed. The solution selected by the XVM is several-fold: First, to organize scopes of execution in a hierarchical manner, with allocations occurring at the lowest possible point in the hierarchy; second, to move an allocation up in the hierarchy only when it *escapes* the scope that previously contained it; third, to explicitly differentiate between mutable and immutable data, and to leverage that knowledge to optimize memory management; fourth, to decompose garbage collection hierarchically such that each portion of the hierarchy is responsible for collection within its scope; and fifth, to only halt the progress of scopes of execution whose mutable stores are being garbage-collected.

The decision to provide an explicit execution-localized store is based on several realizations. The first is that very little data escapes its execution locale (such as a thread) relative to the amount of data that is allocated, and so automated memory management can be made far more efficient if only that small portion of data which has escaped needs to be garbage-collected in a coordinated manner. Second, that objects shared across multiple threads of execution often achieve thread safety (i.e. data integrity and functional correctness despite the potential for concurrent modification) through the use of explicit concurrency control, when the use of thread local data – which is thread-safe by definition – would suffice. A common example of this scenario is a data type that is conceptually immutable, but which has unobservable mutability, such as occurs with lazily-deferred idempotent functions that cause unobservable mutation by storing (caching) calculated results. Lastly, it stands to reason that in order to provide reliable and predictable execution, that thread safety is itself a first-order concern, and therefore data which is not safely accessed and/or mutated by concurrent execution must not be permitted to escape its locale in the first place, penalizing by exception only the execution context from which the unsafe data attempts to escape, as opposed to accruing a deferred penalty of unpredictable (and potentially undiagnosable) data corruption.

The concept of localizing allocations in order to localize GC is not new. Systems built around an explicit threading model have employed *escape analysis* in order to determine which allocations can safely be performed using a thread local allocator (such as a slab allocator), and which allocations need to be made from a shared store. This represents a hierarchy with a fixed depth of two: Global and thread local. While a dramatic improvement over a single shared store, it still implies a global stoppage for GC execution of the shared store.

The primary benefit to GC of the localization of the store is that a significant portion of overall GC execution can be localized entirely within each thread. First, each thread can perform its own independent garbage collection without coordinating with any other threads, which is to say that management of a thread-local or execution context-local store can occur without any concurrency control. Additionally, having stores that are localized to each execution context enables a thread to exactly measure and meter – in real time – the amount of memory that is consumed by each execution context. Lastly, a range of optimizations are available to the data managed in a thread-localized store: the memory containing the data can be accessed and manipulated efficiently without any hardware-level concurrency control; native code can be optimized specifically for cache locality; and the explicit mechanisms used by the data type implementation for thread safety can be safely ignored and omitted altogether. Since escaping an execution context is an explicit transition, it is even possible to alter more than just which store contains the data; for example, the transition can be used to switch from a thread-local to a concurrent-safe implementation of the data type, as if in C++ parlance one could replace an object’s *Vtable* itself.

A secondary benefit is that machine code optimized for single-threaded execution has dramatic performance advantages compared to machine code that is concurrency-safe, even when concurrency control is optimized using the latest hardware capabilities, such as compare-and-swap (CAS) and other “lockless” instructions. When a programmer makes a conscious decision to utilize only a single thread, or to localize mutable data structures each to a single thread of execution, it allows the resulting execution to approach the theoretical maximum efficiency of the underlying hardware.

The concept of GC optimizations based on immutability is also not new. Several GC implementations have leveraged memory protection barriers (protection faults), for example to protect memory regions being compacted, as if the data were immutable, allowing application execution to proceed (in the absence of a fault) while the GC operated concurrently. Significantly, explicitly immutable data can be compacted without protection, because both the old (pre-compaction) and new (post-compaction) copies of the data are valid – being identical! – enabling the application to continue to execute concurrently and correctly while referring to either copy arbitrarily, deferring the housekeeping task of updating any pointers that point to the lingering old copy, and deferring the recycling of the memory region that was compacted. As an added benefit, GC of regions of data known to be immutable can be performed by any thread – including any application thread, or even a separate thread dedicated to GC.

## On Immutability

In an object-oriented system, immutability refers to the prohibition to alter the state of an object. It turns out that many data types are naturally immutable; consider the number 42[[4]](#footnote-4) for example – it is always the number 42! Other data types are naturally mutable; consider the concept of a *variable* for example – its very purpose is to be able to vary! Many data types are naturally immutable, and with mutable data types, it is desirable to be able to make specific instances immutable.

The XVM explicitly supports immutability. Immutability has several benefits, notably: Predictability, thread safety, security, and available optimizations. Predictability is one of the greatest benefits of good design, and immutability supports predictability by preventing values from changing if they’re not supposed to change. For example, when an object exposes its state, it often does so by exposing immutable data types so that its internal state cannot be directly altered. Without explicit support for immutability, one of two things occurs: Either the mutable state of the object is exposed, which breaks encapsulation, or a copy (or other representation) of the mutable state is created on demand and exposed, which is expensive in terms of both space and time, not to mention complexity. Immutability provides a simple way to ensure that the state of an object *cannot* change, addressing each of these concerns.

With respect to thread safety, an immutable object that escapes a local execution context potentially becomes visible to more than one thread of execution, but it can be safely used without concurrency control and without concern for memory barriers, because *no* thread can alter it. For the same reason, the use of immutable objects as the basis for communication among threads is a widely adopted and preferred approach.

Using immutability for security is powerful, but it is important to understand that security as a topic is simply another facet of predictability, and as a result, the same concepts and conclusions apply. Specifically, when immutability is an intrinsic aspect of a system, and thus cannot be circumvented, it becomes a powerful tool for making assumptions about how aspects of the system will operate, and those assumptions become trusted building blocks for building secure software.

Regarding optimizations, the explicit knowledge of immutability conveys a number of significant advantages to a system of execution. As described previously, for example, explicitly immutable data enables a number of potential optimizations for the purpose of garbage collection, and immutability allows certain concurrency controls to be optimized out. Immutability also supports both pass-by-reference and pass-by-value transparently, as the underlying value itself is immutable and the creation of duplicate copies of the value has no negative consequence.

While strict and complete immutability of an entire object is a desirable capability of the language runtime, there are two other related capabilities worth enumerating. The first is support for lazily initialized state as part of an otherwise-immutable data structure, specifically, by means of a function with presumed-idempotent behavior. Such a capability allows the evaluation of time-expensive computations and/or the allocation of space-expensive data structures to be deferred until actually requested. One common example is the hash function calculation for complex data types, which is often assumed to be expensive enough to defer, but which result once computed should be saved for subsequent use.

The second capability is language-level support for designing a non-mutating reference to an underlying data structure that may itself be mutable. In other words, it is desirable to be able to support multiple references to the same object, with one such reference explicitly omitting mutating operations and the exposure of mutable state. There are many examples of mutable data that must be generally protected from mutation, but which the owner of the data may need to mutate; the secondary reference is called a *read-only* reference. This capability does not rely on immutability, but rather relies on the careful design of a programmer, and thus does not provide the types of guarantees that actual immutability can confer. However, with careful use, it is a useful tool for selectively hiding mutability.

