

ECMA

Standardizing Information and Communication Systems

Common Language Infrastructure (CLI)

Part 1: Architecture

Draft 1 – October 2000

This contribution is being provided “AS IS”, and the SPONSORS EXPRESSLY DISCLAIM ANY AND ALL WARRANTIES REGARDING THIS CONTRIBUTION, INCLUDING ANY WARRANTY THAT THIS CONTRIBUTION DOES NOT VIOLATE THE RIGHTS OF OTHERS OR IS FIT FOR A PARTICULAR PURPOSE.

Brief History

This ECMA Standard was first proposed to ECMA TC39 in October, 2000.

This ECMA Standard has been adopted by the ECMA General Assembly of

Table of contents

1	Scope	5
2	Conformance	5
3	References	5
4	Definitions	5
5	Abbreviations	5
6	Overview of the Common Language Infrastructure	6
6.1	Relationship to Type Safety	6
6.2	Relationship to Managed Metadata-driven Execution	7
6.2.1	Managed Code	7
6.2.2	Managed Data	8
6.3	Introduction to the Common Language Specification (CLS)	8
6.4	Summary	8
7	Type System	9
7.1	Relationship to Object-Oriented Programming	1
7.2	Values and Types	1
7.2.1	Value Types and Reference Types	1
7.2.2	Built-In Types	1
7.2.3	Classes, Interfaces and Objects	2
7.2.4	Boxing and Unboxing of Values	3
7.2.5	Identity and Equality of Values	3
7.3	Locations	4
7.3.1	Assignment Compatible Locations	4
7.3.2	Coercion	4
7.3.3	Casting	4
7.4	Type Members	5
7.4.1	Fields, Array Elements, and Values	5
7.4.2	Methods	5
7.4.3	Static Fields and Static Methods	5
7.4.4	Virtual Methods	6
7.5	Naming	6
7.5.1	Valid Names	6
7.5.2	Assemblies and Scoping	7
7.5.3	Visibility, Accessibility, and Security	8
7.6	Contracts	10
7.6.1	Signatures	11

7.7	Assignment Compatibility	14
7.8	Type Safety and Verification	14
7.9	Type Definers	14
7.9.1	Array Types	15
7.9.2	Pointer Types	16
7.9.3	Interface Type Definition	16
7.9.4	Class Type Definition	17
7.9.5	Object Type Definitions	18
7.9.6	Value Type Definition	20
7.9.7	Type Inheritance	21
7.9.8	Object Type Inheritance	21
7.9.9	Value Type Inheritance	21
7.9.10	Interface Type Inheritance	21
7.10	Member Inheritance	22
7.10.1	Field Inheritance	22
7.10.2	Method Inheritance	22
7.10.3	Property and Event Inheritance	22
7.10.4	Hiding, Overriding, and Layout	22
7.11	Member Definitions	23
7.11.1	Method Definitions	24
7.11.2	Field Definitions	24
7.11.3	Property Definitions	24
7.11.4	Event Definitions	25
7.11.5	Nested Type Definitions	26
8	CLI Metadata	26
8.1	Components and Assemblies	26
8.2	Accessing Metadata	27
8.2.1	Metadata Tokens	27
8.2.2	Member Signatures in Metadata	27
8.3	Unmanaged Code	27
8.4	Method Implementation Metadata	27
8.5	Class Layout	28
8.6	Assemblies: Name Scopes for Types	28
8.7	Metadata Extensibility	30
8.8	Globals, Imports, and Exports	31
8.9	Scoped Statics	31
9	Common Language Specification	31
9.1	Marking Items as CLS-Compliant	32
9.2	Identifiers	32
9.3	Overloading	33
9.4	Operator Overloading	34
9.4.1	Unary Operators	34
9.4.2	Binary Operators	34
9.4.3	Conversion Operators	36

9.5	Naming Patterns	36
9.6	Collected CLS Rules	37
10	Supported Data Types	39
10.1	Natural Size: I, U, O and &	40
10.1.1	Unmanaged Pointers as Type U	40
10.1.2	Managed Pointer Types: O and &	41
10.1.3	Portability: Storing Pointers in Memory	41
10.2	Handling of Short Integer Data Types	41
10.3	Handling of Floating Point Datatypes	42
10.4	CIL Instructions and Numeric Types	43
10.5	CIL Instructions and Pointer Types	44
10.6	Aggregate Data	45
10.6.1	Homes for Values	45
10.6.2	Operations on Value Type Instances	46
10.6.3	Opaque Classes	47
11	Executable Image Information	47
12	Machine State	48
12.1	The Global State	48
12.2	The Memory Store	49
12.2.1	Alignment	49
12.2.2	Byte Ordering	49
12.3	Method State	50
12.3.1	The Evaluation Stack	51
12.3.2	Local Variables and Arguments	51
12.3.3	Variable Argument Lists	52
12.3.4	Local Memory Pool	52
13	Control Flow	52
14	Method Calls	53
14.1	Call Site Descriptors	53
14.2	Calling Instructions	54
14.3	Computed Destinations	54
14.4	Virtual Calling Convention	54
14.5	Parameter Passing	55
14.5.1	By-Value Parameters	55
14.5.2	By-Ref Parameters	55
14.5.3	Typed Reference Parameters	56
14.5.4	A Note on Interactions	56
15	Exception Handling	57
15.1	Exceptions Thrown by the CLI Itself	57
15.2	Overview of Exception Handling	58
15.3	CIL Support for Exceptions	58

15.4	Lexical Nesting of Protected Blocks	59
15.5	Control Flow Restrictions on Protected Blocks	59
16	Atomicity of Memory Accesses	60
17	OptIL: An Instruction Set Within CIL	61
18	Index	62

1 Scope

This ECMA Standard defines an Infrastructure in which applications written in multiple high level languages may be executed in different system environments without the need to rewrite the application to take into consideration the unique characteristics of those environments. This ECMA Standard consists of several parts in order to facilitate understanding various components by describing those components in their separate part. These parts are:

Part 1: Architecture

Part 2: General

Part 3: CIL Instruction Set

Part 4: Base Class Libraries

Part 5: Annexes

2 Conformance

A system claiming conformance to this ECMA Standard shall implement all the mandatory requirements of this standard, and shall specify the profiles that it implements. The minimal implementation shall be the Kernel Profile. A conforming implementation may also include additional functionality. For example, it may provide additional classes, new methods on existing classes, or a new interface on a Standardized class.

A compiler that generates CIL and claims conformance to this ECMA Standard shall produce output files in the format specified in this standard and the CIL it generates shall be valid CIL as specified in this standard. Such a compiler may also claim that it generates *verifiable* code, in which case the CIL it generates shall be verifiable as specified in this standard.

A conforming compiler may claim to be CLS compliant, either as a CLS producer or a CLS consumer. Such a compiler shall generate externally visible types, methods, properties, events, and fields that abide by the restrictions specified for the appropriate level of CLS conformance as specified in this standard, or shall mark them as non-conforming as specified in this standard.

3 References

IEEE 754 standard, "IEEE Standard for Binary Floating-point Arithmetic"

Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-61633-5)

TBD

4 Definitions

For the purpose of this ECMA Standard, the following definitions apply.

TBD

5 Abbreviations

CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLS	Common Language Specification
CTS	Common Type System
JIT	Just In Time (typically abbreviates "JIT Compiler")
OOP	Object-Oriented Programming
VES	Virtual Execution System

6 Overview of the Common Language Infrastructure

At the center of the CLI is a single type system, the Common Type System (CTS), which is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types. The CTS establishes a framework that enables cross-language integration, type safety, and high performance code execution. This part describes the architecture of CLI by describing the CTS.

The following four areas are covered in this part:

- **The Common Type System.** The Common Type System (CTS) provides a rich type system that supports the types and operations found in many programming languages. The Common Type System is intended to support the complete implementation of a wide range of programming languages.
- **Metadata.** The CLI uses metadata to describe and reference the types defined by the Common Type System. Metadata can be stored (“persisted”) in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools that manipulate programs (compilers, debuggers, etc.) as well as between these tools and the Virtual Execution System.
- **The Common Language Specification.** While not strictly part of the CTS, the Common Language Specification is an agreement between language designers and framework (class library) designers. It specifies a subset of the CTS Type System and a set of usage conventions. Languages provide their users the greatest ability to access frameworks by implementing at least those parts of the CTS that are part of the CLS. Similarly, frameworks can be most widely used if their publicly exposed aspects (classes, interfaces, methods, fields, etc.) use only types that are part of the CLS and adhere to the CLS conventions.
- **The Virtual Execution System.** The Virtual Execution System (VES) implements and enforces the CTS model. The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code and data, using the metadata to connect separately generated modules together at runtime (late binding).

Together, these aspects of the CLI form a unifying framework for designing, developing, deploying, and executing distributed components and applications. The appropriate subset of the Common Type System is available from each programming language that targets the CLI. Language-based tools communicate with each other and with the Virtual Execution System using metadata to define and reference the types used to construct the application. The Virtual Execution System uses the metadata to create instances of the types as needed and to provide data type information to other parts of the infrastructure (such as remoting services, assembly downloading, security, etc.).

6.1 Relationship to Type Safety

Type safety is usually discussed in terms of what it does, e.g. guaranteeing encapsulation between different objects, or in terms of what it prevents, e.g. memory corruption by writing where one shouldn't. However, from the point of view of the Common Type System, type safety is about guaranteeing:

- **References are what they say they are** - Every reference is typed and the thing referenced, the definition, also has a type, and they are compatible in a strict sense.
- **Identities are who they say they are** - There is no way to corrupt or spoof an object, and by implication a user or security domain. The access to an object is through accessible functions and fields. An object can still be poorly designed. The key is that a local analysis of the object and the things it uses, as opposed to a global analysis of all uses of an object, is sufficient to understand the vulnerabilities
- **Only appropriate operations can be invoked** – The reference type defines the accessible functions and fields. This includes limiting visibility based on where the reference is, e.g. protected fields only visible in subclasses

The Common Type System promotes type safety e.g. everything is typed. Type safety can be optionally enforced. The hard problem is determining if an implementation conforms to a typesafe declaration. Since the declarations are carried along as metadata with the compiled form of the program, a compiler from the Common Intermediate Language (CIL) to native code (see [Type Safety and Verification](#)) can type-check the implementations. When coupled with code signing, the issue is then when to type-check and when to trust. For more information, see Declarative Security and Custom Attributes for Security in Part 2.

6.2 Relationship to Managed Metadata-driven Execution

Metadata describes code by describing the types that the code defines and the types that it references externally. The compiler produces the metadata when the code is produced. Enough information is stored in the metadata to:

- **Manage code execution** – not just load and execute, but also memory management and execution state inspection.
- **Administer the code** – Installation, resolution, and other services
- **Reference types in the code** – Importing into other languages and tools as well as scripting and automation support.

The Common Type System assumes that the execution environment is metadata-driven. Using metadata allows the CLI to support:

- **Multiple execution models** - The metadata also allows the execution environment to deal with a mixture of interpreted, JITted, native and legacy code and still present uniform services to things like debuggers or profilers, consistent exception handling and unwinding, reliable code access security, and efficient memory management.
- **Auto support for services** - Since the metadata is available at execution time, the execution environment and the base libraries can automatically supply support for reflection, automation, serialization, remote objects, and inter-op with existing unmanaged native code with little or no effort on the part of the programmer.
- **Better optimization** – Using metadata references instead of physical offsets, layouts, and sizes allows the CLI to optimize the physical layouts of members and dispatch tables. In addition, this allows the generated code to be optimized to match the particular CPU or environment.
- **Reduced binding brittleness** – Using metadata references reduces version-to-version brittleness by replacing compile-time object layout with load-time layout and binding by name.
- **Flexible deployment resolution** - Since we can have metadata for both the reference and the definition of a type, more robust and flexible deployment and resolution mechanisms are possible. Resolution means that by looking in the appropriate set of places it is possible to find the implementation that best satisfies these requirements for use in this context. There are five elements of information in the foregoing: two items are made available via metadata (requirements and context); the others come from the application packaging and deployment story (where to look, how to find an implementation, and how to decide the best match).

6.2.1 Managed Code

Managed code is simply code that provides enough information to allow the CLI to provide a set of core services, including

- Given an address inside the code for a method, locate the metadata describing the method
- Walk the stack
- Handle exceptions
- Store and retrieve security information

6.2.2 Managed Data

Managed data is data that is allocated and released automatically by the CLI, through a process called **garbage collection**. Only managed code can access managed data, but programs that are written in managed code can access both managed and unmanaged data.

6.3 Introduction to the Common Language Specification (CLS)

The Common Language Specification (CLS) is discussed in greater detail below (see [Common Language Specification](#)). It is a set of conventions intended to promote language interoperability. Throughout this document, and collected together in a single section (see Collected CLS Rules), there are specific rules that must be followed in order to conform to the CLS. These rules apply only to items that are exposed for use by other programming languages. In particular, they apply to types that are visible in assemblies other than those in which they are defined, and to the members (fields, methods, properties, events, and nested types) that are accessible outside the assembly (i.e. those that have an accessibility of **public**, **family**, or **family or assembly**).

The rules are described in a common format where they are first introduced. For example, the cardinal rule is introduced as follows:

***CLS Rule 0:** CLS rules apply only to those parts of a type that are exposed outside of the defining assembly.*

***CLS (consumers):** no impact.*

***CLS (extenders):** when checking CLS compliance at compile time, be sure to apply the rules only to information that will be exposed outside the assembly.*

***CLS (frameworks):** CLS rules do not apply to internal implementation within an assembly.*

The first paragraph specifies the rule itself. This is then followed by a brief description of how the rule applies in three distinct cases:

- **CLS (consumers):** describes how the rule applies to CLS consumer languages and tools. Consumer languages and tools are designed to allow access to all of the features supplied by CLS-compliant frameworks (libraries). Programmers in CLS consumer languages may not be able to extend these frameworks by creating new types or interfaces, but they can make use of any predefined types.
- **CLS (extenders):** describes how the rule applies to CLS extender languages. Extender languages and tools are designed to allow programmers to both use and extend CLS-compliant frameworks. Programmers can use existing types and define new types and interfaces.
- **CLS (frameworks):** describes how the rule applies to the design of CLS-compliant frameworks. Frameworks (libraries) are designed for use by a wide range of programming languages and tools, including both CLS consumer and extender languages.

6.4 Summary

The Common Type System is about integration between languages: using another language's objects as if they were one's own.

The CLI is all about making it easier to write components and applications from any language. It does this by defining a standard set of types, making all components fully self-describing, and providing a high performance common execution environment. This ensures that all CLI compliant system services and components will be accessible to all CLI aware languages and tools. In addition, this simplifies deployment of components and applications that use them; all in a way that allows compilers and other tools to leverage the high performance execution environment. The Common Type System covers, at a high level, the concepts and interactions that make all of this possible.

