TODO

*Token:*

*Identifier*

*Keyword*

*Literal*

*Separator*

*Operator*

## Imports

An *import* creates a local alias for a structure. An alias is just a name, but names are very important in programming, and in Ecstasy almost everything (no matter how small!) has a name – and that can make for a lot of names! Using hierarchical organization, it’s always possible to identify a structure – whether a package, class, interface, type, method, property, function, and so on – by spelling out the entire hierarchical path to the structure. Such long names, though, become difficult to *read*, and readability is incredibly important. To avoid having to spell out such long names more than once, a developer can *import* the structure in the code that uses it, as if the structure were itself local – but it’s not: all that an import does is create a local alias for that specified structure. Additionally, because popular names tend to collide, the structure can be assigned a different name when it is imported; this is known as an import alias.

*ImportStatement:*

import *QualifiedName ImportAliasopt* ;

*ImportAlias*

as *Name*

Compared to other statements, import statements are also unique in one respect: They are permitted to exist in a source file outside of the scope of the structure being declared in the file:

*CompilationUnit:*

*ImportStatementsopt TypeDeclaration*

*AliasStatements:*

*AliasStatement*

*AliasStatements AliasStatement*

*AliasStatement*

*ImportStatement*

*TypeDefStatement*

Import statements are allowed at the file level in order to allow the type declaration itself to be made more readable. However, since import statements are scoped, they must occur within a scope, and thus they cannot exist at the file level for the module file itself.

## TypeDefs

A *typedef* creates a local alias for a type. TODO

*TypeDefStatement:*

typedef *Type Name* ;

## Source Files and Type Declarations

The purpose of an Ecstasy source file is to contain a *type declaration*, and optionally to contain further type declarations that are nested inside that first type declaration.

*Modifiers:*

*Modifier*

*Modifiers Modifier*

*Modifier:*

*Accessibility*

static

*Accessibility:*

public

protected

private

*TypeCategory:*

module

package

class

interface

service

const

enum

trait

mixin

## 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.

An *interface* declares a set of properties and methods, and can provide default implementations for those. An interface is not itself instantiable, but is useful as a representation of a type, and in the composition of classes.

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*, *constants*, *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. Modules, packages, and enumerations are automatically singletons; constants and services can be singletons.

An immutable object is an object whose state cannot be mutated. Modules, packages, constants, and enumerations are automatically immutable; with the exception of a service, any object can be made immutable at runtime.

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. A module is a singleton constant class; a module can contain any of the elements of a class definition, such as classes, properties, and methods.

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

A *class* can contain classes, properties, and methods.

A constant (or *const* for short) 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 constant value, such as support for comparison of equality.

An enumeration (or *enum* for short) is a singleton constant 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.

## TODO Notes

All Ecstasy code must import the Ecstasy module as if it included the line:

import ecstasy.xtclang.org as x;

That means that every Ecstasy module contains a package “x” containing the Ecstasy language’s class library.

A class declares 0 or more type parameters. Each type parameter has a name and a constraint. If the constraint is not specified in the source code, then the constraint is declared (in the binary structure) as Object.

Super-type type parameters must either be declared or specified by a class. For example, if a class has two direct super-types List<T> and Logger<M>, then it must declare T and M as its own type parameters, or it must specify a type for whichever of those type parameters T and M that it does not declare as its own type parameters.

Since type parameters are named, the super-types’ type parameters names must not collide unless that is the desired outcome, i.e. merging several super-types’ type parameters that happen to have the same name.

A singleton cannot *declare* type parameters, although it can have super-types that declare type parameters if each and every type parameter of its super-type(s) are explicitly specified by the singleton declaration. E.g. “singleton value Names extends List<String> ..”

Modules, packages, and enumerations are singletons, so they cannot declare type parameters.

An enumeration is a combination of an enumeration type (such as Boolean) and singleton values that extend it. An enumeration type itself is non-instantiable – one cannot have a “new Boolean()” or even a singleton instance of Boolean. The only Booleans are the declared values False and True; False and True are “instanceof” Boolean, and no other object can be. (No other class can extend an enumeration.)

## 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[[1]](#footnote-1). 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
* @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 TODO must have a default value (e.g. Nullable type) or be lazy in order to be soft
* @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 @atomic? @transfer?

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

--

Each object has an *inception type*, which is class of the object as it was instantiated, including:

* Optional annotations mixed into the class;
* Type parameters specified for the class;
* The conditional incorporation mixins based on those types parameters.

## Containers

TODO

## Modules

TODO

## Constants

A significant portion of the

## Portable Binary

TODO

## Instruction Set

TODO

TODO explain how objects can hide inside of a reference

TODO explain how one object “body” can be used to provide multiple different objects

TODO explain how objects could be stack based vs. alloc()

TODO explain how objects could be embedded in other objects (example being an array)

1. See <http://dilbert.com/strip/1993-12-10> [↑](#footnote-ref-1)