## On References

The family of languages influenced by C++ share an implicit trait: A compile-time knowledge of accessibility. For example, it is the C++ compiler that prevents access to private members, and the compiler that allows access for a friend. Subsequent languages, like Java, built on this model, and carry sufficient information in the compiled form of the language to allow the accessibility implied by the compiled code to be validated for a second time at runtime, to prevent an end-run around the language’s security features. One by-product of this design is the ability to use a single pointer as the identity of an object, and more specifically, a pointer that is – in C++ parlance – a Vtable\*\* (a pointer to a pointer to a virtual function table).

From a mechanical-simplicity and efficiency standpoint, the benefits of this model are difficult to overstate. However, there are several specific costs to account for as well. First, the type system defines accessibility in a static manner, predicted on the class of the referrer and the class of the referent, and how those two classes relate. Consider C++ protected members, which are accessible to sub-classes, or Java package-private members, which are accessible to any class within the same package. Second, the facets that a class exposes are statically fixed, such as the set of public members, allowing any referrer that obtains a reference to exploit runtime capabilities such as Java’s *reflection* to fully examine public members, access any public data fields, and invoke any public methods. While allowing arbitrary access to public members does not seem at first to be a concern, it can be; consider that members of an interface are always public in an implementing class, which leads to the third issue: It is not possible for an object to selectively expose interfaces to itself.

This lack of selective exposure can lead to complexity, particularly when the goals of rich functionality and security collide. One need look no further than Java’s serialization capabilities, in which an object must be able to expose its state to a serialization mechanism, but that same state – often private in nature – must be protected from all other referrers. The solution to these conflicting goals was to create exceptions to the rules. Unfortunately, each exception to a rule will grow the complexity of the system, and faults inevitably emerge from complexity.

TODO requires its members exposed by the object must always be composed of public members, which means that resonate as terrifying seem natural and , it is .. TODO .. making exceptions to the static accessibility model is

Ecstasy employs a different model entirely.

## On Types vs. Classes

## On Access Modifiers

O

## On Isolation

Similarly, the XVM relies on the concept of strong isolation to further divide what would be a global store into a hierarchy of stores. Among other things, the purpose of isolation is to limit the effects of a single thread or portion of the system of execution from negatively impacting other threads or portions of the system, and to limit the access from one thread or portion of the system to information managed by another thread or portion of the system. The XVM execution system is divided into one or more *execution contexts*; an execution context represents an isolatable unit with a life cycle, which has an isolated store, and which has its own type system. Each thread in the XVM is created within and belongs to an execution context, and is isolated within that execution unit.

Each new execution context has a parent execution context, which is the execution context of the thread that created the new execution context. The definition of the type system for an execution context is provided by its parent execution context. In this manner, an execution context can introduce and alter data types at runtime, but only within nested (and thus isolated) execution contexts.

Isolation also provides the means to limit the memory or other resources used by a portion of the execution system. This is a critical capability for any number of systems that host multiple applications, or host applications on behalf of multiple parties. Being able to limit resource utilization is a key aspect of providing predictable and reliable execution in a shared environment.

Since isolation within execution contexts means that there is no global store, isolation introduces a small amount of complexity, particularly related to the sharing of information across multiple contexts. The XVM requires that communication between execution contexts occur via message passing, with the messages being immutable. While this may represent an increase in programming complexity, it has the benefit of allowing an execution context to be truly isolated, even to a separate process or machine.

## On Event Modeling

TODO

## On Portability

Similar to the Java Virtual Machine specification, portability is a primary consideration for the XVM. Specifically, the XVM is explicitly designed to be implementable in a particularly efficient manner on modern hardware. While the JVM specification included the definition of a minimal class library, the reality is that there was for all practical purposes a single implementation of that class library. Portability was aided by the use of that single implementation, but the side effect was a reduced ability to optimize the JVM for specific platforms as the result of a single Java implementation for all but a few intrinsic JVM data types (primarily the primitive types).

It is highly desirable to be able to provide a truly minimal set of intrinsic data types in the execution system itself and to be able to implement all other data types using that minimal set, and to deliver those data types as part of a type library that runs on the execution system. Similarly, it is highly desirable to be able to provide an arbitrarily richer set of intrinsic data types as part of the execution system itself, and to be able to be able to implement arbitrary sets of functionality either in the execution system itself or as part of a type library that runs on the execution system. In other words, the ability to trade-off complexity and size versus performance is based on the ability to define standardized capabilities in abstract terms that can be implemented either as an intrinsic part of the execution system or in software that is executed by the execution system.

To accomplish this goal, a rich set of abstract types is defined as part of the XVM specification, and a reference implementation for those types is also specified. Data types within the XVM can freely mix interface and implementation, allowing one XVM implementation to rely almost entirely on the reference implementation of these types (thus requiring only a nominal set of intrinsic types), while another XVM implementation could implement the full set of defined types intrinsically.

Further, the XVM provides this pattern to developers such that any type implementation can be provided in whole or in part in a native form for any set of environments, without requiring a native implementation to be available for other environments – as long as a software (i.e. non-native) implementation is included.

With the advent of the CLI specification, best known in the Microsoft .NET CLR incarnation and to a lesser extent in the open source Mono project, a new facet was added to the concept of virtual machine portability: Explicit multiple-language support. There are inevitable trade-offs in creating an execution system that is intended to support multiple languages; generally, an execution system is optimized for a specific language, and the implementation of any other language trades off between execution performance and the adherence to the “foreign” language specification.

One can conceivably implement any programming language on any Turing-complete execution system, resorting to full runtime interpretation if necessary; the obvious trade-off is correctness versus space/time efficiency. With respect to multiple language support, the primary goal of the XVM is to enable the efficient implementation of a *specific class of language* that can be described by its attributes: object-oriented, composable, single-dispatch, message passing, concurrent, thread-safe, with automatic memory management. Alternatively, the class of language could be described as an incremental evolution of the Java and C# languages, with support for type composition and Software Transactional Memory (STM), and an abolition of primitive types, explicit synchronization, non-virtual constructors and non-virtual state. While it is desirable that other classes of languages be efficiently implementable on the XVM, it is an explicit non-requirement that the semantics of those languages be intrinsically supported by the XVM where those semantics differ from those of the targeted specific class of language.

## On Startup Time

Another side-effect of the core JVM type system being largely implemented on top of the JVM instead of within can be seen in the large number of classes required for the simplest “hello world” application. Just as in the children’s song that iterates over the body a bit at a time, like “the knee bone’s connected to the shin bone” and so on, the JVM is forced to load a relatively large graph of its type system as a result of the cross-dependencies of intrinsic types. Loading a type likely requires I/O, parsing, validation, internal type definition, the execution of any specific type initialization code, and the recursive loading of types on which the loaded type depends. While loading a relatively large graph of a type system is acceptable for a long-running process (which can amortize that loading cost over hours, days or months), it is far more intrusive a cost for a short-running process.