The discussion is broken down into three areas:

- Type System – What types are and how to define them.

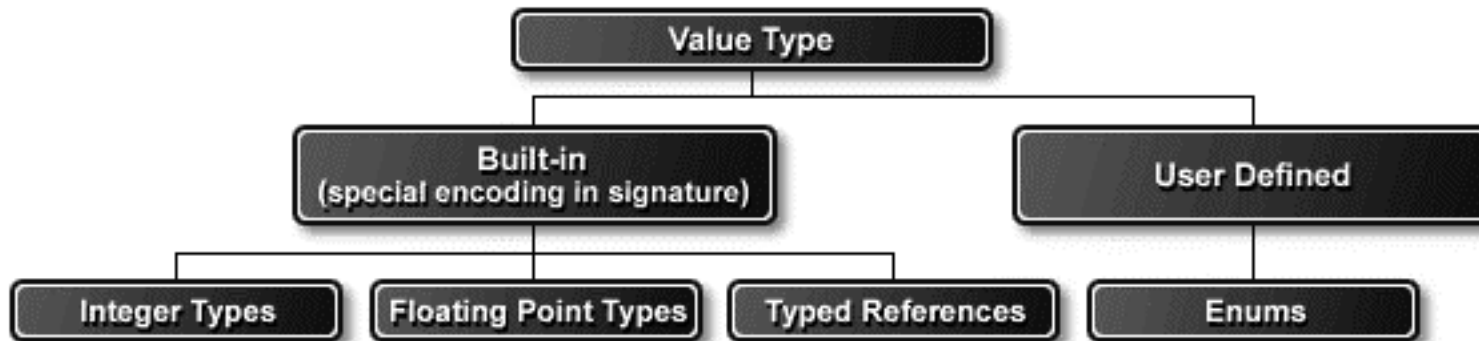
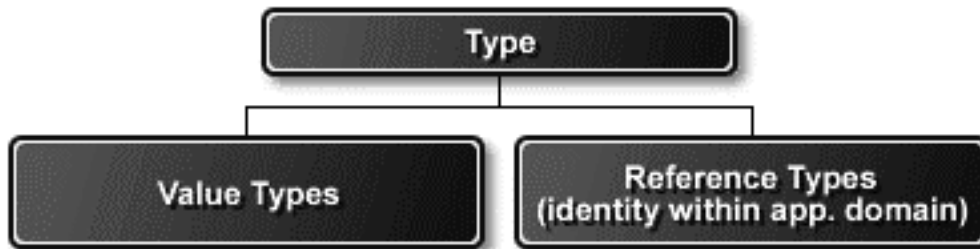
- Metadata - How types are externalized and persisted.
- Virtual Execution System - How code is executed and types are instantiated, interact, and die.

7 Type System

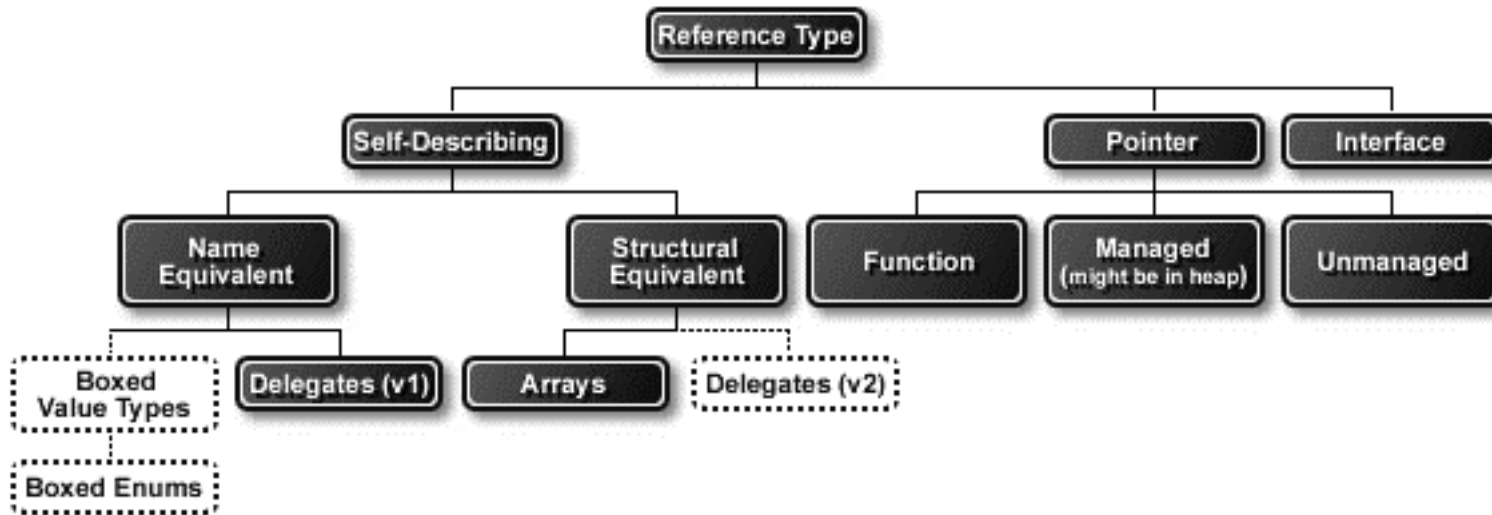
Types describe values and specify a contract (see Contracts) that all values of that type must support. Because the CTS supports Object-Oriented Programming (OOP) as well as functional and procedural programming languages, it deals with two kinds of entities: Objects and Values. Values are simple bit patterns for things like integers and floats; each value has a type that describes both the storage that it occupies and the meanings of the bits in its representation, and also the operations that can be performed on that representation. Values are intended for representing the corresponding simple types in programming languages like C, and also for representing non-objects in languages like C++ and Java.

Objects have rather more to them than do values. Each object is self-typing, that is, its type is explicitly stored in its representation. It has an identity that distinguishes it from all other objects, and it has slots that store other entities (which may be either objects or values). While the contents of its slots may be changed, the identity of an object never changes.

There are several kinds of Objects and Values, as shown in the following diagram.



VOS Reference Types



7.1 Relationship to Object-Oriented Programming

The term **type** is often used in the world of value-oriented programming to mean data representation. In the object-oriented world it usually refers to behavior rather than to representation. In the CTS, type is used to mean both of these things: two entities have the same type if and only if they have both compatible representations and behaviors. Thus, in the CTS, if one type is derived from a base type, then instances of the derived type may be substituted for instances of the base type because **both** the representation and the behavior are compatible.

In the CTS, unlike some OOP languages, two objects that have fundamentally different representations have different types. Some OOP languages use a different notion of type. They consider two objects to have the same type if they respond in the same way to the same set of messages. This notion is captured in the CTS by saying that the objects implement the same interface.

Similarly, some OOP languages (e.g. SmallTalk) consider message passing to be the fundamental model of computation. In the CTS, this corresponds to calling virtual methods (see [Virtual Methods](#)), where the signature of the virtual method serves the role of the message.

The CTS itself does not directly capture the notion of “typeless programming.” That is, there is no way to call a non-static method without knowing the type of the object. Nevertheless, typeless programming can be implemented in the VES by writing a custom message dispatch mechanism and using it directly, generally based on the facilities provided by the types in **System.Reflection**. When interoperating with other languages this dispatch mechanism needs to be packaged into exposed methods so that other languages can access the underlying mechanism. Similarly, the message dispatch mechanism must understand and interoperate with the CTS method mechanism to allow objects implemented in other languages to be accessed.

7.2 Values and Types

Types describe values. All places where values are stored, passed, or operated upon have a type, e.g. all variables, parameters, evaluation stack locations, and method results. The type defines the allowable values and the allowable operations supported by the values of the type. All operators and functions have expected types for each of the values accessed or used.

A value can be of more than one type. A value that supports many interfaces is an example of a value that is of more than one type, as is a value that inherits from another.

7.2.1 Value Types and Reference Types

There are two kinds of types: **Value Types** and **Reference Types**.

- Value Types - Value Types describe values that are represented as sequences of bits.
- Reference Types – Reference Types describe values that are represented as the location of a sequence of bits. There are three kinds of Reference Types:
 - An **object type** is a reference type of a self-describing value (see [Classes, Interfaces and Objects](#)). Some object types (e.g. abstract classes) are only a partial description of a value.
 - An **interface type** is always a partial description of a value, potentially supported by many object types.
 - A **pointer type** is a compile time description of a value whose representation is a machine address of a location.

7.2.2 Built-In Types

The following data types are an integral part of the CTS and are supported directly by the Virtual Execution System (VES). They have special encoding in the persisted metadata:

Table 1: Special Encoding

Name in CIL assembler	CLS Type?	Name in class library	Description
bool	Yes	System.Boolean	True/false value

char	Yes	System.Char	Unicode 16-bit char.
class System.Object	Yes	System.Object	Object or boxed value type
class System.String	Yes	System.String	Unicode string
float32	Yes	System.Single	IEEE 32-bit float
float64	Yes	System.Double	IEEE 64-bit float
int8	No	System.SByte	Signed 8-bit integer
int16	Yes	System.Int16	Signed 16-bit integer
int32	Yes	System.Int32	Signed 32-bit integer
int64	Yes	System.Int64	Signed 64-bit integer
natural int	Yes	System.IntPtr	Signed integer, natural size
natural unsigned int	No	System.UIntPtr	Unsigned integer, natural size
typedref	No	System.TypedReference	Pointer plus runtime type
unsigned int8	Yes	System.Byte	Unsigned 8-bit integer
unsigned int16	No	System.UInt16	Unsigned 16-bit integer
unsigned int32	No	System.UInt32	Unsigned 32-bit integer
unsigned int64	No	System.UInt64	Unsigned 64-bit integer

7.2.3 Classes, Interfaces and Objects

Every value has an **exact type** that **fully describes** the value. A type fully describes a value if it completely defines the value's representation and the operations defined on the value.

For a Value Type, defining the representation entails describing the sequence of bits that make up the value's representation. For a Reference Type, defining the representation entails describing the location and the sequence of bits that make up the value's representation.

A named **method** describes an operation that can be performed on values of an exact type. Defining the set of operations allowed on values of an exact type entails specifying named methods for each operation.

Some types are only a partial description, e.g. **interface types**. Interface types describe a subset of the operations and none of the representation, and hence, cannot be an exact type of any value. Hence, while a value has only one exact type, it may also be a value of many other types as well. Furthermore, since the exact type fully describes the value, it also fully specifies all of the other types that a value of the exact type can have.

While it is true that every value has an exact type, it is not always possible to determine the exact type by inspecting the representation of the value. In particular, it is *never* possible to determine the exact type of a value of a Value Type. Consider two of the built-in Value Types, 32-bit signed and unsigned integers. While each type is a full specification of their respective values, i.e. an exact type, there is no way to derive that exact type from a value's particular 32-bit sequence.

For some values, called **objects**, it *is* always possible to determine the exact type from the value. Exact types of objects are also called **object types**. Objects are values of Reference Types, but not all Reference Types describe objects. Consider a value that is a pointer to a 32-bit integer, a kind of Reference Type. There is no way to discover the type of the value by examining the pointer bits, hence it is not an object. Now consider the built-in CTS Reference Type **System.String**. The exact type of a value of this type is always determinable by examining the value, hence values of type **System.String** are objects and **System.String** is an object type.

7.2.4 Boxing and Unboxing of Values

For every Value Type, the CTS defines a corresponding Reference Type called the **boxed type**. The reverse is not true: Reference Types do not in general have a corresponding Value Type. The representation of a value of a boxed type (a **boxed value**) is a location where a value of the Value Type can be stored. A boxed type is an object type and a boxed value is an object.

All Value Types have an operation called **box**. Boxing a value of any Value Type produces its boxed value, i.e. a value of the corresponding boxed type containing a bit copy of the original value. All boxed types have an operation called **unbox**. Unboxing is the opposite operation to boxing: it copies the bit sequence from the boxed value to create a value of the underlying Value Type.

Notice that interfaces and inheritance are defined only on Reference types. Thus, while a Value Type definition (see Value Type Definition) can specify both interfaces that must be implemented by the Value Type and a class from which it inherits, these apply only to boxed values.

***CLS Rule 1:** Boxed value types are not part of the CLS. Instead, use `System.Object`, `System.ValueType` or `System.Enum`, as appropriate.*

***CLS (consumers):** need not import boxed value types.*

***CLS (extenders):** need not provide syntax for defining or using boxed value types.*

***CLS (frameworks):** must not use boxed value types in their publicly exposed aspects.*

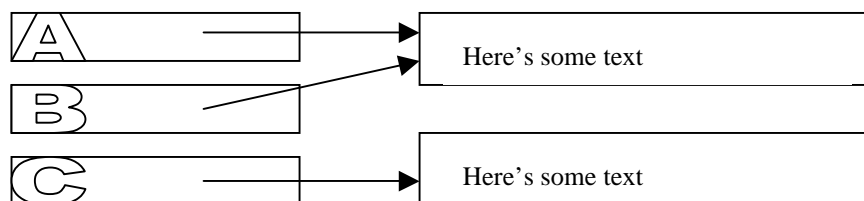
7.2.5 Identity and Equality of Values

There are two binary operators defined on all pairs of values, **identity** and **equality**, which return a Boolean result. Both of these operators are mathematical equivalence operators, i.e. they are:

- Reflexive - $a \text{ op } a$ is true.
- Symmetric - $a \text{ op } b$ is true if and only if $b \text{ op } a$ is true.
- Transitive - if $a \text{ op } b$ is true and $b \text{ op } c$ is true, then $a \text{ op } c$ is true

In addition, identity always implies equality, but not the reverse, i.e., the equality operator need not be the same as the identity operator as long as two identical values are also equal values.

To understand the difference between these operations, consider three variables whose type is **System.String**, where the arrow is intended to mean “is a reference to”:



The values of the variables are **identical** if the locations of the sequences of characters are the same, i.e., there is in fact only one string in memory. The values stored in the variables are **equal** if the sequences of characters are the same. Thus, the values of variables A and B are identical, the values of variables A and C as well as B and C are not identical, and the values of all three of A, B, and C are equal.

7.2.5.1 Identity

The identity operator is defined by the CTS as follows.

- If the values have different exact types, then they are not identical.
- Otherwise, if their exact type is a Value Type, then they are identical if and only if the bit sequences of the values are the same, bit by bit.

- Otherwise, if their exact type is a Reference Type, then they are identical if and only if the locations of the values are the same.