In order to be able to solve this challenge, the intrinsic type system of the XVM is defined only in the abstract. TODO

## On Composite Oriented Programming

## On Footprint

## On Modularity

## On Managing Complexity

## On Openness

Undoubtedly, the largest change in software in the past decade has been cultural and not technological. The term “open” had long been used for marketing commercial software that had some slight yet often only theoretical potential for interoperability with other commercial software. Today, most core software components, libraries, operating systems and applications are available in complete source code form for use under open source or software libré license, and many of the specifications and standards – including languages and execution systems – that enable interoperability are similarly open and available. From an economic standpoint, it would appear that the demand for a fundamental set of software standards and components being available as a public good eventually out-weighed the cost of creating and managing that public good (even in some cases lacking any consistent centralized authority!), and the cost of reverting to private goods for that fundamental set of software standards and components is unacceptable for all but the most especial of requirements.

It is in this spirit that the XVM specification is made available, with its ideas and concepts inspired from others’ open work, and – if any prove worthwhile – its ideas and concepts freely available for re-use and recycling as the reader sees fit.

## Credits

None of these ideas occurred in a vacuum. The initial inspiration for this effort is the Java Virtual Machine, which was ground-breaking in its timely fusion of brilliance and pragmatism, and which is described in beautiful detail by the Java Virtual Machine Specification and the related Java Language Specification. Many concepts of the XVM will also be familiar to Smalltalk programmers, who to this day righteously espouse that all subsequent programming advances were already present in and perfected by Smalltalk, which unfortunately met its untimely demise as the result of a vast right-wing conspiracy. More recently, languages such as C#, Ruby and Scala have stretched our imaginations, with execution systems such as the Sun Hotspot JVM and Microsoft CLR managing to occasionally exceed even the scope of imagination with their ingenuity.

# Types

A system of execution has at its core a concept of data types. The XVM relies on a core set of types that are – by self-reference – necessary to bootstrap the XVM and those types themselves; these types, being intrinsic to the XVM, are referred to as *intrinsic types*. The meaning of intrinsic may differ from implementation to implementation of the XVM specification, but it specifically means that these types can be counted on to be present in any compliant XVM implementation.

The XVM provides a pure object-oriented type system, which is to say that it does not include support for non-object types, which are also known as *primitive types*. The lack of a primitive type system is familiar to Smalltalk programmers, and has emerged as a trend in newer languages. There are several reasons for the emergence of pure object type systems, but the fundamental reason is that it dramatically simplifies the languages and runtime environments by reducing the number of intrinsic type systems – typically from two (objects and primitives) to just one (objects). The result is uniformity and simplicity, and thus elegance.

However, there are legitimate technical reasons why primitives still exist in most contemporary languages. First of all, they are compact: While an object often consumes 16 bytes of memory at a minimum, a primitive byte is still one byte. Second, an object is likely to require an additional dereferencing operation to access. Third, it is relatively easy to generate efficient native code for program code that uses primitive types, as if the code had written in a primitive language such as C. Finally, there are native hardware accelerators, such as Single-Instruction Multiple-Data (SIMD) co-processors and CPU instruction sets that require information to be arranged in very explicit form using primitive types.

Even though raw performance concerns have diminished as the processing throughput of CPUs has increased, it is still crucial for the success of an execution system to carefully consider performance trade-offs. Nonetheless, there are two fundamental realizations that enable the abandonment of a primitive type system without concern for an impact on performance. First, the use of runtime profiling information to optimize code has opened new possibilities by illustrating how conceivable-yet-daunting performance optimizations can be safely realized; this trend is successfully and dramatically illustrated by the Sun (now Oracle) Hotspot JVM. Second, in a pure object-oriented type system, primitive types are translated to objects that are immutable and whose identity corresponds only to the represented value; this knowledge unleashes a slew of potential optimizations. For example, multiple copies of the object can be safely created without concern of identity confusion, since any two objects corresponding to an identical primitive value are in fact the same *singleton* object whether or not their location in memory is the same; as a result, it is conceivable to arbitrarily optimize to using a native representation on the stack, in a register, or as an immediate value, thus achieving the same performance benefits in the executing native code as would have been possible with an explicit primitive type system.

The loss of support for a primitive type system without the loss of potential performance is not accidental. In order to achieve this, there are several common programming concepts that must be hidden such as memory location and method of allocation, and several programming concepts that must be made explicit such as immutability and equality (sameness). These concepts are not limited to those types formerly known as primitive; the associated benefits can be realized with any intrinsic or user types that share these same attributes.

## Objects

In a traditional imperative language like C, complex data types are typically represented as *structs*; a struct is considered a complex data type because it is composed of any number of *fields*, each of which has its own data type. It is possible both to instantiate a struct “on the stack” without dynamic allocation of memory, and to instantiate a struct “on the heap” using dynamic allocation of memory. Generally, *pointers* to those structs are used as an efficient mechanism to pass a reference to the struct to a function, to store a reference to the struct in a variable or any data structure, etc., without actually having to copy or move the struct itself. Following a pointer to get to the pointed-to struct is called de-referencing the pointer.

In an object-oriented language like C++ (which itself extends the C language), a *class* is a data type that extends the concept of a struct, but by default hides the fields of the struct as *private* members. Like C, it is possible to pass a C++ object using a pointer to the object – which would require de-referencing the pointer to access the object – but it is also possible to pass a C++ *reference* to the object, thus partially hiding the concept of a pointer. Furthermore, code that is part of the definition of the class is automatically invoked when an instance of the class is constructed and destructed. All of these aspects of C++ are designed to support and encourage encapsulation, which is a key tenet of object-oriented systems. C++ also enables the definition of the class itself to be parameterized using *templates*; these templates are used to extend otherwise-generic class definitions with additional compile-time type safety and type-specific optimizations.

Java removed a number of the C and C++ language capabilities altogether: structs, manipulable pointers, explicit destruction of objects, and control over the memory layout of classes. The result was an elegant compromise that allowed developers to focus on a pure object type system, with a few obvious exceptions, such as:

* The non-object type system represented by primitive values;
* The not-entirely-pure object type system represented by array types, including arrays of primitive values;
* Global functions (functions not related to object instances) retained via the static keyword;
* Non-virtualized object instantiation (the exact type being specified following the new keyword);
* The inability to forcibly stack-allocate objects;
* The inability to forcibly destroy objects;
* The inability to forcibly control object lifetimes, using patterns such as RAII;
* The inability to optimize multiple object instantiation into a single allocations;
* The lack of support for multi-dimensional array types.

Java does not expose pointers to the programmer; instead, Java has *object references*. These act very similarly to pointers, but with a few notable differences:

* An object reference can only b

to Java objects are does have reference to an object does support the On the other hand, Java retained a few interesting aspects of C and C++ that h

## Meta Model

While the intrinsic type system is intended to provide the fundamental building blocks from which the software developer can construct their own arbitrarily elaborate types, it also provides the software developer with a definition of the XVM itself, including the type system; in other words, the type system provides intrinsic types that define itself. The use of a self-referential and self-defining type system is not new, but the extent to which the XVM carries the concept is unprecedented for a .. TODO turtles

TODO inheritance, type hierarchy, mixins / overlays, duck typing

TODO equals() -> how can it be recursive?

Values

# Constants

A significant portion of the

# The XTC Language

The XTC[[5]](#footnote-5) language is the reference language implementation for the XVM.