7.2.5.2 Equality

For value types, the equality operator is part of the definition of the exact type. Definitions of equality should obey the following rules:

- Equality should be an equivalence operator, as defined above.
- Identity should imply equality, as stated earlier.
- If either (or both) operand is a boxed value, equality should be computed by
 - first unboxing any boxed operand(s), and then
 - applying the usual rules for equality on the resulting values.

7.3 Locations

Values are stored in **locations**. A location can hold a single value at a time. All locations are typed. The type of the location embodies the requirements that must be met by values that are stored in the location. Examples of locations are local variables and parameters.

More importantly, the type of the location specifies the restrictions on usage of any value that is loaded from the location. For example, a location can hold values of potentially many exact types as long as all of the values are assignment compatible with the type of the location (see below). All values loaded from a location are treated as if they are of the type of the location. Only operations valid for the type of the location can be invoked even if the exact type of the value stored in the location is capable of additional operations.

7.3.1 Assignment Compatible Locations

A value can be stored in a location only if one of the types of the value is **assignment compatible** with the type of the location. A type is always assignment compatible with itself. Assignment compatibility can often be determined at compile time, in which case there is no need for testing at run time. Assignment compatibility is described in detail in [Assignment Compatibility](#).

7.3.2 Coercion

Sometimes it is desirable to take a value of a type that is *not* assignment compatible with a location and convert the value to a type that *is* assignment compatible. This is accomplished through **coercion** of the value. Coercion takes a value of a particular type and a desired type and attempts to create a value of the desired type that has equivalent meaning to the original value. Coercion can result in representation changes as well as type changes, hence coercion does not necessarily preserve the identity of two objects.

There are two kinds of coercion: **widening**, which never loses information, and **narrowing**, in which information may be lost. An example of a widening coercion would be coercing a value that is a 32-bit signed integer to a value that is a 64-bit signed integer. An example of a narrowing coercion is the reverse: coercing a 64-bit signed integer to a 32-bit signed integer. Programming languages often implement widening coercions as **implicit conversions** and narrowing coercions as **explicit conversions**.

Some widening coercion is built directly into the VES operations on the built-in types (see [Supported Data Types](#)). All other coercion must be explicitly requested. For the built-in types, the CTS provides operations to perform widening coercions with no runtime checks and narrowing coercions with runtime checks.

7.3.3 Casting

Since a value can be of more than one type, a use of the value needs to clearly identify which of its types is being used. Since values are read from locations, which are typed, the type of the value that is used is the type of the location from which the value was read. If a different type is to be used, the value is **cast** to one of its other types. Casting is usually a compile time operation, but if the compiler cannot statically know that the value is of the target type, a runtime cast check is done. Unlike

coercion, a cast never changes the actual type of an object nor does it change the representation. Casting preserves the identity of objects.

For example, a runtime check may be needed when casting a value read from a location that is typed as holding values of a particular interface. Since an interface is an incomplete description of the value, casting that value to be of a different interface type will usually result in a runtime cast check.

7.4 Type Members

As stated above, the type defines the allowable values and the allowable operations supported by the values of the type. If the allowable values of the type have a substructure, that substructure is described via fields or array elements of the type. If there are operations that are part of the type, those operations are described via methods on the type. Fields, array elements, and methods are called **members** of the type. Properties and events are also members of the type.

7.4.1 Fields, Array Elements, and Values

The representation of a value can be subdivided into sub-values. These sub-values are either named, in which case they are called **fields**, or they are accessed by an indexing expression, in which case they are called **array elements**. Types that describe values composed of array elements are **array types**. Types that describe values composed of fields are **compound types**. A value contains either fields or array elements, although a field of a compound type can be an array type and an array element can be a compound type.

Array elements and fields are typed. All of the elements of an array type must have the same type. Each field of a compound type may have a different type. The element type of an array type and the field types of a compound type never change.

7.4.2 Methods

A type can associate operations with the type or with each value of the type. Such operations are called methods. A method is named, and has a signature (see [Signatures](#)) that specifies the allowable types for all of its arguments and for its return value, if any.

A method that is associated only with the type itself (as opposed to a particular instance of the type) is called a static method.

A method that is associated with a value of the type is either an instance method or a virtual method. When they are invoked, instance and virtual methods are passed the value on which this invocation is to operate (known as **this** or a **this pointer**).

The fundamental difference between an instance method and a virtual method is in how the implementation is located. An instance method is invoked by specifying a class and the instance method within that class. The object passed as **this** may be **null** (a special value indicating that no particular value is being specified) or an instance of any type that inherits (see [Type Inheritance](#)) from the class that defines the method. A virtual method can also be called in this manner. This occurs, for example, when an implementation of a virtual method wishes to call the implementation supplied by its parent class. The CTS allows **this** to be **null** inside the body of a virtual method.

Rationale: Allowing a virtual method to be called with a non-virtual call eliminates the need for a “call super” instruction and allows version changes between virtual and non-virtual methods. It requires IL generators to insert explicit tests for a null pointer if they don’t want the null this pointer to propagate to called methods.

A virtual method (but not an instance method) may also be called by a different mechanism, a **virtual call**. Any type that inherits from a type that defines a virtual method can provide its own implementation of that method (this is known as **overriding**, see [Hiding, Overriding, and Layout](#)). It is the exact type of the object (determined at runtime) that is used to decide which of the implementations to invoke

7.4.3 Static Fields and Static Methods

Types may declare locations that are associated with the type rather than any particular value of the type. Such locations are **static fields** of the type. As such, static fields declare a location that is shared by all values of the type. Just like non-static (instance) fields, a static field is typed and that type never

changes. Static fields are always restricted to a single application domain basis (see Error! Reference source not found.), but they may also be allocated on a per-thread basis.

Similarly, types can also declare methods that are associated with the type rather than with values of the type. Such methods are **static methods** of the type. Since an invocation of a static method does not have an associated value on which the static method operates, there is no **this** pointer available within a static method.

7.4.4 Virtual Methods

An object type can declare any of its methods as **virtual**. Unlike other methods, each exact type that implements the type may provide its own implementation of a virtual method. A virtual method may be invoked through the ordinary method call mechanism that uses the static type, method name, and types of parameters to choose an implementation, in which case the **this** pointer may be **null**. In addition, however, a virtual method can be invoked by a special mechanism (a **virtual call**) that chooses the implementation based on the dynamically detected type of the instance used to make the virtual call rather than the type statically known at compile time. Virtual methods may be marked **final** (see Method Inheritance).

7.5 Naming

Names are given to entities of the type system so that they can be referred to by other parts of the type system or by the implementations of the types. Types, fields, methods, properties and events have names. With respect to the type system values, locals, and parameters do not have names. An entity of the type system is given a single name, e.g. there is only one name for a type.

7.5.1 Valid Names

The CTS itself imposes two restrictions on names:

1. All names are encoded as 16-bit Unicode strings.
2. Names are not permitted to have an embedded (16-bit) value of 0x0000.

All comparisons are done on a byte-by-byte (i.e. case sensitive, locale-independent, also known as code-point comparison) basis. Where names are used to access built-in VES-supplied functionality (for example, the class initialization method) there is always an accompanying indication on the definition so as not to build in any set of reserved names.

CLS Rule 2: Languages must follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-61633-5) governing the set of characters permitted to start and be included in identifiers, available on-line at <http://www.unicode.org/unicode/reports/tr15/tr15-18.html>. Identifiers must be persisted in their original case so that case sensitive comparisons work as expected. For CLS purposes, however, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they must differ in more than simply their case.

CLS Rule 3: Languages must provide a mechanism to reference identifiers that happen to be keywords in their own language. In addition, extender languages must provide a mechanism for defining and overriding virtual methods with names that are keywords in their language.

CLS (consumer): need not consume types that violate these rules, but must have a mechanism to allow access to named data that uses one of its own keywords as the name.

CLS (extender): need not create types that violate these rules. Must provide a mechanism for defining names that obey these rules but are the same as a keyword in the language. In order to override an inherited definition the CTS requires the precise encoding of the original declaration be used.

CLS (framework): must not export types that violate these rules. Should avoid the use of names that are commonly used as keywords in programming languages.

7.5.2 Assemblies and Scoping

Generally, names are not unique. Names are collected into groupings called **scopes**. Within a scope, a name may refer to multiple entities as long as they are of different **kinds** or have different signatures. In the CTS, the following kinds exist: methods, fields, nested types, properties, and events.

***CLS Rule 4:** All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not.*

***CLS Rule 5:** Fields and nested types must be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) must differ by more than just the return type (although there is a special exception discussed in Conversion Operators).*

***CLS (consumers):** need not consume types that violate these rules after ignoring any members that are marked as not CLS-compliant.*

***CLS (extenders):** need not provide syntax for defining types that violate these rules.*

***CLS (frameworks):** must not mark types as CLS-compliant if they violate these rules unless they mark sufficient offending items within the type as not CLS-compliant so that the remaining members do not conflict with one another.*

A named entity has its name in exactly one scope. Hence, to identify a named entity, both a scope and a name need to be supplied. The scope is said to **qualify** the name. Types provide a scope for the names in the type; hence types qualify the names in the type. For example, consider a compound type `Point` which has a field named `x`. The name “field `x`” by itself does not uniquely identify the named field, but the **qualified name** “field `x` in type `Point`” does.

Since types are named, the names of types are also grouped into scopes. To fully identify a type, the type name must be qualified by the scope that includes the type name. Type names are scoped by the **assembly** that contains the implementation of the type. An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality. The type name is said to be in the **assembly scope** of the assembly that implements the type. Assemblies themselves have names that form the basis of the CTS naming hierarchy.

The **type definition**:

- Defines a name for the type being defined, i.e. the **type name**, and specifies a scope in which that name will be found
 - Defines a **member scope** in which the names of the different kinds of members (fields, methods, events, and properties) are bound. The tuple of (member name, member kind, and member signature) is unique within a member scope of a type.
 - Implicitly assigns the type to the assembly scope of the assembly that contains the type definition.
- The CTS supports a **type distinct alias**, also called an **enum** or **enumeration type**, an alternate name for an existing type. For purposes of matching signatures a type distinct alias is *not* the same as the underlying type. Instances of a type distinct alias, however, are assignment compatible with the underlying type and vice versa. That is: no cast or coercion is required to convert from the alias to the underlying type, nor are they required from the underlying type to the alias. A type distinct alias is considerably more restricted than a true type:
 - It must have exactly one instance field, and the type of that field defines the underlying type of the enumeration.
 - It may not have any methods of its own.
 - It may not specify a base type nor any inherit from any interfaces.

- It may not have any properties or events.
- It may not have any static fields unless they are literal.

The underlying type must be a built-in integer type. Enums must be derived from `System.Enum`, hence they are value types. Like all value types, they must be sealed.

CLS Rule 6: *The underlying type of an enum must be a built-in CLS integer type.*

CLS Rule 7: *There are two distinct kinds of enums, indicated by the presence or absence of the `System.FlagsAttribute` custom attribute. One represents named integer values, the other named bit flags that can be combined together to generate an unnamed value. While languages are encouraged to distinguish these, they need not do so.*

CLS Rule 8: *Literal static fields of an enum must have the type of the enum itself.*

CLS (consumer): *Must accept definition of enums that follow these rules, but need not distinguish flags from named values.*

CLS (extender): *Same as consumer, but should in addition consider allowing the authoring of both kinds of enums for use in other languages.*

CLS (frameworks): *must not expose enums that violate these rules, and must not assume that enums have only the specified values (even for enums that are named values).*

7.5.3 Visibility, Accessibility, and Security

To refer to a named entity in a scope, both the scope and the name in the scope must be **visible**. Visibility is determined by the entity that contains the reference (the referent) and the entity that contains the name being referenced. Consider the following pseudo-code:

```
class A
{ int32 IntInsideA;
}
class B inherits from A
{ method X(int32, int32) returning Boolean
  { IntInsideA := 15;
  }
}
```

If we consider the reference to the field **IntInsideA in class A**:

- We call class B the **referent** because it has a method that refers to that field,
- We call **IntInsideA in class A** the **referenced entity**.

There are two fundamental questions that need to be answered in order to decide whether the referent is allowed to access the referenced entity. The first is whether the name of the referenced entity is **visible** to the referent. If it is visible, then there is a separate question of whether the referent is **accessible**.

Access to a member of a type is permitted only if all three of the following conditions are met:

1. The type is visible.
2. The member is accessible.
3. All relevant security demands have been granted.

7.5.3.1 Visibility of Types

Only type names, not member names, have controlled visibility. Type names fall into one of the following three categories

- **Exportable** from the assembly in which they are defined. While a type can be marked to *allow* it to be exported from the assembly, it is the configuration of the assembly that decides whether the type name *is* made available.
- **Not exported** outside the assembly in which they are defined.
- **Nested** within another type. In this case, the type itself has the visibility of the type inside of which it is nested (its **enclosing class**). See Nested Types.

7.5.3.2 Accessibility of Members

A type scopes all of its members, and it also specifies the accessibility rules for its members. Except where noted, accessibility is decided based only on the statically visible type of the member being referenced and the type and assembly that is making the reference. The CTS supports six different rules for accessibility:

- **Private** – accessible only to referents in the implementation of the exact type that defines the member.
- **Family** – accessible to referents that support the same type, i.e. an exact type and all of the types that inherit from it. For verifiable code (see Type Safety and Verification), there is an additional requirement that may require a runtime check: the reference must be made through an item whose exact type supports the exact type of the referent. That is, the item whose member is being accessed must inherit from the type performing the access.
- **Assembly** – accessible only to referents in the same assembly that contains the implementation of the type.
- **Family and Assembly** – accessible only to referents that qualify for both Family and Assembly access.
- **Family or Assembly** – accessible only to referents that qualify for either Family or Assembly access.
- **Public** – visible to all referents.

In general, a member of a type can have any one of these accessibility rules assigned to it. There are two exceptions, however:

1. Members defined by an interface must be public.
2. When a type defines a virtual method that overrides an inherited definition, the accessibility must either be identical in the two definitions or the overriding definition must permit more access than the original definition. For example, it is possible to override an **assembly virtual** method with a new implementation that is **public virtual**, but not with one that is **family virtual**. In the case of overriding a definition derived from another assembly, it is not considered restricting access if the base definition has **family or assembly** access and the override has only **family** access.

Rationale: Languages including C++ allow this “widening” of access. Restricting access would provide an incorrect illusion of security since simply casting an object to the base class (which occurs implicitly on method call) would allow the method to be called despite the restricted accessibility.

CLS Rule 9: Accessibility must not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility **family or assembly**. In this case the override must have accessibility **family**.