## Lexical Structure

XTC code is at its most fundamental level a sequence of Unicode code-points, referred to as *characters*. When a sequence of these characters is stored in a file, it is typically stored in an *encoded* format, for example using the UTF-8 encoding. A number of possible encodings exist for sequences of Unicode characters, but the details of those encodings are outside of the scope of this specification. All encodings must support at least the Unicode code-points in the range U+0000 through U+007F, corresponding to the ASCII character set, and encodings may support up to the entire range of legal Unicode code-points, which at the time of writing is U+0000 through U+10FFFF.

The XTC source code being lexically analyzed is called the *Input*. For purposes of lexical analysis, the *Input* is considered to be a character stream, and that stream is read from left to right.

The first stage of lexical analysis tracks the *line number* and the *line offset* in order to identify exact locations within the source code; both the line number and line offset are zero-based. Any *location* within the source can be identified by a combination of the line number and the line offset, while a selection of the source is identified by a starting location (inclusive) and an ending location (exclusive). The location tracking uses the definition of the *LineTerminator* sequence to determine when to increment the line number and reset the line offset:

*LineTerminator:*

U+000A

U+000B

U+000C

U+000D U+000Aopt

U+0085

U+2028

U+2029

Each *RawCharacter* in the character stream is evaluated to determine if it is the beginning of a *LineTerminator* sequence; if it is, then reading past the *LineTerminator* sequence causes the line number to be incremented and the line offset to be reset to zero. Reading any other *RawCharacter* (that is not the beginning of a *LineTerminator* sequence) causes the line offset to be incremented and does not change the line number. Note that in the case of the two-character *LineTerminator* sequence also known as “CR/LF”, the line number and line offset are undefined in between the contiguous characters U+000D and U+000A.

The second stage of lexical analysis translates Unicode character escape sequences into characters. A Unicode character escape sequence can occur at any point within the source, and must specify a legal Unicode code-point.

*UnicodeCharacterEscapeSequence:*

\u *hex-digit* *hex-digit* *hex-digit* *hex-digit*

\U *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit*

Here are a few examples of how specific characters could be encoded using a Unicode character escape sequence:

|  |  |  |
| --- | --- | --- |
| Character | Code Point | Escape Sequence |
| $ | U+0024 | \u0024 or \U00000024 |
| ¢ | U+00A2 | \u00A2 or \U000000A2 |
| € | U+20AC | \u20AC or \U000020AC |
| 𤭢 | U+24B62 | \U00024B62 |

Since the stream is being analyzed as if it is being read left to right, one character at a time, the character that results from a Unicode character escape sequence is not further evaluated to determine if it could be part of another Unicode character escape sequence; for example, the Unicode code-point U+005C is the character ‘\’, but the source “\u005cu005c” translates simply to the six characters ‘\’, ‘u’, ‘0’, ‘0’, ‘5’, ‘c’, and not further to the single character ‘\’. Similarly, since the Unicode character escape sequences are evaluated after the line number and line number processing, the Unicode character escape sequence “\u000a” does not affect the line number; instead, since the escape sequence is six characters long, it increases the line offset by six, even though a subsequent stage of lexical analysis would only encounter a single character. Note that neither a recursively encoded format nor an escapable format is supported; “\uu0024” is translated simply as the seven characters ‘\’, ‘u’, ‘u’, ‘0’, ‘0’, ‘2’, ‘4’, and “\\u0024” is translated as the two characters ‘\’, ‘$’.

The third stage of lexical analysis converts the residual *Input* stream of characters into a stream of *InputElements*, which are *WhiteSpace*, *Comments,* and *Tokens*, prefaced with an optional *BeginningOfFile* serving as Unicode byte-ordering indicator, and terminated with an optional *EndOfFile*; the *WhiteSpace* and *EndOfFile* elements are discarded.

*Input:*

*BeginningOfFileopt InputElementsopt EndOfFileopt*

*BeginningOfFile:*

U+FEFF

*InputElements:*

*InputElement*

*InputElements InputElement*

*InputElement:*

*WhiteSpace*

*Comment*

*Token*

*EndOfFile:*

U+001A

Whitespace is defined as the union of the traditional ASCII whitespace characters and the three Unicode “separator” categories. Here is the entire table of whitespace characters:

|  |  |  |  |
| --- | --- | --- | --- |
| Code Point | Decimal | Name | Description |
| U+0009 | 9 | HT | Horizontal Tab |
| U+000A | 10 | LF | Line Feed |
| U+000B | 11 | VT | Vertical Tab |
| U+000C | 12 | FF | Form Feed |
| U+000D | 13 | CR | Carriage Return |
| U+001C | 28 | FS | File Separator |
| U+001D | 29 | GS | Group Separator |
| U+001E | 30 | RS | Record Separator |
| U+001F | 31 | US | Unit Separator |
| U+001A | 26 | SUB | End-of-File, or “control-Z” |
| U+0020 | 32 | SP | Space |
| U+0085 | 133 | NEL | Next Line |
| U+00A0 | 160 | &nbsp; | Non-breaking space |
| U+1680 | 5760 |  | Ogham Space Mark |
| U+2000 | 8192 |  | En Quad |
| U+2001 | 8193 |  | Em Quad |
| U+2002 | 8194 |  | En Space |
| U+2003 | 8195 |  | Em Space |
| U+2004 | 8196 |  | Three-Per-Em Space |
| U+2005 | 8197 |  | Four-Per-Em Space |
| U+2006 | 8198 |  | Six-Per-Em Space |
| U+2007 | 8199 |  | Figure Space |
| U+2008 | 8200 |  | Punctuation Space |
| U+2009 | 8201 |  | Thin Space |
| U+200A | 8202 |  | Hair Space |
| U+2028 | 8232 | LS | Line Separator |
| U+2029 | 8233 | PS | Paragraph Separator |
| U+202F | 8239 |  | Narrow No-Break Space |
| U+205F | 8287 |  | Medium Mathematical Space |
| U+3000 | 12288 |  | Ideographic Space |

Since the longest possible translation is used at each stage of lexical analysis, *WhiteSpace* and *Comments* are often necessary for separating two tokens that could otherwise be translated as a single, larger token.