CLS (consumer): need not accept types that widen access to inherited virtual methods.

CLS (extender): need not provide syntax to widen access to inherited virtual methods.

CLS (frameworks): must not rely on the ability to widen access to a virtual method, either in the exposed portion of the framework or by users of the framework.

7.5.3.3 Security Permissions

Access to members is also controlled by security demands that can be attached to an assembly, type, method, property, or event. Security demands are not part of a type contract (see [Contracts](#)), and hence are not inherited. There are two kinds of demands:

- An **inheritance demand**. When attached to a type it requires that any type that wishes to inherit from this type must have the specified security permission. When attached to a non-final virtual method it requires that any type that wishes to override this method must have the specified permission. It may not be attached to any other member.
- A **reference demand**. Any attempt to resolve a reference to the marked item must have specified security permission.

Only one demand of each kind can be attached to any item. Attaching a security demand to an assembly implies that it is attached to all types in the assembly unless another demand of the same kind is attached to the type. Similarly, a demand attached to a type implies the same demand for all members of the type unless another demand of the same kind is attached to the member.

7.5.3.4 Nested Types

A type (called a nested type) can be a member of an enclosing type. A nested type has the same visibility as the enclosing type and has an accessibility as would any other member of the enclosing type. This accessibility determines which other types can make references to the nested type. That is, for a class to define a field or array element of a nested type, have a method that takes a nested type as a parameter or returns one as value, etc., the nested type must be both visible and accessible to the referencing type. A nested type is part of the enclosing type so its methods have access to all members of its enclosing type, as well as family access to members of the type from which it inherits (see [Type Inheritance](#)). The names of nested types are scoped by their enclosing type, not their assembly (only top-level types are scoped by their assembly). There is no requirement that the names of nested types be unique within an assembly.

CLS Rule 10: *Extenders need not allow authoring nested types.*

CLS (consumer): *must accept types that include nested types, and must provide a mechanism for instantiating and manipulating instances of nested types.*

CLS (extender): *need not provide syntax for defining new nested types.*

CLS (frameworks): *may define nested types, but must not rely on the ability to author new nested types in order to use the framework.*

7.6 Contracts

Contracts are named. They are the shared assumptions on a set of **signatures** (see [Signatures](#)) between all implementers and all users of the contract. The signatures are the part of the contract that can be checked and enforced.

Contracts are not types; rather they specify requirements on the implementation of types. Types state which contracts they abide by, i.e. which contracts all implementations of the type must support. An implementation of a type can be verified to check that the enforceable parts of a contract, the named signatures, have been implemented. The kinds of contracts are:

- **Class contract** – A class contract is specified with a class definition. Hence, a class definition defines both the class contract and the **class type**. The name of the class contract and the name of the class type are the same. A class contract specifies the representation of the values of the class type. Additionally, a class contract specifies the other contracts that the class type supports, e.g., which interfaces, methods, properties and events must be implemented. A class contract, and hence the class type, can be supported by other class types as well. A class type that supports the class contract of another class type is said to **inherit** from that class type.
- **Interface contract** – An interface contract is specified with an interface definition. Hence, an interface definition defines both the interface contract and the **interface type**. The name of the interface contract and the name of the interface type are the same. Many types can support an interface contract. Like a class contract,

interface contracts specify which other contracts the interface supports, e.g. which interfaces, methods, properties and events must be implemented. Note that an interface type can never fully describe the representation of a value. Therefore an interface type can never support a class contract, and hence can never be a class type or an exact type.

- **Method contract** – A method contract is specified with a method definition. A method contract is a named operation that specifies the contract between the implementation(s) of the method and the callers of the method. A method contract is always part of a type contract (class, value type, or interface), and describes how a particular named operation is implemented. The method contract specifies the contracts that each parameter to the method must support and the contracts that the return value must support, if there is a return value.
- **Property contract** – A property contract is specified with a property definition. There is an extensible set of operations for handling a named value, which includes a standard pair for reading the value and changing the value. A property contract specifies method contracts for the subset of these operations that must be implemented by any type that supports the property contract. A type can support many property contracts, but any given property contract can be supported by exactly one type. Hence, property definitions are a part of the type definition of the type that supports the property.
- **Event contract** – An event contract is specified with an event definition. There is an extensible set of operations for managing a named event, which includes three standard methods (register interest in an event, revoke interest in an event, fire the event). An event contract specifies method contracts for all of the operations that must be implemented by any type that supports the event contract. A type can support many event contracts, but any given event contract can be supported by exactly one type. Hence, event definitions are a part of the type definition of the type that supports the event.

7.6.1 Signatures

Signatures are the part of a contract that can be checked and automatically enforced. Signatures are formed by adding constraints to types and other signatures. A constraint is a limitation on the use of or allowed operations on a value or location. Example constraints would be whether a location can be overwritten with a different value or whether a value can ever be changed.

All locations have signatures, as do all values. Assignment compatibility requires that the signature of the value, including constraints, is compatible with the signature of the location, including constraints. There are four fundamental kinds of signatures: type signatures, location signatures, parameter signatures, and method signatures.

***CLS Rule 11:** All types appearing in a signature must be CLS-compliant.*

***CLS Rule 12:** The visibility and accessibility of types and members must be such that types in the signature of any member must be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly must not have an argument whose type is visible only within the assembly.*

***CLS (consumer):** need not accept types whose members violate these rules.*

***CLS (extender):** need not provide syntax to violate these rules.*

***CLS (framework):** must not violate this rule in its exposed types and their members.*

The following sections describe the various kinds of signatures. These descriptions are cumulative: the simplest signature is a type signature; a location signature is a type signature plus (optionally) some additional attributes; and so forth.

7.6.1.1 Type Signatures

Type signatures define the constraints on a value and its usage. A type, by itself, is a valid type signature. The type signature of a value cannot be determined by examining the value or even by knowing the class type of the value. The type signature of a value is derived from the location

signature (see below) of the location from which the value is loaded. Normally the type signature of a value is the type in the location signature from which the value is loaded.

***Rationale:** The distinction between a Type Signature and a Location Signature (below) is not currently useful. It is made because certain constraints, such as “constant,” are constraints on values not locations. Future versions of this standard, or non-standard extensions, may introduce type constraints, thus making the distinction meaningful.*

7.6.1.2 Location Signatures

All locations are typed. This means that all locations have a **location signature**, which defines constraints on the location, its usage, and on the usage of the values stored in the location. Any valid type signature is a valid location signature. Hence, a location signature contains a type and may additionally contain the constant constraint. The location signature may also contain **location constraints** that give further restrictions on the uses of the location. The location constraints are:

- The **init-only constraint** promises (hence, requires) that once the location has been initialized, its contents never change. Namely, the contents are initialized before any access, and after initialization, no value can be stored in the location. The contents are always identical to the initialized value (see Classes, Interfaces and Objects). This constraint, while logically applicable to any location, shall only be placed on fields (static or instance) of compound types.
- The **literal constraint** promises that the value of the location is actually a fixed value of a built-in type. The value is specified as part of the constraint. Compilers are required to replace all references to the location with its value, and the VES therefore need not allocate space for the location. This constraint, while logically applicable to any location, shall only be placed on static fields of compound types. Fields that are so marked are not permitted to be referenced from CIL (they shall be in-lined to their constant value at compile time), but are available using Reflection and tools that directly deal with the metadata.
- The **volatile constraint** on a location requires that the value stored in the location not be cached between accesses. While logically applicable to any location, there is no encoding for this constraint. Instead, CIL includes a **volatile** prefix to certain instructions which specifies that the location to which the instruction refers should be treated as volatile.

Init-only describes a location into which a new value cannot be stored. Volatile describes a location that is changing outside the control of this portion of the program. Constant declares that the values loaded from the location are not to be used to make modifications to the value. Any or all of these constraints can be specified as part of a location signature.

Consider a location whose type is “Array of 32-bit signed integers.” If the location signature contains the init-only constraint, then the location always contains a reference to the same array in memory, but this reference can be used to store different integers within that array. If the location signature contains the constant constraint (actually a type signature constraint, but stored in the location signature) then over time, the location may contain references to different arrays. However, at any given time the current array reference should not be used to change any of the integers stored within the array. If the location signature contains the volatile constraint, then any array reference read from this location should not be cached, i.e. every reference in the source code to the location should re-load the array reference from the location.

7.6.1.3 Local Signatures

A **local signature** specifies the contract on a local variable allocated during the running of a method. A local signature contains a full location signature, plus it may specify one additional constraint:

The **byref** constraint states that the content of the corresponding location is a **managed pointer**. A managed pointer may point either to a local variable, parameter, field of a compound type, or element of an array. Managed pointers must be reported to the garbage collector, since the location to which they refer may be moved.

Note: While strictly speaking the **byref** constraint means that a managed pointer is to be passed, when the call crosses a remoting boundary (see **Error! Reference source not found.**) this may be implemented by a copy-in/copy-out mechanism. Thus programs must not rely on the aliasing behavior of true pointers.

In addition, there is one special local signature. The **typed reference** local signature states that the local will contain both a managed pointer to a location and the a runtime representation of the type that may be stored at that location. A typed reference signature is similar to a byref constraint, but while the byref specifies the type as part of the byref constraint (and hence as part of the type description), a typed reference provides the type information dynamically. Hence a typed reference is a full signature in itself and cannot be combined with other constraints. In particular, it is not possible to specify a **byref** whose type is **typed reference**.

The typed reference signature is actually represented as a built-in value type, like the integer and floating point types. In the Base Class Library, the type is known as **System.TypedReference** and in the assembly language used in Part 2 it is designated by the keyword **typedref**. This type shall only be used for parameters and local variables. It shall not be boxed, nor shall it be use as the type of a field, element of an array, return value, etc.

CLS Rule 13: Typed references are not CLS-compliant.

CLS (consumer): there is no need to accept this type.

CLS (extender): there is no need to provide syntax to define this type or to extend interfaces or classes that use this type.

CLS (framework): this type may not appear in exposed members.

7.6.1.4 Parameter Signatures

Parameter signatures define constraints on how an individual value is passed as part of a method invocation. Parameter signatures are declared by method definitions. Any valid local signature is a valid parameter signature.

7.6.1.5 Method Signatures

Method signatures are composed of

- a calling convention,
- a list of zero or more parameter signatures, one for each parameter of the method,
- and a type signature for the result value if one is produced.

Method signatures are declared by method definitions. Only one constraint can be added to a method signature in addition to those of parameter signatures:

- The **varargs** constraint can be included to indicate that all parameters past this point are optional. When it appears, the calling convention must be one that supports variable argument lists.

Method signatures are used in two different ways. They are used as part of a method definition and as a description of a calling site when calling through a function pointer. In this latter case, the method signature indicates

- the calling convention (which may include platform-specific calling conventions)
- the type of all the argument values that are being passed,
- if needed, a varargs marker indicating where the fixed parameter list ends and the variable parameter list begins

When used as part of a method definition, the varargs constraint is represented by the choice of calling convention.

CLS Rule 14: *The varargs constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.*

CLS (consumer): *there is no need to accept methods with variable argument lists or unmanaged calling convention.*

CLS (extender): *there is no need to provide syntax to declare varargs methods or unmanaged calling conventions.*

CLS (framework): *neither varargs methods nor methods with unmanaged calling conventions may be exposed externally.*

7.7 Assignment Compatibility

The constraints in the type signature and the location signature affect assignment compatibility of a value to a location. Assignment compatibility of a value (described by a type signature) to a location (described by a location signature) is defined as follows:

One of the types supported by the exact type of the value is the same as the type in the location signature.

This allows, for example, an instance of a class that inherits from a base class (hence supports the base class's type contract) to be stored into a location whose type is that of the base class.

7.8 Type Safety and Verification

Since types specify contracts, it is important to know whether a given implementation lives up to these contracts. An implementation that lives up to the enforceable part of the contract (the named signatures) is said to be **typesafe**. An important part of the contract deals with restrictions on the visibility and accessibility of named items as well as the mapping of names to implementations and locations in memory.

Typesafe implementations only store values described by a type signature in a location that is assignment compatible with the location signature of the location (see [Signatures](#)). Typesafe implementations never apply an operation to a value that is not defined by the exact type of the value. Typesafe implementations only access locations that are both visible and accessible to them. In a typesafe implementation, the exact type of a value cannot change.

Verification is a mechanical process of examining an implementation and asserting that it is typesafe. Verification is said to succeed if the process can prove that an implementation is typesafe. Verification is said to fail if that process cannot prove the type safety of an implementation. Verification is necessarily conservative: it may report failure for a typesafe implementation, but it never reports success for an implementation that is not typesafe. For example, most verification processes report implementations that do pointer-based arithmetic as failing verification, even if the implementation is in fact typesafe.

There are many different processes that can be the basis of verification. The simplest possible process simply says that all implementations are not typesafe. While correct and efficient this is clearly not particularly useful. By spending more resources (time and space) a process can correctly identify more typesafe implementations. It can be proven, however, that no mechanical process can in finite time and with no errors correctly identify all implementations as either typesafe or not typesafe. The choice of a particular verification process is thus a matter of engineering, based on the resources available to make the decision and the importance of detecting the typesafety of different programming constructs.

7.9 Type Definers

Type definers construct a new type from existing types. **Implicit types** (e.g., built-in types, arrays, and pointers including function pointers) are defined when they are used. The mention of an implicit type in a signature is in and of itself a complete definition of the type. Implicit types allow the VES to manufacture instances with a standard set of members, interfaces, etc. Implicit types need not have user-supplied names.

All other types must be explicitly defined using an explicit type definition. The explicit type definers are:

- interface definitions – used to define interface types
- class definitions – used to define:

- object types
- value types and their associated boxed types

Note that while class definitions always define class types, not all class types require a class definition. Array types and pointer types, which are implicitly defined, are also class types. See [Classes, Interfaces and Objects](#).

Similarly, not all types defined by a class definition are object types. Array types, explicitly defined object types, and boxed types are object types. Pointer types, function pointer types, and value types are not object types. See [Classes, Interfaces and Objects](#).

7.9.1 Array Types

An **array type** is defined by specifying the element type of the array, the **rank** (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array. All of these are included in any signature of an array type, although they may be marked as dynamically (rather than statically) supplied. Hence, no separate definition of the array type is needed.

Values of an array type are objects; hence an array type is a kind of object type (see [Classes, Interfaces and Objects](#)). Array objects are defined by the CTS to be a repetition of locations where values of the array element type are stored. The number of repeated values is determined by the rank and bounds of the array.

Only type signatures, not location signatures, are allowed as array element types.