*WhiteSpace:*

*WhiteSpaceElement*

*WhiteSpace WhiteSpaceElement*

*WhiteSpaceElement:*

*SpacingElement*

*LineTerminator*

*SpacingElement:*

U+0009

U+001C

U+001D

U+001E

U+001F

U+0020

U+00A0

U+1680

U+2000

U+2001

U+2002

U+2003

U+2004

U+2005

U+2006

U+2007

U+2008

U+2009

U+200A

U+202F

U+205F

U+3000

*LineTerminator:*

U+000A

U+000B

U+000C

U+000D U+000Aopt

U+0085

U+2028

U+2029

*Comment:*

*EndOfLineComment*

*EnclosedComment*

*EndOfLineComment:*

// *InputCharactersopt*

*InputCharacters:*

*InputCharacter*

*InputCharacters InputCharacter*

*InputCharacter:*

*RawCharacter* except *LineTerminator*

*EnclosedComment:*

/\* *EnclosedCommentTail*

*EnclosedCommentTail:*

\* *AsteriskCommentTail*

*NotAsteriskCharacter EnclosedCommentTail*

*AsteriskCommentTail:*

/

\* *AsteriskCommentTail*

*NotAsteriskOrSlashCharacter* *EnclosedCommentTail*

*NotAsteriskCharacter:*

*RawCharacter* except \*

*NotAsteriskOrSlashCharacter:*

*RawCharacter* except / or \*

*RawCharacters:*

*RawCharacter*

*RawCharacters RawCharacter*

*RawCharacter:*

any legal Unicode code-point except *BeginningOfFile* or *EndOfFile*

It should be obvious that the “control-Z” character used for the *EndOfFile* production, which is an historical anachronism, is only going to be permitted to occur at the end of the file. Similarly, the Unicode byte-ordering indicator character used for the *BeginningOfFile* production is only permitted to occur at the beginning of the file. Note that the *BeginningOfFile* production can be utilized in a number of different Unicode encodings, including UTF-8, UTF-16, and UTF-32 encoded files, and in either little-endian or big-endian byte order for both the UTF-16 and UTF-32 formats, and thus represents a number of different byte sequences that one may encounter at the beginning of a file. Any program operating on XTC source files should recognize and support no less than these five Unicode encodings and the plain ASCII encoding.

Since the stream is being analyzed as if it is being read left to right, and since the *EndOfLineComment* is seeking only for a *LineTerminator* or the end of the stream, any “//”, “/\*”, or “\*/” character sequences encountered while producing an *EndOfLineComment* will be ignored. Likewise, since the *EnclosedComment* is seeking only for the “\*/” character sequence, any “//” or “/\*” character sequences encountered while producing an *EnclosedComment* will be ignored. By logical inference, comments cannot be nested.

TODO

*Token:*

*Identifier*

*Keyword*

*Literal*

*Separator*

*Operator*

*Literal:*

*NumericLiteral*

*StringLiteral*

*NumericLiteral:*

*IntegerLiteral*

*FloatingPointLiteral*

*IntegerLiteral:*

*Signopt UnsignedIntegerLiteral*

*Sign:* one of

+ -

*UnsignedIntegerLiteral:*

*BitLiteral*

*DigitLiteral*

*HexitLiteral*

*BitLiteral:*

0 *BinaryIndicator Bits*

*BinaryIndicator:* one of

B b

*Bits:*

*Bit BitsOrUnderscoresopt*

*Bit:* one of

0 1

*BitsOrUnderscores:*

*BitOrUnderscore*

*BitOrUnderscores BitOrUnderscore*

*BitOrUnderscore:*

*Bit*

\_

*DigitLiteral:*

*Digits*

*Digits:*

*Digit DigitsOrUnderscoresopt*

*Digit:* one of

0 1 2 3 4 5 6 7 8 9

*DigitsOrUnderscores:*

*DigitOrUnderscore*

*DigitOrUnderscores DigitOrUnderscore*

*DigitOrUnderscore:*

*Digit*

\_

*HexitLiteral:*

0 *HexIndicator Hexits*

*HexIndicator:* one of

X x

*Hexits:*

*Hexit HexitsOrUnderscoresopt*

*Hexit:* one of

0 1 2 3 4 5 6 7 8 9 A a B b C c D d E e F f

*HexitsOrUnderscores:*

*HexitOrUnderscore*

*HexitOrUnderscores HexitOrUnderscore*

*HexitOrUnderscore:*

*Hexit*

\_

*FloatingPointLiteral:*

*IntegerLiteral Exponent*

*Signopt FractionalLiteral Exponentopt*

*FractionalLiteral:*

0 *BinaryIndicator Bitsopt* . *Bits*

*Digitsopt* . *Digits*

0 *HexIndicator Hexitsopt* . *Hexits*

*Exponent:*

*ExponentIndicator IntegerLiteral*

*ExponentIndicator:*

*DecimalExponentIndicator*

*BinaryExponentIndicator*

*DecimalExponentIndicator:* one of

E e

*BinaryExponentIndicator:* one of

P p

*StringLiteral:*

" *CharacterStringopt* "

{" *MultilineCharacterStringopt MultilineCharacterStringTail*

*CharacterString:*

*SingleCharacter*

*CharacterString SingleCharacter*

*SingleCharacter:*

*InputCharacter* except \ or "

*CharacterEscape*

*CharacterEscape:*

\ \

\ "

\ b

\ f

\ n

\ r

\ t

*MultilineCharacterString:*

*MultilineSingleCharacter*

*MultilineCharacterString MultilineSingleCharacter*

*MultilineSingleCharacter:*

*InputCharacter* except \ or "

*CharacterEscape*

*MLCharacterString:*

*LineTerminator SpacingElementsopt MultilineContinuation SpacingElementopt CharacterStringopt*

*MultilineCharacterStrings:*

*MultilineCharacterString*

*MultilineCharacterStrings MultilineCharacterString*

*MultilineCharacterStringTail:*

" QuoteMultilineStringTail

*NotQuoteCharacter MultilineCharacterStringTail*

*QuoteMultilineStringTail:*

}

" *QuoteMultilineStringTail*

*NotQuoteOrCurlyCharacter* *MultilineCharacterStringTail*

*NotQuoteCharacter:*

*InputCharacter* except "

*NotQuoteOrCurlyCharacter:*

*InputCharacter* except " or }

*MultilineCharacterString:*

*LineTerminator SpacingElementsopt MultilineContinuation SpacingElementopt CharacterStringopt*

*MultilineContinuation:*

|

+

*SpacingElements:*

*SpacingElement*

*SpacingElements SpacingElement*

## A Brief Introduction to Object Oriented Programming

A number of languages are described as Object-Oriented, which means that the language is intended to allow the structure of a program to be neatly divided into discrete units that each define a specific portion of the program’s data and the logic that is closely associated with that data; these units are referred to as *classes*. To explain some terms courtesy of an overused analogy, consider a dog named Spot and a cat named Fluffy: Spot is an object whose class is dog, while Fluffy is an object whose class is cat. The term *class* represents the definition of the thing, while the term *object* represents one of those things; sometimes the term *instance* is used, as in: Spot is *an instance of* dog.

Now that the obligatory animal analogy has been made, let us instead consider a graphics system on a device or computer, starting with the concept of color. One representation of color is called RGB, in which three separate numeric values for red, green, and blue are mathematically combined into a single numeric value that represents the color; perhaps the simplest implementation of RGB values represents the three color channels as 8-bit integers, and the combined RGB value as a 24-bit integer. In an object oriented language, one can define a class named Color whose properties are Red, Green, Blue, and RGB. The notion of *encapsulation* allows the detail of how that information is stored inside a Color object to be hidden; for example, a Color could have three separate integers that it stores its red, green, and blue information in, or it could have one integer that it stores the combined RGB value in, or it could use some other representation of the data altogether. Regardless, from any point of view outside of the Color class, the *state* of a color object is easily represented by the *properties* of the object in a manner that is consistent with the description of what a color is, such that values of the Red, Green, Blue, and RBG properties are consistent.

Another valuable aspect of object oriented programming is *polymorphism*, which is when different classes of objects share some common programming *interface*, and thus are substitutable to some degree. Extending our previous example, a second class representing color could be created in a manner that would be substitutable in some situations for the first, by providing the same programming interface as the Color class provides, but potentially with a different data structure and implementation. For example, some JPEG and video formats use the YCbCr representation of color that is composed of a luma value (Y), and chroma values for both blue difference (CB) and red difference (CR). Since a YCbCr value can be converted to and from an RGB value, a second Color class (perhaps named YCbCrColor) could be defined that provides properties of Luma, BlueDiff, and RedDiff, but also provides properties of Red, Green, Blue, and RGB – just like the original Color class. Again, as with the previous example, encapsulation allows the class to hide the specific manner in which the object manages its internal information (also called its *state*) behind its programming interface, while exposing that information as both an YCbCr value *and* as an RGB value. Polymorphism allows an instance of the new YCbCrColor class to be used where the program was expecting an instance of the original Color class.

One way in which the YCbCrColor class could have been defined is through the use of *inheritance*, which is when a new class is defined by starting with the definition of an existing class, and then adding things to it. Inheritance is quite powerful, particularly because it allows the re-use of existing class definitions, and because it provides a natural means of supporting polymorphism. In our example, if YCbCrColor is inherited from Color, then it would automatically inherit all of the programming interface of Color, and an instance of YCbCrColor could be transparently used in place of an instance of Color, because YCbCrColor *is a* Color. However, as powerful as inheritance may seem, it has often been perceived to be overused and even abused as a tool; as a result, developers have started to “favor object composition over class inheritance”[[6]](#footnote-6).

While inheritance provides an “*is a*” relationship, composition provides a “*has a*” relationship. For example, instead of the YCbCrColor class inheriting from the Color class, it could simply have a property of the Color class; in other words, the programming interface for YCbCrColor could include a way to obtain a separate object that *is a* Color value, just like it includes a way to obtain Luma, BlueDiff, and RedDiff values.

A third option is to use interface inheritance, which allows the programming interface from an existing class to be re-used while not inheriting other aspects of the existing class definition, such as the manner in which data is structured and managed. For example, a programming interface called Color could be defined that has a single property RGB, whose value is in the form used natively by the device, such as a 24- or 32-bit integer or floating point value. Any number of class definitions could then *implement* the Color interface, thus allowing instances of those various classes to be utilized any time a Color is required. For example, the Color class from our original example could be renamed as RGBColor, and both it and the YCbCrColor class could be made to implement the new Color interface. Similarly, the interface could be implemented by additional class definitions, such as a YUVColor class (the color value used in PAL televisions), and a YIQColor class (the color value used in NTSC televisions).

Conceptually speaking, a programming interface is composed of the accessible *properties* (state) of a class and the *methods* (behavior) that the class exposes; collectively, these are the *members* of the interface. Some programming languages allow any class to act as a substitute for another class, as long as it has the same members as the class that it is substituting for; this is an example of *dynamic typing*, and is often specifically referred to as *duck typing* because “if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.” Other languages require that a class explicitly declare the names of the programming interfaces for which it can act as a substitute; generally speaking, this is an example of *static typing*. However, it is important not to approach terms such as static and dynamic typing too rigidly, because these concepts represent a continuum of possibilities, as opposed to some single binary choice. Furthermore, every programming language has many different aspects, each of which can be located at a completely different point on that continuum.

In object oriented programming, each class is a specific data type; in our examples, each of the various color-related classes and programming interfaces is its own separate data type. When a programmer wishes to create a class that operates on some yet-to-be-specified second class, that second class can be represented by a place-holder, which is called a *type parameter*. A class that has one or more type parameters is called a *parameterized type*, or simply a *generic type*. We have already discussed “*is a*” and “*has a*” relationships; a generic type supports an “*of a*” relationship. Consider the array, which is the most basic of data structures, supported even by assembly languages. An array is a contiguous sequence of a specific number of elements of a specific type, each of which can be loaded from or stored into the array using the element’s index, which is an absolute position in the array. While static typing makes it possible to specify that one must pass a Color to particular method, consider a method that allows any number of Color objects to be passed to it; without generic types, the programmer would have to define the method to accept *any* Array (whose contents are of an unspecified type), or alternatively could create a specific type – perhaps ArrayOfColors – that can hold a number of different colors. A generic type is an alternative to creating a separate specific type (such as ArrayOfColors) for each separate “*of a*” relationship; instead, a single type is defined with a place-holder, such as Array<T> which uses T as a place-holder for some unspecified type. Using a generic type would allow the method in our example to be defined to accept a parameter of type Array<Color>, which is read as “an Array of Color”. The benefit of generic types is the extension of type safety (and any other facilities that rely on or can benefit from type knowledge) to any classes that support one or more “*of a*” relationships. This is also referred to as *parametric polymorphism*, which refers to both the *parameterized type* aspect, and polymorphism, which is a direct benefit of using parameterized types (generic types) as the basis for generic programming.

One of the valuable capabilities provided by inheritance, and perhaps one of the causes for the overuse of inheritance, is the ability to define state and behavior in one class that is then inherited by many different classes. Elimination of redundant code (*a la* “cut and paste”) is a noble goal indeed, and is at the core of the “Don’t Repeat Yourself” (DRY) principle, which states: “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system[[7]](#footnote-7).” In addition to inheritance, you can see this principle at work in programming interface definitions, polymorphism, and generic types. The concept is well described by Steve Smith[[8]](#footnote-8):

Of all the principles of programming, Don't Repeat Yourself (DRY) is perhaps one of the most fundamental. The developer who learns to recognize duplication, and understands how to eliminate it through appropriate practice and proper abstraction, can produce much cleaner code than one who continuously infects the application with unnecessary repetition.

Every line of code that goes into an application must be maintained, and is a potential source of future bugs. Duplication needlessly bloats the code-base, resulting in more opportunities for bugs and adding accidental complexity to the system. The bloat that duplication adds to the system also makes it more difficult for developers working with the system to fully understand the entire system, or to be certain that changes made in one location do not also need to be made in other places that duplicate the logic they are working on.

Repetition in logic can take many forms. Copy-and-paste[..] is among the easiest to detect and correct. Many design patterns have the explicit goal of reducing or eliminating duplication in logic within an application. In fact, the formulation of design patterns themselves is an attempt to reduce the duplication of effort required to solve common problems and discuss such solutions.

A powerful mechanism for achieving DRY in object oriented programming is the *mix-in*, which in many ways is like a class definition, but one that can be “mixed into” (added to) other class definitions. Using the concepts of generic programming, a mix-in *applies to* some yet-to-be-specified second class that provides a specified programming interface, allowing the logic of the mix-in to be constructed around the abilities of that programming interface. Then, any time that logic is required in another class, the mix-in can be applied to that class; while various other mechanisms provide “*is a*”, “*has a*”, and “*of a*” relationships, a mix-in satisfies the “*needs a*” relationship. Continuing our color example, some graphics systems support the notion of color transparency, which is sometimes expressed in the range of 0% to 100%, but is often encoded as an 8-bit value prepended to a 24-bit RGB value, creating a 32-bit integer value. A ColorTransparency mix-in can be defined that applies to any Color, adding a Transparency property to hold a value representing the transparency level, and overriding the RGB property to incorporate the transparency information into the resulting value.