Exact array types are created automatically by the VES when they are required. Hence, the operations on an array type are defined by the CTS. These generally are: allocating the array based on size and lower bound information, indexing the array to read and write a value, computing the address of an element of the array (a managed pointer), and querying for the rank, bounds, and the total number of values stored in the array.

CLS Rule 15: *Arrays elements must have a CLS-compliant type, a fixed number of dimensions, and all dimensions of the array must have zero lower bounds. While the abstract type **System.Array** is considered CLS-compliant it does have types that inherit from it but are not CLS-compliant.*

CLS (consumer): *there is no need to support arrays of non-CLS types, even when dealing with instances of **System.Array**.*

CLS (extender): *there is no need to provide syntax to define non-CLS types of arrays or to extend interfaces or classes that use non-CLS array types. Must provide access to the type **System.Array**, but may assume that all instances will have a CLS-compliant type.*

CLS (framework): *non-CLS array types may not appear in exposed members.*

Array types form a hierarchy, with all array types inheriting from the type **System.Array**. This is an abstract class that represents all arrays regardless of the type of their elements or their rank. Arrays of one dimension with zero lower bound for their elements (sometimes called **vectors**) have a type based on the type of the elements in the array, regardless of the upper bound. Arrays with more than one dimension or one dimension but with non-zero lower bound have the same type if they have the same element type and rank, regardless of lower bound on the array. Zero-dimensional arrays are not supported.

Consider the following examples, using the syntax of CIL as described in Part 2:

Table 2: Array Examples

Static specification of type	Actual type constructed	Allowed in CLS?
<code>int32[]</code>	vector of <code>int32</code>	Yes
<code>int32[0..5]</code>	vector of <code>int32</code>	Yes
<code>int32[1..5]</code>	array, rank 1, of <code>int32</code>	No
<code>int32[,]</code>	array, rank 2, of <code>int32</code>	Yes
<code>int32[0..3, 0..5]</code>	array, rank 2, of <code>int32</code>	Yes
<code>int32[1., 0..]</code>	array, rank 2, of <code>int32</code>	No

7.9.2 Pointer Types

A **pointer type** is defined by specifying a location signature for the location the pointer references. Any signature of a pointer type includes this location signature. Hence, no separate definition of the pointer type is needed.

While pointer types are Reference Types, values of a pointer type are not objects (see [Classes, Interfaces and Objects](#)), and hence it is not possible, given a value of a pointer type, to determine its exact type. The CTS provides two typesafe operations on pointer types: one to load the value from the location referenced by the pointer and the other to store an assignment compatible value into that location. The CTS also provides three operations on pointer types (byte-based address arithmetic): adding and subtracting integers from pointers, and subtracting one pointer from another. The results of the first two operations are pointers to the same type signature as the original pointer. See Part 3 (the CIL Instruction Set Specification) for details.

CLS Rule 16: *Unmanaged pointer types are not CLS-compliant.*

CLS (consumer): *there is no need to support unmanaged pointer types.*

CLS (extender): *there is no need to provide syntax to define or access unmanaged pointer types.*

CLS (framework): *unmanaged pointer types must not be externally exposed.*

7.9.3 Interface Type Definition

An **interface definition** defines an interface type. An interface type is a named group of methods, locations and other contracts that must be implemented by any object type that supports the interface contract of the same name. An interface definition is always an incomplete description of a value, and as such can never define a class type or an exact type, nor can it be an object type.

Zero or more object types can support an interface type, and only object types can support an interface type. An interface type can require that objects that support it must also support other (specified) interface types. An object type that supports the named interface contract must provide a complete implementation of the methods, locations, and other contracts specified (but not implemented by) the interface type. Hence, a value of an object type is also a value of all of the interface types the object type supports. Support for an interface contract is declared, never inferred, i.e. the existence of implementations of the methods, locations, and other contracts required by the interface type does not imply support of the interface contract.

CLS Rule 17: *CLS-compliant tools must deal with the fact that a single type may implement two interfaces and those interfaces may each require the definition of a method of the same name and signature. These methods are considered distinct and need not have the same implementation.*

CLS (consumer): *must provide some means for accessing all methods of all interfaces supported by a type*

CLS (extender): must provide a mechanism for providing independent implementations for all methods of all interfaces supported by a type. That is, it is **not** sufficient for an extender to require a single code body to implement all interface methods of the same name and signature.

CLS (framework): must assume that implementations of methods of the same name and signature on different interfaces are independent.

CLS Rule 18: CLS-compliant interfaces must not require the definition of non-CLS compliant methods in order to implement them.

CLS (consumer): there is no need to deal with such interfaces.

CLS (extender): need not provide a mechanism for defining such interfaces..

CLS (framework): must not expose any non-CLS compliant methods on interfaces it defines for external use.

Interface types are necessarily incomplete since they say nothing about the representation of the values of the interface type. For this reason, an interface type definition cannot provide field definitions for values of the interface type (i.e. instance fields), although it can declare static fields (see [Static Fields and Static Methods](#)).

Similarly, an interface type definition cannot provide implementations for any methods on the values of the types. However, an interface type definition can and usually does define method contracts (method name and method signature) that must be implemented by supporting types. An interface type definition can define and implement static methods (see [Static Fields and Static Methods](#)) since static methods are associated with the interface type itself rather than with any value of the type.

Interfaces may have static or virtual methods, but shall not have instance methods.

CLS Rule 19: CLS-compliant interfaces do not define static or instance methods, nor do they define fields. They may define properties, events, and virtual methods.

CLS (consumer): need not accept interfaces that violate these rules.

CLS (extender): need not provide syntax to author interfaces that violate these rules.

CLS (framework): must not externally expose interfaces that violate these rules. Where static methods, instance methods, or fields are required a separate class may be defined that provides them.

Interface types can also define event and property contracts that must be implemented by object types that support the interface. Since event and property contracts reduce to sets of method contracts ([Contracts](#)), the above rules for method definitions apply. For more information, see [Event Definitions](#) and [Property Definitions](#).

Interface type definitions can specify other interface contracts that implementations of the interface type are required to support. See [Interface Type Inheritance](#) for specifics.

An interface type is given a visibility attribute, as described in section [Visibility, Accessibility, and Security](#), that controls from where the interface type can be referenced. An interface type definition is separate from any object type definition that supports the interface type. Hence, it is possible, and often desirable, to have a different visibility for the interface type and the implementing object type. However, since accessibility attributes are relative to the implementing type rather than the interface itself, all members of an interface must have public accessibility, and no security permissions may be attached to members or to the interface itself.

7.9.4 Class Type Definition

All types other than interfaces, and those types for which a definition is automatically supplied by the CTS, are defined by **class definitions**. A **class type** is a complete specification of the representation of the values of the class type and all of the contracts (class, interface, method, property, and event) that are supported by the class type. Hence, a class type is an exact type. A class definition, unless it

specifies that the class is an **abstract object type**, not only defines the class type: it also provides implementations for all of the contracts supported by the class type.

A class definition, and hence the implementation of the class type, always resides in some assembly. An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality.

Note that while class definitions always define class types, not all class types require a class definition. Array types and pointer types, which are implicitly defined, are also class types. See [Classes, Interfaces and Objects](#).

An explicit class definition is used to define:

- An object type (see [Classes, Interfaces and Objects](#)).
- A value type and its associated boxed type (see [Boxing and Unboxing of Values](#)).

An explicit class definition:

- Names the class type.
- Implicitly assigns the class type name to a scope, i.e. the assembly that contains the class definition, (see [Assemblies and Scoping](#)).
- Defines the class contract of the same name (see [Contracts](#)).
- Defines the representations and valid operations of all values of the class type using member definitions for the fields, methods, properties, and events ([Member Definitions](#)).
- Defines the static members of the class type (see [Member Definitions](#)).
- Specifies any other interface and class contracts also supported by the class type.
- Supplies implementations for member and interface contracts supported by the class type.
- Explicitly declares a visibility for the type, either public or assembly (see [Visibility](#)).
- May optionally specify a method to be called to initialize the type before any access to it is permitted. In general, this method is guaranteed to be called before any instances are created, any static variables are referenced (or their address computed), and any methods on it are called.

7.9.5 Object Type Definitions

All objects are instances of an **object type**. The object type of an object is set when the object is created and it is immutable. The object type describes the physical structure of the instance and the operations that are allowed on it. All instances of the same object type have the same structure and the same allowable operations. Object types are explicitly declared by a class type definition, with the exception of Array types, which are intrinsically provided by the VES.

7.9.5.1 Scope and Visibility

Since object type definitions are class type definitions, object type definitions implicitly specify the scope of the name of object type to be the assembly that contains the object type definition, see the Scoping of Names section. Similarly, object type definitions must also explicitly state the visibility attribute of the object type (either **public** or **assembly**); see [Visibility](#).

7.9.5.2 Concreteness

An object type may be marked as **abstract** by the object type definition. An object type that is not marked **abstract** is by definition **concrete**. Only object types can be declared as abstract. Only an abstract object type is allowed to define method contracts for which the type or the VES does not also provide the implementation. Such method contracts are called abstract methods (see [Member Definitions](#)). All methods on an abstract class need not be abstract.

It is an error to attempt to create an instance of an abstract object type, whether or not the type has abstract methods. An object type that derives from an abstract object type can be concrete if it provides implementations for any abstract methods in the base object type and is not itself marked as

abstract. Instances can be made of such a concrete derived class. Locations may have an abstract type, and instances of a concrete type that derives from the abstract type may be stored in them.

7.9.5.3 Type Members

Object type definitions include member definitions for all of the members of the type. Briefly, members of a type include fields into which values are stored, methods that can be invoked, properties that are available, and events that can be raised. Each member of a type can have attributes as described in [Type Members](#).

- Fields of an object type specify the representation of values of the object type by specifying the component pieces from which it is composed (see [Fields, Array Elements, and Values](#)). Static fields specify fields associated with the object type itself (see [Static Fields and Static Methods](#)). The fields of an object type are named and they are typed via location signatures. The names of the members of the type are scoped to the type (see [Assemblies and Scoping](#)). Fields are declared using a field definition ([Field Definitions](#)).
- Methods of an object type specify operations on values of the type (see [Methods](#)). Static methods specify operations on the type itself (see [Static Fields and Static Methods](#)). Methods are named and they have a method signature. The names of methods are scoped to the type (see [Assemblies and Scoping](#)). Methods are declared using a method definition (see [Method Definitions](#)).
- Properties of an object type specify named values that are accessible via methods that read and write the value. The name of the property is the grouping of the methods; the methods themselves are also named and typed via method signatures. The names of properties are scoped to the type (see [Assemblies and Scoping](#)). Properties are declared using a property definition (see [Property Definitions](#)).
- Events of an object type specify named state transitions in which subscribers can register/unregister interest via accessor methods. When the state changes, the subscribers are notified of the state transition. The name of the event is the grouping of the accessor methods; the methods themselves are also named and typed via method signatures. The names of events are scoped to the type (see [Assemblies and Scoping](#)). Events are declared using an event definition (see [Event Definitions](#)).

7.9.5.4 Supporting Interface Contracts

Object type definitions can declare that they support zero or more interface contracts. Declaring support for an interface contract places a requirement on the implementation of the object type to fully implement that interface contract. Implementing an interface contract always reduces to implementing the required set of methods, i.e. the methods required by the interface type.

The different types that the object type implements, i.e. the object type and any implemented interface types, are each a separate logical grouping of named members. If a class **Foo** implements an interface **IFoo** and **IFoo** declares a member method **int a()** and the class also declares a member method **int a()**, there are two members, one in the **IFoo** interface type and one in the **Foo** class type. An implementation of **Foo** must provide an implementation for both, potentially shared.

Similarly, if a class implements two interfaces **IFoo** and **IBar** each of which defines a method **int a()** the class must supply two method implementations, one for each interface, although they may share the actual code of the implementation.

CLS Rule 20: *CLS-compliant classes, value types, and interfaces must not require the implementation of non-CLS-compliant interfaces.*

CLS (consumer): *need not accept classes, value types or interfaces that violate this rule.*

CLS (extender): *need not provide syntax to author classes, value types, or interfaces that violate this rule.*

CLS (framework): *must not externally expose classes, value types, or interfaces that violate this rule.*

7.9.5.5 Supporting Class Contracts

Object type definitions can declare support for one other class contract. Declaring support for another class contract is synonymous with object type inheritance (see [Object Type Inheritance](#)).

7.9.5.6 Constructors

New values of an object type are created via **constructors**. Constructors are defined via a special form of method contract, which defines the method contract as a constructor for a particular object type. The constructors for an object type are part of the object type definition. While the CTS and VES ensure that only a properly defined constructor is used to make new values of an object type, the ultimate correctness of a newly constructed object is dependent on the implementation of the constructor itself.

Object types must define at least one constructor method, but that method need not be public. Creating a new value of an object type by invoking a constructor involves the following steps in order:

1. Space for the new value is allocated in managed memory.
2. VES data structures of the new value are initialized and user-visible memory is zeroed.
3. The specified constructor for the object type is invoked.

Inside the constructor, the object type can do any initialization it chooses (possibly none).

CLS Rule 21: *An object constructor must call some class constructor of its base class before any access occurs to inherited instance data. Note that this does not apply to value types, which are not even required to have constructors.*

CLS Rule 22: *For reference types, an object constructor shall not be called except as part of the creation of an object, and no object should be initialized twice.*

CLS (consumer): *Must provide syntax for choosing the constructor to be called when an object is created.*

CLS (extender): *Must provide syntax for defining constructor methods with different signatures. May issue a compiler error if the constructor does not obey these rules.*

CLS (framework): *May assume that object creation includes a call to one of the constructors, and that no object is initialized twice. **System.MemberwiseClone** and deserialization (including object remoting) may not run constructors.*

7.9.5.7 Finalizers

A class definition that creates an object type can supply an instance method to be called when an instance of the class is no longer accessible. The garbage collector calls finalizers, at most once per instance, automatically.

7.9.6 Value Type Definition

Not all types defined by a class definition are object types (see [Classes, Interfaces and Objects](#)); in particular, value types are not object types but they are defined using a class definition. A class definition for a value type defines both the (unboxed) value type and the associated boxed type (see [Boxing and Unboxing of Values](#)). The members of the class definition define the representation of both:

1. When a non-static method (i.e. an instance or virtual method) is called on the value type its **this** pointer is a managed reference to the instance, whereas when the method is called on the associated boxed type the **this** pointer is an object reference.

Instance methods on value types receive a **this** pointer that is a managed pointer to the unboxed type whereas virtual methods (including those on interfaces implemented by the value type) receive an instance of the boxed type.