A mix-in that does not define its own state is called a *trait*; a trait is simply a stateless mix-in. As such, a trait can only be used to inject logic (behavior definition) into a class. Using the color example, an Inverting trait can be defined that applies to any Color; the Inverting trait adds an invertedRGB() method that calculates and returns an RGB value that is the inverse of the RGB property value.

This “brief” introduction was not intended to be an exhaustive explanation of object oriented programming; hopefully, it has introduced a number of terms and concepts, and with enough context to provide a basis for understanding the following constructs of the Ecstasy language.

In the end, these concepts are simply mechanisms for organizing functionality together into logical units, in manners designed explicitly to minimize redundancy. Regardless of the language, the computer underneath still executes in the same manner, and its concepts are far simpler, and its mechanisms far fewer in number. Each discrete piece of logic is, at some level, an assembly language function, and accessing that logic is, at some level, nothing more than a call instruction. It is that world that each of these object oriented constructs must translate into, such that every conceivable operation can be expressed as nothing more than a function call, and the magic, if any, is in doing so efficiently without the developer ever being aware of that original intent.

## Ecstasy Types

An Ecstasy *data type* (often referred to simply as a *type*) represents the *interface* of an object. A data type is defined entirely by its state (*properties*) and behavior (*methods*); in other words, a data type can be thought of as a set whose members are property and method signatures. Like a set, a type can be manipulated at runtime in order to create a new type, by adding and/or removing properties and methods. Since each set of methods and properties represents a different type, conceptually there exists an infinite number of types. Any two of these types can be compared in the same manner as one would compare two sets; among other operations, one can compare two types for equality, or to determine if one type is a superset or subset of the other.

While a data type represents only an interface to an object, a *class* is a named *implementation* of a data type; in other words, a class is composed of both a type definition and a type implementation. Ecstasy source code defines classes, and each object at runtime *is an instance of* a particular class. In addition to ordinary classes, Ecstasy supports the definition of specialized forms of classes, including *modules*, *packages*, *values*, *enumerations*, *services*, *traits*, and *mix-ins*. Some classes may be *singleton* classes, and instances of some classes may be *immutable*.

A singleton is a class for which no more than one instance will exist within its runtime context, and which instance can always be obtained by the name of its class.

An immutable object is an object whose state cannot be mutated. Values and enumerations are automatically immutable.

A module is the root of the organization of Ecstasy code. In well-known terms, a library is a module, and an application is a module; additionally, a module can contain other modules, allowing the module granularity to be as fine (small) as desired. In addition to containing other modules, a module can contain packages. As a class, a module can contain any of the elements of a class definition, such as classes, properties, and methods. Finally, a module is a singleton within its runtime context.

Packages provide a hierarchical namespace within a module; if the module is the root, packages are the directories. A package can contain other packages. As a class, a package can contain any of the elements of a class definition such as classes, properties, and methods. A package is a singleton within its runtime context.

A class can contain classes, properties, and methods.

A value is a class that is immutable at the completion of instantiation, and that is automatically endowed with a defined set of properties and behaviors that corresponds to the concept of a value, such as support for comparison of equality.

An enumeration (or *enum* for short) is a singleton value that represents an ordered set of unique singletons values. An obvious example is the class of Boolean values, of which there are two: False and True.

A service is a class that provides for asynchronous (and potentially concurrent) behavior. A service can be declared as a singleton.

A trait is a class that defines behavior that can be added to other classes, and that can also be combined with existing objects at runtime. As a consequence of its ability to be combined with existing objects at runtime, a trait does not and cannot define state of its own.

A mix-in (or *mixin*) is a class that defines state and behavior that can be added to other classes – like a trait – but which cannot be combined with existing objects at runtime.

Compilation and physical packaging can occur at the class, package, or module level; however, the unit of deployment and the unit of versioning is always the module.

## Object References

The type system in Ecstasy is purely a *reference* type system, also known as an object type system. A reference is the means by which one object *refers to* another object. The mechanical aspects of a reference are generally opaque, in terms of the runtime information that composes a reference. Conceptually, a reference is composed of (and represents the composition of) a data type and an object identity.

Consider the number 6[[9]](#footnote-9). In Ecstasy, 6 is a value of a numeric class. As such: 6 is an object; 6 is immutable; 6 has a type; any one of the properties of 6 can be accessed; and any methods of the type of 6 can be invoked. The opaque mechanical aspects of the object 6 include whether or not it is a dynamically allocated object, and if so, whether it is allocated on a stack or from a heap; whether its identity is represented by an address (location in memory), a handle (artificial identity), or by the number 6 itself; whether it carries with it an explicit reference to a specific data type, or whether its data type is implicitly known; and so on. These mechanical aspects are purposefully opaque, which allows the runtime to select a mechanical representation for a reference without the various trade-offs of that selection being surfaced to the developer who is using the number 6.

There are two categories of references: static and dynamic. When the number 6 appears in Ecstasy code, that appearance in the code implies a static reference; in other words, the number 6 is guaranteed to be represented in the compiled form of the Ecstasy code, and as a result, it is available as a reference at runtime. A dynamic reference, on the other hand, represents a reference at runtime that *implies* (but does not *necessitate*) a level of de-reference to obtain.

Dynamic references are represented at runtime by the Ref data type, which provides a RefType property (the data type that the object reference is constrained by), a method get() that returns a reference of that type, and a void method set() that accepts a reference of that type. (It is the existence of this method get() that implies a de-reference for dynamic references.) Each Ref is itself an object, and represents the origination of a reference, while the referenced object (including the data type that the object is referred to as) represents the destination of the reference.

The XVM is an abstract register machine; specifically, the XVM is not an abstract stack machine. In compiler terms, this means that each R-value is either a constant or a register (a Ref instance), and each L-value is a register. A register is used for each function parameter, each local variable, and each temporary result (compiler-generated variable).

There are several critical reasons for this design, but in this context, an explicit design goal was to be able to reduce all operations – including even load and store operations – to explicit sequences of function calls. In reality, and for obvious efficiency reasons, it is not expected that an implementation will strictly adhere to such a *reductio ad absurdum*; however, the result of the design is a dramatically simplified model for expressing predictable runtime behavior, and which allows any of the intrinsic capabilities of the runtime to be replaced or augmented by any fearless Ecstasy programmer.

By having all non-constant references originating from instances of Ref, it is also possible to encapsulate a number of advanced reference capabilities within the same Ref concept. Specifically, behind the Ref interface, and in augmentation to it, it is possible to support hard, soft, weak, phantom, thread-local, lazy, future, and watchable (change-notification) reference functionality; each of these items represents a concern that can be explained in terms of a referrer’s responsibility (or information optionally relevant to the referrer), and not that of the referent.

* @readonly (or @ro) specifies that the property is read-only and cannot be set; a read-only property also allows an object to be made immutable without the read-only property’s value itself being required to be immutable
* @threadlocal (or @tlo) is a thread-local property, which maintains a separate value for the property on behalf of each thread that accesses and/or manipulates the property on this object
* @lazy is a read-only property defers the initialization of a read-only property until its value is requested
* @soft does not prevent the underlying property value from being garbage collected if the running application needs to free up memory, and triggers an event when the value has been garbage collected
* @weak does not prevent the underlying property value from being garbage collected if no other non-weak references to that object remain, and triggers an event when the value has been garbage collected
* @opaque prevents the access of the underlying property value; Nullable properties will return null, and non-Nullable properties will raise an exception on access
* @phantom combines the functionality of the @opaque and @weak annotations
* @watch triggers an event when the property value has been changed

TODO define smallest set: Type property, to<T>() & as<T>() methods

TODO Typedef.

A property signature is composed of the property name and the property type.

A method signature is composed of the method name, the signatures of the method invocation parameters, and the signatures of the method return values. (Method type parameters are not part of the method signature.)

TODO (Discuss: partial binding, target binding, function, parameter binding.)

## Ecstasy Properties

The information that an object contains is called the object’s *state*, and the state is composed of *properties*; a *property* represents a named piece of an object’s state. Properties are sometimes called *instance variables*, and for good reason: A property has many of the same characteristics as a variable in a programming language, in that it has a name, it has a declared type, and it can act as both an L-Value (i.e. a variable, something that can have a value assigned to it) and an R-Value (i.e. a value).

Ecstasy local variables and properties share a common interface, Ref, which represents the combination of an L-Value and an R-Value with a declared type. The behavior of a specific property can be augmented using one or more traits or mix-ins, including several built-in annotations:

* @readonly (or @ro) specifies that the property is read-only and cannot be set; a read-only property also allows an object to be made immutable without the read-only property’s value itself being required to be immutable
* @threadlocal (or @tlo) is a thread-local property, which maintains a separate value for the property on behalf of each thread that accesses and/or manipulates the property on this object
* @lazy is a read-only property defers the initialization of a read-only property until its value is requested
* @soft does not prevent the underlying property value from being garbage collected if the running application needs to free up memory, and triggers an event when the value has been garbage collected
* @weak does not prevent the underlying property value from being garbage collected if no other non-weak references to that object remain, and triggers an event when the value has been garbage collected
* @opaque prevents the access of the underlying property value; Nullable properties will return null, and non-Nullable properties will raise an exception on access
* @phantom combines the functionality of the @opaque and @weak annotations
* @watch triggers an event when the property value has been changed

These annotations can be combined in order to achieve desired behavior; for example a “@threadlocal @weak @lazy” property is one that lazily loads (calculates) a value as needed, which can be discarded by the garbage collector if nothing else holds on to that value, with a separate value being calculated and stored for each thread that accesses the property.

Additionally, certain language operators may be applicable to a property, based on the declared type of the property. Operators that can modify the property’s value are automatically incorporated into the property class itself, allowing the mutating operation to occur in place. These operators are:

* PreIncrement: ++x
* PreDecrement: --x
* PostIncrement: x++
* PostDecrement: --x
* AddAssign: x += …
* SubAssign: x -= …
* MulAssign: x \*= …
* DivAssign: x /= …
* ModAssign: x %= …
* AndAssign: x &= …
* OrAssign: x |= …
* XorAssign: x ^= …
* LShiftAssign: x <<= …
* RShiftAssign: x >>= …
* UShiftAssign: x >>>= …

For example, a boolean property will incorporate the three operators AndAssign, OrAssign, and XorAssign, because the Boolean value type (which is also imported as “boolean”) implements the AndOp, OrOp, and XorOp interfaces.

The Property interface provides a get() method to obtain the value of the property, and methods to modify the value of a property, including set(T) and CAS(T,T). The implementation of the Property class is responsible for providing storage for the object reference that it contains; however, if the get() method of the Property class for a non-@opaque property is unreachable, then no storage will be reserved for the property, and an attempt to store a value in the property will result in an exception. The get() method of the Property class is unreachable if the get() method is overridden without invoking the super function. (Note that while a @lazy property does not provide direct access to the get() method of the Property class, it does invoke that method internally, and thus the get() method of the Property class for a @lazy property is considered to be reachable.)

It is common for a property to have its get() method overridden in order to provide a calculated property, which is a property whose value is derived, for example as a calculation involving the values of other properties. A property that can calculate its value and store the result of that calculation for subsequence accesses is called a lazily calculated property, and is implemented using the @lazy annotation, which can delegate the calculation to a provided function, or whose own calculate() method can be overridden.

In both cases – an overridden get() method that performs a calculation or a @lazy property – the property is considered to be read-only, and attempts to set the property’s value will result in an exception. It is also possible to declare a property as being @readonly, which means that attempts to set the property’s value via the property reference obtained from the public interface of the object will result in an exception.

Each property is public, protected, or private. A public property is part of the public, protected, and private interfaces of the declaring type. A protected property is part of the protected and private interfaces of the declaring type. A private property is only part of the private interface of the declaring type.

While it is possible for the same property to appear in the public, protected, and private interfaces of a the declaring type, it is important to understand that multiple references can exist for the same property, exposing different capabilities to different accessibility levels (public vs. protected vs. private.) Annotating a public property as @readonly, for example, leads to multiple simultaneous implementations existing for the various property-mutating methods such as set(T), such that the property reference in the public and protected interfaces contains an implementations of set(T) that throws an exception, while the property reference in the private interface contains the default implementation of set(T) that stores the passed value. In other words, declaring a property as @readonly makes it read-only to anyone outside of the declaring class, while not explicitly preventing the declaring class from modifying the property’s value itself.

TODO properties show up in a Type as “Property<T> name()”

TODO (To allow sub-classes to mutate the property, a protected set(T value) { super(value); }

Maybe because properties are somehow treated especial .. allowed to have both a public and a protected of the same method?

TODO Similarly, converting a property to @readonly will result in a similar reference to the aforementioned public and protected

TODO e.g. dif between an interface with readonly and a class .. other read-only cases .. private or protected setter .. multiple setters (with public setter throwing) .. ability to cast the object or the property to @readonly

Intrinsic Types

Integer – signed, unsigned, 16- 32- 64- and 128-bit

*If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them.*

1. XVM is an abbreviation for the XTC Virtual Machine [↑](#footnote-ref-1)
2. <https://en.wikipedia.org/wiki/Sandbox_(computer_security)> [↑](#footnote-ref-2)
3. <https://en.wikipedia.org/wiki/Privilege_escalation> [↑](#footnote-ref-3)
4. See <http://hitchhikers.wikia.com/wiki/42>, [↑](#footnote-ref-4)
5. XTC is an abbreviation for XVM Translatable Code, and is also known as “Ecstasy” [↑](#footnote-ref-5)
6. From the book “Design Patterns: Elements of Reusable Object-Oriented Software”, GoF *et al* [↑](#footnote-ref-6)
7. From the book “The Pragmatic Programmer”, by Andy Hunt and Dave Thomas [↑](#footnote-ref-7)
8. With edits, licensed using [Creative Commons Attribution 3](mailto:https://creativecommons.org/licenses/by/3.0/us/) by Steve Smith [↑](#footnote-ref-8)
9. See <http://dilbert.com/strip/1993-12-10> [↑](#footnote-ref-9)