1. Value types do not support interface contracts, but their associated boxed types do.

2. A value type does not inherit; rather the base type specified in the class definition defines the base type of the boxed type.
3. The base type of a boxed type cannot have any fields.
4. Unlike object types, instances of value types do not require a constructor to be called when an instance is created. Instead, the contents of a value type are guaranteed to be initially zero (null for object fields).

7.9.7 Type Inheritance

Inheritance of types is another way of saying that the derived type guarantees support for all of the type contracts of the base type. In addition, the derived type usually provides additional functionality or specialized behavior. A type inherits from a base type by implementing the type contract of the base type. An object type inherits from one other object type and zero or more interface types. An interface type inherits from zero or more other interfaces. Value types do not inherit, although the associated boxed type is an object type and hence inherits from other types

The derived class type must support all of the supported interfaces contracts, class contracts, event contracts, method contracts, and property contracts of its base type. In addition, all of the locations defined by the base type are also defined in the derived type. The inheritance rules guarantee that code that was compiled to work with a value of a base type will still work when passed a value of the derived type. Because of this, a derived type also inherits the implementations of the base type. The derived type may extend, override, and/or hide these implementations.

7.9.8 Object Type Inheritance

With the sole exception of **System.Object**, which does not inherit from any other object type, all object types must either explicitly or implicitly declare support for (inherit from) exactly one other object type. The graph of the inherits-relation must form a singly rooted tree with **System.Object** at the base, i.e. all object types must eventually inherit from the type **System.Object**.

An object type declares it cannot be used as a base type (be inherited from) by declaring that it is a **sealed** type. It is an error if an object type is marked both **abstract** and **sealed** at the same time.

CLS Rule 23: A CLS-compliant class must inherit from a CLS-compliant class. *System.Object* is CLS-compliant.

Arrays are object types and as such inherit from other object types. Since arrays object types are manufactured by the VES, the inheritance of arrays is fixed. See [Array Types](#).

7.9.9 Value Type Inheritance

Value Types do not inherit from any other type. Logically, the boxed type corresponding to a value type

- Is an object type.
- Must specify what object type is its base type, i.e. the object type from which it inherits.
- Must have a base type that has no fields defined.
- Is **sealed** so no other type can inherit from it.

For simplicity, boxed value types shall inherit directly from **System.ValueType** unless they are enumerations, in which case they shall inherit directly from **System.Enum**. Boxed value types shall be sealed.

7.9.10 Interface Type Inheritance

Interface types can inherit from multiple interface types, i.e. an interface contract can list other interface contracts that must also be supported. Any type that implements support for an interface type must also implement support for all of the inherited interface types. This is different from object type inheritance in two ways.

- Object types form a single inheritance tree; interface types do not.

- Object type inheritance specifies how implementations are inherited; interface type inheritance does not, since interfaces do not define implementation. Interface type inheritance specifies additional contracts that an implementing object type must support.

To highlight the last difference, consider an interface, **IFoo**, that has a single method. An interface, **IBar**, which inherits from it is requiring that any object type that supports **IBar** also support **IFoo**. It does not say anything about what methods **IBar** itself must have.

7.10 Member Inheritance

Only object types can inherit implementations, hence only object types can inherit members (see [Type Inheritance](#)). Interface types, while they do inherit from other interface types, only inherit the requirement to implement method contracts, never fields or method implementations.

7.10.1 Field Inheritance

A derived object type inherits all of the non-static fields of its base object type. This allows instances of the derived type to be used wherever instances of the base type are expected (the shapes, or layouts, of the instances will be the same). Static fields are not inherited. Just because a field exists does not mean that it can be read or written. The type visibility, field accessibility, and security attributes of the field definition (see [Visibility, Accessibility, and Security](#)) determine if a field is accessible to the derived object type.

7.10.2 Method Inheritance

A derived object type inherits all of the instance and virtual methods of its base object type. It does not inherit constructors or static methods. Just because a method exists does not mean that it can be invoked. It must be accessible via the typed reference that is being used by the referencing code. The type visibility, method accessibility, and security attributes of the method definition (see [Visibility, Accessibility, and Security](#)) determine if a method is accessible to the derived object type.

A derived object type can hide a non-virtual (i.e. static or instance) method of its base type by providing a new method definition with the same name or same name and signature. Either method can still be invoked, subject to method accessibility rules, since the type that contains the method always qualifies a method reference.

Virtual methods can be marked as **final**, in which case they cannot be overridden in a derived object type. This ensures that the implementation of the method is available, by a virtual call, on any object that supports the contract of the base class that supplied the final implementation. If a virtual method is not final it is possible to demand a security permission in order to override the virtual method, so that the ability to provide an implementation can be limited to classes that have particular permissions. When a derived type overrides a virtual method, it may specify a new accessibility for the virtual method, but the accessibility in the derived class must permit at least as much access as the access granted to the method it is overriding. See [Visibility, Accessibility, and Security](#).

7.10.3 Property and Event Inheritance

Properties and events are fundamentally constructs of the metadata intended for use by tools that target the CLI and are not directly supported by the VES themselves. It is, therefore, the job of the source language compiler and the Reflection library to determine rules for name hiding, inheritance, and so forth. The source compiler must generate CIL that directly accesses the methods named by the events and properties, not the events or properties themselves.

7.10.4 Hiding, Overriding, and Layout

There are two separate issues involved in inheritance. The first is what contracts an object must implement and hence what member names and signatures it must provide. The second is the layout of the object so that an instance of a derived type can be substituted for an instance of any of its base types. Only the non-static fields and the virtual methods that are part of the derived type affect the layout of an object.

The CTS provides independent control over both the names that are visible from a base type (**hiding**) and the sharing of layout slots in the derived class (**overriding**). Hiding is controlled by marking a member in the derived class as either **hide by name** or **hide by name-and-signature**. Hiding is always performed based on the kind of member, that is, derived field names may hide base field names, but not

method names, property names, or event names. If a derived member is marked hide by name, then members of the same kind in the base class with the same name are not visible in the derived class; if the member is marked hide by name-and-signature then only a member of the same kind with exactly the same name and type (for fields) or method signature (for methods) is hidden in the derived class. Implementation of the distinction between these two forms of hiding is provided entirely by source language compilers and the Reflection library; it has no direct impact on the VES itself.

For example:

```
class Base
{ field int32          A;
  field System.String A;
  method int32         A();
  method int32         A(int32);
}
class Derived inherits from Base
{ hide-by-name         field int32 A;
  hide-by-name+signature method int32 A();
}
```

The member names available in type **Derived** are:

Table 3: Member names

Kind of member	Type / Signature of member	Name of member
Field	int32	A
Method	() -> int32	A
Method	(int32) -> int32	A

While hiding applies to all members of a type, overriding deals with object layout and is applicable only to instance fields and virtual methods. The CTS provides two forms of member overriding, **new slot** and **expect existing slot**. A member of a derived type that is marked as a new slot will always get a new slot in the object's layout, guaranteeing that the base field or method is available in the object by using a qualified reference that combines the name of the base type with the name of the member and its type or signature. A member of a derived type that is marked as expect existing slot will re-use (i.e. share or override) a slot that corresponds to a member of the same kind (field or method), name, and type if one already exists from the base type; if no such slot exists, a new slot is allocated and used.

The general algorithm that is used for determining the names in a type and the layout of objects of the type is roughly as follows:

- Flatten the inherited names (using the “hide by name” or “hide by name-and-signature” rule) *ignoring* accessibility rules.
- For each new member that is marked “expect existing slot”, look to see if an exact match on kind (i.e. field or method), name, and signature exists and use that slot if it is found, otherwise allocate a new slot.
- After doing this for all new members, add these new member-kind/name/signatures to the list of members of this type
- Finally, remove any inherited names that match the new members based on the “hide by name” or “hide by name-and-signature” rules.

7.11 Member Definitions

Object type definitions, interface type definitions, and value type definitions can include member definitions. Field definitions define the representation of values of the type by specifying the substructure of the value. Method definitions define operations on values of the type and operations on the type itself (static methods). Property and event definitions can only be defined on object types. Property and events define named groups of accessor method definitions that implement the named event or property behavior.

Nested type declarations define types whose names are scoped by the enclosing type and whose instances have full access to all members of the enclosing class.

Depending on the kind of type definition, there are restrictions on the member definitions allowed.

7.11.1 Method Definitions

Method definitions are composed of a name, a method signature, and optionally an implementation of the method. The method signature defines the calling convention, type of the parameters to the method, and the return type of the method (see [Method Signatures](#)). The implementation is the code to execute when the method is invoked. A value type or object type can define only one method of a given name and signature. However, a derived object type may have methods that are of the same name and signature as its base object type. See [Method Inheritance](#) and [Hiding, Overriding, and Layout](#).

The name of the method is scoped to the type (see [Assemblies and Scoping](#) section). Methods can be given accessibility attributes (see [Accessibility of Members](#)). Methods can only be invoked with arguments that are assignment compatible with the parameters types of the method signature. The return value of the method must also be assignment compatible with the location in which it is stored.

Methods can be marked as **static**, indicating that the method is not an operation on values of the type but rather an operation associated with the type as a whole. Methods not marked as static define the valid operations on a value of a type. When a non-static method is invoked, a particular value of the type, referred to as **this** or the **this pointer**, is passed as an implicit parameter.

A method definition that does not include a method implementation must be marked as **abstract**, unless the method definition is part of an interface definition. All non-static methods of an interface definition are abstract. Abstract method definitions are only allowed in object types that are marked as abstract.

A non-static method definition in an object type may be marked as **virtual**, indicating that an alternate implementation may be provided in derived types. All method definitions in interface definitions must be virtual methods. Virtual method can be marked as **final**, indicating that derived object types are not allowed to override the method implementation.

7.11.2 Field Definitions

Fields definitions are composed of a name and a location signature. The location signature defines the type of the field and the accessing constraints, see [Location Signatures](#). A value type or object type can define only one field of a given name. However, a derived object type may have fields that are of the same name as its base object type. See the [Field Inheritance and Hiding, Overriding, and Layout sections](#).

The name of the field is scoped to the type (see [Assemblies and Scoping](#)). Fields can be given accessibility attributes, see the [Visibility, Accessibility, and Security](#) section. Fields can only store values that are assignment compatible with the type of the field (see the [Assignment Compatible Locations](#) section).

Fields can be marked as **static**, indicating that the field is not part of values of the type but rather a location associated with the type as a whole. Locations for the static fields are created when the type is loaded and initialized when the type is initialized.

Fields not marked as static define the representation of a value of a type by defining the substructure of the value (see the [Fields, Array Elements, and Values](#) section). Locations for such fields are created within every value of the type whenever a new value is constructed. They are initialized during construction of the new value. A non-static field of a given name is always located at the same place within every value of the type.

A field that is marked **serializable** is to be serialized as part of the persistent state of a value of the type. See the [Serialization Specification](#) for details of serialization and related security issues.

7.11.3 Property Definitions

A property definition defines a named value and the methods that access the value. A property definition defines the accessing contracts on that value. Hence, the property definition specifies which accessing methods exist and their respective method contracts. An implementation of a type that

declares support for a property contract must implement the accessing methods required by the property contract. The implementation of the accessing methods defines how the value is retrieved and stored.

A property definition is always part of either an interface definition or a class definition. The name and value of a property definition is scoped to the object type or the interface type that includes the property definition. While all of the attributes of a member may be applied to a property (accessibility, static, etc.) these are not enforced by the CTS. Instead, the CTS requires that the method contracts that comprise the property must match the method implementations, as with any other method contract. There are no CIL instructions associated with properties, just metadata.

By convention, properties define a **getter** method (for accessing the current value of the property) and optionally a **setter** method (for modifying the current value of the property). The CTS places no restrictions on the set of methods associated with a property, their names, or their usage.

CLS Rule 24: *The methods that implement the **getter** and **setter** methods of a property must be marked **mdSpecialName** in the metadata.*

CLS Rule 25: *The accessibility of the property and of its accessors must be identical.*

CLS Rule 26: *The property and its accessors must all be static, all be virtual, or all be instance.*

CLS Rule 27: *The type of the property must be the return type of the **getter** and the type of the last argument of the **setter**. The types of the parameters of the property must be the types of the parameters to the **getter** and the types of all but the final parameter of the **setter**. All of these types must be CLS-compliant, and must not be managed pointers (i.e. cannot be passed by reference).*

CLS Rule 28: *Properties must adhere to a specific naming pattern. See [Naming Patterns](#).*

CLS (consumer): *Must ignore the **mdSpecialName** bit in appropriate name comparisons and must adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods which define the property.*

CLS (extender): *Must ignore the **mdSpecialName** bit in appropriate name comparisons and must adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods which define the property. In particular, an extender need not be able to define properties.*

CLS (framework): *Must design understanding that not all CLS languages will access the property using special syntax.*

7.11.4 Event Definitions

The CTS supports events in precisely the same way that it supports properties (see the [Property Definitions section](#)). The conventional methods, however, are different and include means for subscribing and unsubscribing to events as well as for firing the event.

CLS Rule 29: *The methods that implement the event must be marked **mdSpecialName** in the metadata.*

CLS Rule 30: *The **add** and **remove** methods for an event must both either be present or absent.*

CLS Rule 31: *The **add** and **remove** methods for an event must each take one parameter whose type defines the type of the event and that must be derived from **System.Delegate**.*

CLS Rule 32: *Events must adhere to a specific naming pattern. See [Naming Patterns](#)*

CLS (consumer): *Must ignore the **mdSpecialName** bit in appropriate name comparisons and must adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods that define the event.*

*CLS (extender): Must ignore the **mdSpecialName** bit in appropriate name comparisons and must adhere to identifier rules. Otherwise, no direct support other than the usual access to the methods that define the event. In particular, an extender need not be able to define events.*

CLS (framework): Must design understanding that not all CLS languages will access the event using special syntax.

7.11.5 Nested Type Definitions

A nested type definition is identical to a top-level type definition, with one exception: a top-level type has a visibility attribute, while the visibility of a nested type is the same as the visibility of the enclosing type. See [Nested Types](#).

8 CLI Metadata

New types – value types and reference types – are introduced into the CTS via type declarations expressed in **metadata**. Not limited to type declarations, metadata is a structured way to represent all information that the CLI uses to locate and load classes, lay out instances in memory, resolve method invocations, translate CIL to native code, enforce security, and set up runtime context boundaries. Every CLI PE/COFF image (see Part 2 for details) carries a compact metadata binary that is emitted into the image by the CLI-enabled development tool or compiler.

Each CLI-enabled language will expose a language-appropriate syntax for declaring types and members and for decorating them with attributes that express what services they require of the infrastructure. Type imports are also handled in a language-appropriate way, and it is the development tool or compiler that consumes the metadata to expose the types that the developer sees.

In fact, the typical component or application developer will not need to be aware of the rules for emitting and consuming CLI metadata. While it may help a developer to understand the structure of metadata, the rules outlined in this section are primarily of interest to tool builders and compiler writers.

8.1 Components and Assemblies

Because each CLI image carries the metadata for declarations, implementations, and references specific to that image, the image-specific metadata is referred to as **component metadata**, and the resulting component is said to be **self-describing**. To draw an analogy, this information integrates and extends what a combination of typelibs, IDL files, DLLRegisterServer, and a myriad of custom .xxx files express in COM or CORBA, in disparate formats and as side notes to the actual executable file. In fact, the presence of metadata in the image is part of what it means to be a CLI component.

Collections of CLI components and other files are packaged together for deployment into **assemblies**, discussed in more detail in a later section. More than just a deployment-time concept, an assembly is a logical unit of functionality that serves as the primary unit of reuse in the CLI. Effectively, assemblies establish a name scope for types.

This means that types declared and implemented in individual components are exported for use by other implementations via the assembly in which the component participates. All references to a type are scoped by the identity of the assembly in whose context the type is being used. The CLI provides services to locate a referenced assembly and “ask” that assembly to resolve the type reference. It is this mechanism that provides an isolation scope for applications: the assembly alone controls what implementation bits participate in the assembly, and the infrastructure honors these rules.

Every assembly has a **manifest** that declares what components make up the assembly, what types are exported, how type references are resolved, how assembly references are resolved, configuration rules, etc. An assembly may carry an explicit manifest or the manifest may be implicit. When explicit, the manifest includes information about exactly what implementations (specific dependent component and assembly versions) were used to successfully build the manifest, and, optionally, version binding rules to be enforced at runtime. When implicit, the manifest is derived from the components that are declared to make up the assembly, where the binding rules are, effectively, “these bits only.”

Just as CLI components are self-describing via component metadata persisted into the CLI image, so are assemblies self-describing via their manifests: All of the resource declarations and resolution rules are expressed declaratively and persisted into a binary image that can be consumed by the CLI metadata engine. When a single file makes up a CLI assembly, there is no need to carry two separate CLI files, one

to contain the component metadata and one to contain the assembly metadata. Rather, the component and assembly metadata may be combined into the metadata binary of the CLI loadable image.

8.2 Accessing Metadata

Metadata is emitted into and read from a CLI image using either direct access to the file format as described in Part 2 or through the Reflection library. It is possible to create a tool that verifies a CLI image, including the metadata, during development, based on the specifications supplied in Parts 2 and 5.

When a class is loaded at runtime, the CLI loader imports the metadata into its own in-memory data structures, which can be browsed via the CLI Reflection services. The Reflection services should be considered as similar to a compiler; they automatically walk the inheritance hierarchy to obtain information about inherited methods and fields, they have rules about hiding by name or name-and-signature, rules about inheritance of methods and properties, and so forth.

8.2.1 Metadata Tokens

Although purely an implementation artifact, the phrase **metadata token** is found throughout the CLI specifications. Part 2 talks about metadata tokens embedded in various sections of a CLI PE/COFF image. Metadata tokens are embedded in CIL and native code to encode method invocations and field accesses at call sites; the token is used by various infrastructure services to retrieve information from metadata about the reference and the type on which it was scoped in order to resolve the reference.

Logically, a metadata token is a typed identifier of a metadata object (type declaration, member declaration, etc.). Given a token, its type can be determined and it is possible to retrieve the specific metadata attributes that had been persisted for that metadata object. However, it's important to understand that a metadata token is not a persistent identifier. Rather it is scoped to a specific metadata binary. Physically, a metadata token is an index into a metadata data structure, so access is fast and direct.

8.2.2 Member Signatures in Metadata

Every location — including fields, parameters, method return values, and properties — has a type, and a specification for its type is carried in metadata.

A value type describes values that are represented as a sequence of bits. A reference type describes values that are represented as the location of a sequence of bits. The CLI provides an explicit set of built-in types, each of which has a default runtime form as either a value type or a reference type. The metadata APIs may be used to declare additional types, and part of the type specification of a variable encodes the identity of the type as well as what form (value or reference) the type is to take at runtime.

Metadata tokens representing encoded types are passed to CIL instructions that accept a type (**newobj**, **newarray**, **ldtoken**). See the CIL instruction set specification in Part 3.

Encoded types are also embedded in member signatures. To optimize runtime binding of field accesses and method invocations, the type and location signatures associated with fields and methods are encoded into member signatures in metadata. A member signature embodies all of the contract information that is used to decide whether a reference to a member succeeds or fails.

8.3 Unmanaged Code

It is possible to pass data from CLI managed code to unmanaged code. This always involves a transition from managed to unmanaged code, which has some runtime cost, but data can often be transferred without copying. When data must be reformatted the VES provides a reasonable set of default behavior, but it is possible to use metadata to explicitly require other forms of **marshalling** (i.e. reformatted copying). The metadata also allows access to unmanaged methods through pre-existing mechanisms (DLL imports by name and by EAT offset).

8.4 Method Implementation Metadata

For each method for which an implementation is supplied in the current CLI image, the tool or compiler will emit information used by the CIL-to-native code compilers, the CLI loader, and other infrastructure services. This information includes:

- Whether the code is managed or unmanaged.
- Whether the implementation is in native code or CIL.

- The location of the method body in the current image, as an address relative to the start of the image file in which it is located (a **Relative Virtual Address**, or **RVA**). Or, alternatively, the RVA is encoded as 0 and other metadata is used to tell the infrastructure where the method implementation can be found, including:
 - An implementation to be located via the CLI Interoperability Services. See related specifications for details.
 - Forwarding calls through an imported global static method.

8.5 Class Layout

In the general case, the CLI loader is free to lay out the instances of a class in any way it chooses, consistent with the rules of the CTS. However, there are times when a tool or compiler needs more control over the layout. In the metadata, a class is marked with an attribute indicating whether its layout rule is:

- **autolayout**: A class marked “autolayout” indicates that the loader is free to lay out the class in any way it sees fit; any layout information that may have been specified is ignored. This is the default.
- **layoutsequential**: A class marked “layoutsequential” guides the loader to preserve field order as emitted, but otherwise the specific offsets are calculated based on the CLI type of the field; these may be shifted by explicit offset, padding, and/or alignment information.
- **explicitlayout**: A class marked “explicitlayout” causes the loader to ignore field sequence and to use the explicit layout rules provided, in the form of field offsets and/or overall class size or alignment. There are restrictions on legal layouts, specified in Part 5.

It is also possible to specify an overall size for a class. This enables a tool or compiler to emit a value type specification where only the size of the type is supplied. This is useful in declaring CLI built-in types (such as 32 bit integer). It is also useful in situations where the data type of a member of a structured value type does not have a representation in CLI metadata (e.g., C++ bit fields). In the latter case, as long as the tool or compiler owns the layout, and CLI doesn’t need to know the details or play a role in the layout, this is sufficient. Note that this means that the VES can move bits around but can’t marshal across machines – the emitting tool or compiler will need to handle the marshaling.

Optionally, a developer may specify a packing size for a class. This is layout information that is not often used but it allows a developer to control the alignment of the fields. It is not an alignment specification, per se, but rather serves as a modifier that places a ceiling on all alignments. Typical values are 1, 2, 4, 8, or 16.

8.6 Assemblies: Name Scopes for Types

An assembly is a collection of resources that are built to work together to deliver a cohesive set of functionality. An assembly carries all of the rules necessary to ensure that cohesion. It is the unit of access to resources in the CLI.

From the outside looking in, an assembly is a collection of exported resources, including types. Resources are exported by name. From the inside, an assembly is a collection of public (exported) and private (internal to the assembly) resources. It is the assembly that determines what resources are to be exposed outside of the assembly and what resources are accessible only within the current assembly scope. It is the assembly that controls how a reference to a resource, public or private, is mapped onto the bits that implement the resource. For types in particular, the assembly may also supply runtime configuration information. A CLI image or file can be thought of as a packaging of type declarations and implementations, where the packaging decisions could change under the covers without affecting clients of the assembly.

The identity of a type is its assembly scope + its declared name. The same declared type deployed in two different assemblies is considered two different types.

Assembly Dependencies: An assembly may depend on other assemblies. This happens when implementations in the scope of one assembly reference resources that are scoped in / owned by another assembly.

- All references to other assemblies are resolved under the control of the current assembly scope. This gives an assembly an opportunity to control how a reference to another assembly is mapped onto a particular version (or other characteristic) of that referenced assembly (although that target assembly has sole control over how the referenced resource is resolved to an implementation).
- It is always possible to determine which assembly scope a particular implementation is running in. All requests originating from that assembly scope are resolved relative to that scope.

From a deployment perspective, an assembly may be deployed by itself, with the assumption that any other referenced assemblies will be available in the deployed environment. Or, it may be deployed with its dependent assemblies.

Manifests: Every assembly has a manifest that declares what modules make up the assembly, what types are exported, how type references are resolved, how assembly references are resolved, configuration rules, etc. An assembly may carry an explicit manifest or the manifest may be implicit. When explicit, the manifest includes information about exactly what implementations (specific dependent module and assembly versions) were used to successfully build the manifest, regardless of whether the assembly chooses to require binding to these specific versions at runtime. When implicit, the manifest is derived from the modules that are declared to make up the assembly, effectively binding to specific implementation versions.

Applications: Assemblies introduce isolation semantics for applications. An application is simply an assembly that has an external entry point that triggers (or causes a hosting environment such as IE to trigger) the creation of a new Application Domain. This entry point is effectively the root of a tree of request invocations and resolutions. Some applications are a single, self-contained assembly. Others require the availability of other assemblies to provide needed resources. In either case, when a request is resolved to a module to load, the module is loaded into the same Application Domain from which the request originated. It is possible to monitor or stop an application via the Application Domain.

References: A reference to a type always qualifies a type name with the assembly scope within which the reference is to be resolved – that is, an assembly establishes the name scope of available resources. However, rather than establishing relationships between individual modules and referenced assemblies, every reference is resolved through the current assembly. This allows each assembly to have absolute control over how references are resolved. This is depicted in [Figure 1: Assemblies](#).

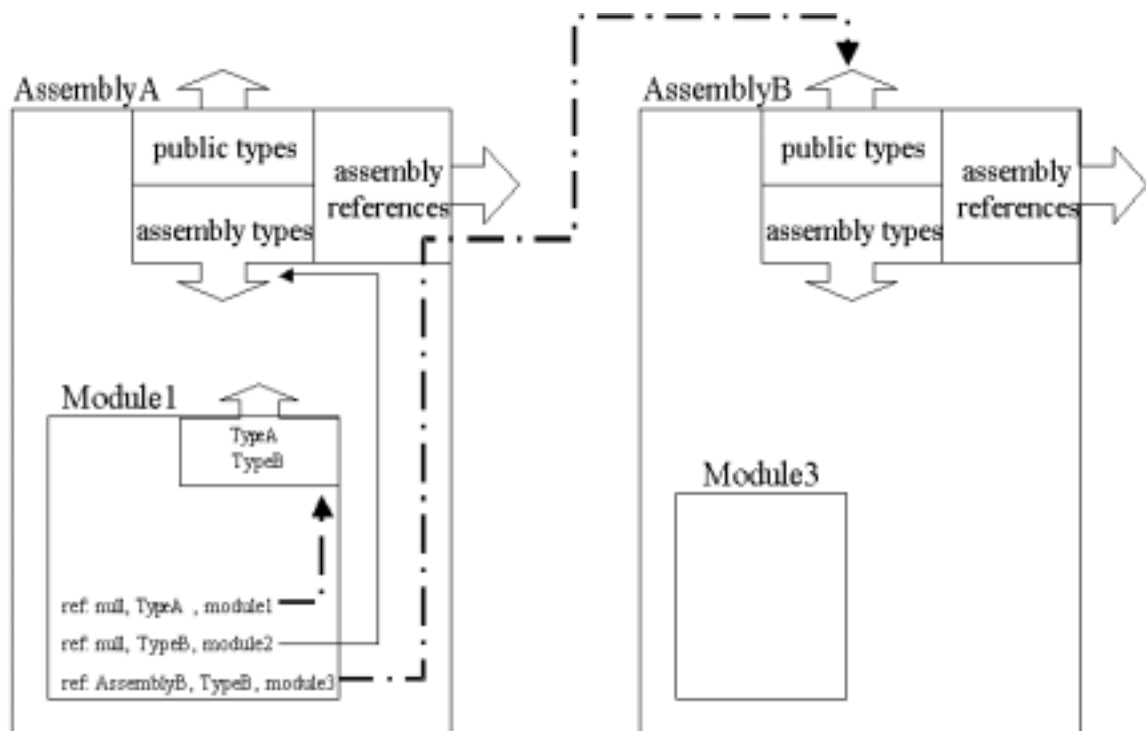


Figure 1: Assemblies

8.7 Metadata Extensibility

CLI metadata is extensible. There are three reasons this is important:

- The Common Language Specification (CLS) is a specification for conventions that languages and tools agree to support in a uniform way for better language integration. The CLS constrains parts of the CTS model, and the CLS introduces higher-level abstractions that are layered over the CTS. It is important that the metadata be able to capture these sorts of development-time abstractions that are used by tools even though they are not recognized or supported explicitly by the CLI.
- It should be possible to represent language-specific abstractions in metadata that are neither CLI nor CLS language abstractions. For example, it should be possible, over time, to enable languages like C++ to not require separate header files or IDL files in order to use types, methods, and data members exported by compiled modules.
- It should be possible, in member signatures, to encode types and type modifiers that are used in language-specific overloading. For example, to allow C++ to distinguish **int** from **long** even on 32-bit machines where both map to the underlying type **int32**.

This extensibility comes in the following forms:

- Every metadata object can carry custom attributes, and the metadata APIs provide a way to declare, enumerate, and retrieve custom attributes. Custom attributes may be identified by a simple name, where the value encoding is opaque and known only to the specific tool, language, or service that defined it. Or, custom attributes may be identified by a type reference, where the structure of the attribute is self-describing (via data members declared on the type) and any tool including the CLI Reflection services may browse the value encoding.

CLS Rule 33: *CLS-compliant tools are only required to deal with a subset of the encodings of custom attributes. The only types that can appear in these encodings are: **System.Type**, **System.String**, **System.Char**, **System.Boolean**, **System.Byte**, **System.Int16**, **System.Int32**, **System.Int64**, **System.Single**, **System.Double**, any enumeration type based on a CLS-compliant base integer type, as well as single-dimensional, zero-based arrays made of any type that can be encoded in this way (“closed under vector-of operation”).*

CLS (consumer): *Must be able to read attributes encoded using the restricted scheme. Must be able to attach attributes based on existing attribute classes to any metadata that is emitted. Must implement the rules for the **System.AttributeUsageAttribute**.*

CLS (extender): *Must be able to author new classes and new attributes, as well as all requirements for CLS consumer.*

CLS (framework): *Must externally expose only attributes that are encoded within the CLS rules and following the conventions specified for **System.AttributeUsageAttribute***

- In addition to CTS type extensibility, it is possible to emit custom modifiers into member signatures. The CLI will honor these modifiers for purposes of method overloading and hiding, as well as for binding, but will not enforce any of the language-specific semantics. These modifiers can reference the return type or any parameter of a method, or the type of a field. They come in two kinds: **required modifiers** that anyone using the member must understand in order to correctly use it, and **optional modifiers** that can be ignored if the modifier is not understood.

CLS Rule 34: *CLS-compliant tools shall not generate publicly visible required modifiers, and shall ignore optional modifiers they do not understand. They need not emit either kind of modifier, although they must be able to copy both kinds of modifiers should they exist in metadata that they import.*

CLS (consumer): Must be able to read metadata containing optional modifiers and correctly copy signatures that include them. May ignore these modifiers in type matching and overload resolution. May ignore types that become ambiguous when the optional modifiers are ignored, or that use required modifiers.

CLS (extender): Must be able to author overrides for inherited methods with signatures that include optional modifiers. There is no requirement to deal with required modifiers, nor to author new methods that have any kind of modifier in their signature.

CLS (framework): Must not use required modifiers in externally visible signatures unless they are marked as not CLS-compliant. Must not expose two members on a class that differ only by the use of optional modifiers in their signature unless only one is marked CLS-compliant.

8.8 Globals, Imports, and Exports

The CTS does not have the notion of **global statics**: all statics are associated with a particular class. Nonetheless, the metadata is designed to support languages that rely on static data that is stored directly in a PE/COFF file and accessed by its relative virtual address. In addition, while access to managed data and managed functions is mediated entirely through the metadata itself, the metadata provides a mechanism for accessing unmanaged data and unmanaged code through the PE/COFF file's Import Address Table (IAT) and for exposing unmanaged entry points to managed code through the Export Address Table (EAT).

CLS Rule 35: Global static fields and methods are not CLS-compliant.

CLS (consumer): Need not support global static fields or methods.

CLS (extender): Need not author global static fields or methods.

CLS (framework): Must not define global static fields or methods.

8.9 Scoped Statics

The CTS does not include a model for file- or function-scoped static functions or data members. However, there are times when a compiler needs a metadata token to emit into CIL for a scoped function or data member. The metadata allows members to be marked so that they are never visible/accessible outside of the PE/COFF file in which they are declared and for which the compiler guarantees to enforce all access rules.

9 Common Language Specification

The primary design challenge in creating a common language specification is choosing the right size subset – large enough that it is properly expressive and small enough that all languages can reasonably accommodate it.

Because the CLS is about language interoperability, its rules apply only to “externally visible” items. The CLS assumes that language interoperability is important only across the assembly boundary – that is, within a single assembly there are no restrictions as to the programming techniques that are used. Thus, the CLS rules apply only to items that are visible (see [Visibility of Types](#)) outside of their defining assembly and have **public**, **family**, or **family-or-assembly** accessibility (see [Accessibility of Members](#)).

It is useful to define some terminology:

- A type is **CLS-compliant** if all its publicly accessible parts (those classes, interfaces, methods, fields, properties, and events that are available to code executing in another assembly) either
- have signatures composed only of CLS-compliant types, or
- are specifically marked as not CLS-compliant (see [Marking Items as CLS-Compliant](#))
- A tool is a CLS-compliant **consumer tool** if it can completely *use* any CLS-compliant type. That is, call any CLS-compliant method; create an instance of any type CLS-compliant, read and modify any CLS-compliant field, etc.

- A tool is a CLS-compliant **extender tool** if it is a consumer tool and, in addition, can extend *any* (non-sealed) CLS-compliant base class, can implement *any* CLS-compliant interface, and can place *any* CLS-compliant custom attribute on all appropriate elements of metadata. It must also be able to define new CLS-compliant interfaces.

While many things contribute to the decision to exclude a CTS concept from the CLS, one should be explicitly called out: any construct that would make it impossible to rapidly verify code is excluded from the CLS. This allows all CLS-compliant languages to produce verifiable code if they so choose.

9.1 Marking Items as CLS-Compliant

The CLS specifies how to mark externally visible parts of an assembly to indicate whether or not they comply with the CLS requirements. This is done using the custom attribute mechanism to extend metadata. The class **System.CLSCompliantAttribute** indicates what assemblies are CLS-compliant and what types and type members are not CLS-compliant in a compliant assembly.

The constructor for **System.CLSCompliantAttribute** takes a Boolean argument indicating whether the item with which it is associated is or is not CLS-compliant. This allows any item (assembly, type, or type member) to be explicitly marked as CLS-compliant or not.

If no attribute is associated with an item, then:

- an assembly is assumed not to be CLS-compliant.
- a type is assumed to be CLS-compliant if and only if its enclosing type (for nested types) or assembly (for top-level types) is CLS-compliant.
- other members (methods, fields, properties and events) are CLS-compliant if and only if the type in which it is defined is CLS-compliant.

CLS Rule 36: *Types whose CLS-compliance differs from that of the assembly in which they are defined must be so marked with the **System.CLSCompliantAttribute**. Similarly, members whose CLS-compliance differs from that of their class must also be so marked.*

CLS (consumer): *May ignore any member that is marked as not CLS-compliant using the above rules.*

CLS (extender): *Should encourage correct labeling of newly authored assemblies, classes, interfaces, and methods. Compile-time enforcement of the CLS rules is strongly encouraged.*

CLS (framework): *Must correctly label all publicly exposed members as to their CLS compliance. The rules specified here can be used to minimize the number of markers required (for example, label the entire assembly if all types and members are compliant or if there are only a few exceptions that need to be marked).*

9.2 Identifiers

Languages that are either case-sensitive or case-insensitive can support the CLS. Since its rules apply only to items exposed to other languages, **private** members or types that aren't exported from an assembly may use any names they choose. For interoperability, however, there are some restrictions.

In order to make tools work well with a case-sensitive language it is important that the exact case of identifiers be maintained. At the same time, when dealing with non-English languages encoded in Unicode, there may be more than one way to represent precisely the same identifier that includes combining characters. The CLS requires that identifiers obey the restrictions of the appropriate Unicode standard and persist them in Canonical form C, which preserves case but forces combining characters into a standard representation. See CLS Rule 2, in [Collected CLS Rules](#).

At the same time, it is important that externally visible names not conflict with one another when used from a case-insensitive programming language. As a result, all identifier comparisons must be done internally to CLS-compliant tools using the Canonical form KC, which first transforms characters to their case-canonical representation. See CLS Rule 2, in [Collected CLS Rules](#).

When a CLS-compliant language deals with a non-CLS-compliant language it must be aware that the CTS and VES perform all comparisons using code-point (i.e. byte-by-byte) comparison. Thus, even though the CLS requires that persisted identifiers be in Canonical form C, references to non-CLS identifiers will have

to be persisted using whatever encoding the non-CLS language chose to use. It is a language design issue, not covered by the CTS or the CLS, precisely how this should be handled.

9.3 Overloading

The CTS, while it describes inheritance, object layout, name hiding, and overriding of virtual methods does not discuss overloading at all. While this is surprising, it arises from the fact that overloading is entirely handled by compilers that target the CTS and not the type system itself. In the metadata, all references to types and type members are fully resolved and include the precise signature that is intended. This choice was made since every programming language has its own set of rules for coercing types and the VES does not provide a means for expressing those rules.

Following the rules of the CTS, it is possible for names to be defined in the same scope as long as they differ in either kind (field, method, etc.) or signature. The CLS imposes a stronger restriction for overloading methods. Within a single scope, a given name may refer to any number of methods provided they differ in any of the following:

- Number of parameters
- Type of argument

Notice that the signature includes more information but CLS-compliant languages need not produce or consume classes that differ only by that additional information:

- Calling convention
- Custom modifiers
- Return type
- Whether a parameter is passed by value or by reference (i.e. as a managed pointer or by-ref)

There is one exception to this rule. For the special names `op_Implicit` and `op_Explicit` described in [Conversion Operators](#) methods may be provided that differ only by their return type. These are marked specially and may be ignored by compilers that don't support operator overloading.

Note that properties may not be overloaded by type (that is, by the return type of their **getter** method), but they may be overloaded with different number or types of indices (that is, by the number and types of the parameters of its **getter** method). Otherwise, the overloading rules for properties are identical to the method overloading rules.

CLS Rule 37: Only properties and methods may be overloaded.

CLS Rule 38: Properties, instance methods, and virtual methods may be overloaded based only on the number and types of their parameters, except the conversion operators named **op_Implicit** and **op_Explicit** which may also be overloaded based on their return type.

CLS (consumer): May assume that only properties and methods are overloaded, and need not deal with overloading based on return type unless providing special syntax for operator overloading. If return type overloading isn't supported, then the **op_Implicit** and **op_Explicit** may be ignored since the functionality must be provided in some other way by an CLS-compliant framework.

CLS (extender): Should not permit the authoring over overloads other than those specified here. It is not necessary to support operator overloading at all, hence it is possible to entirely avoid dealing with overloading on return type.

CLS (framework): Must not publicly expose overloading except as specified here. Frameworks authors should bear in mind that many modern programming languages, including Object-Oriented languages, do not support overloading and will expose overloaded methods or properties through mangled names. Most languages support neither operator overloading nor overloading based on return type, so **op_Implicit** and **op_Explicit** should always be augmented with some alternative way to gain the same functionality.

9.4 Operator Overloading

Operator overloading is not, strictly speaking, in the CLS. CLS-compliant producer tools are under no obligation to allow defining of operator overloading. CLS-compliant consumer tools do not have to provide some special mechanism to call these methods. This topic is addressed by the CLS so that

- languages that do provide operator overloading can describe their rules in a way that other languages can understand, and
- languages which do not provide operator overloading can still access the underlying functionality without the addition of special syntax.

Operator overloading is described by using the names specified below, and by setting a special bit in the metadata (**mdSpecialName**) so that they do not collide with the user's name space. A CLS-compliant producer tool must provide some means for setting this bit. If these names are used, they should have precisely the semantics described here.

9.4.1 Unary Operators

Unary operators take one argument, perform some operation on it, and return the result. They are represented as static methods on the class that defines the type of their one operand or their return type. Table 4: Unary Operator Names shows the names that are defined.

Table 4: Unary Operator Names

Name	ISO C++ Operator Symbol
op_Decrement	--
op_Increment	++
op_UnaryNegation	- (unary)
op_UnaryPlus	+ (unary)
op_LogicalNot	!
op_True	<i>Not defined</i>
op_False	<i>Not defined</i>
op_AddressOf	& (unary)
op_OnesComplement	~
op_PointerDereference	* (unary)

9.4.2 Binary Operators

Binary operators take two arguments, perform some operation and return a value. They are represented as static methods on the class that defines the type of one of their two operands or the return type. Table 5: Binary Operator Names shows the names that are defined.

Table 5: Binary Operator Names

Name	C++ Operator Symbol
op_Addition	+ (binary)
op_Subtraction	- (binary)
op_Multiply	* (binary)
op_Division	/
op_Modulus	%
op_ExclusiveOr	^
op_BitwiseAnd	& (binary)
op_BitwiseOr	
op_LogicalAnd	&&
op_LogicalOr	
op_Assign	=
op_LeftShift	<<
op_RightShift	>>
op_SignedRightShift	<i>Not defined</i>
op_UnsignedRightShift	<i>Not defined</i>
op_Equality	==
op_GreaterThan	>
op_LessThan	<
op_Inequality	!=
op_GreaterThanOrEqual	>=
op_LessThanOrEqual	<=
op_UnsignedRightShiftAssignment	<i>Not defined</i>
op_MemberSelection	->
op_RightShiftAssignment	>>=
op_MultiplicationAssignment	*=
op_PointerToMemberSelection	->*
op_SubtractionAssignment	-=
op_ExclusiveOrAssignment	^=
op_LeftShiftAssignment	<<=
op_ModulusAssignment	%=
op_AdditionAssignment	+=
op_BitwiseAndAssignment	&=
op_BitwiseOrAssignment	=
op_Comma	,
op_DivisionAssignment	/=

9.4.3 Conversion Operators

Conversion operators are unary operations that allow conversion from one type to another. The operator method must be defined as a static method on either the operand or return type. There are two types of conversions:

- An implicit (widening) coercion must not lose any magnitude or precision. These should be provided using a method named `op_Implicit`
- An explicit (narrowing) coercion may lose magnitude or precision. These should be provided using a method named `op_Explicit`

Conversions provide functionality that can't be generated in other ways, and many languages will not support the use of the conversion operators through special syntax. Therefore, we recommend that the same functionality be made available using the more common `ToXxx` and `FromXxx` naming pattern.

Because these operations may exist on the class of their operand type (so-called “from” conversions) and would therefore differ on their return type only, the CLS specifically allows that these two operators be overloaded based on their return type. The CLS, however, also requires that if this form of overloading is used then the language must provide an alternate means for providing the same functionality since not all CLS languages will implement operators with special syntax.

***CLS Rule 39:** If either `op_Implicit` or `op_Explicit` is overloaded on its return type, an alternate means of providing the coercion must be provided.*

***CLS (consumer):** Where appropriate to the language design, use the existence of `op_Implicit` and/or `op_Explicit` in choosing method overloads and generating automatic coercions.*

***CLS (extender):** Where appropriate to the language design allow the definition of implicit or explicit coercion operators to produce the corresponding `op_Implicit`, `op_Explicit`, `ToXxx`, and/or `FromXxx` methods. Otherwise, if possible, provide a means for explicitly defining those methods including the possibility of overloading `op_Implicit` and `op_Explicit` based on return type alone.*

***CLS (framework):** Consider providing coercion operations. When this is done, always provide functionality via `FromXxx` and `ToXxx` and consider providing `op_Implicit` and `op_Explicit` as well.*

9.5 Naming Patterns

While the CTS does not dictate the naming of properties or events, the CLS does specify a pattern to be observed.

For Events:

An individual event is created by choosing or defining a delegate type that is used to signal the event. Then, three methods are created with names based on the name of the event and with a fixed signature. For the examples below we define an event named **Click** that uses a delegate type named **EventHandler**.

EventAdd, used to add a handler for an event

```
Pattern: void add_<EventName> (<DelegateType> handler)
Example: void add_Click (EventHandler handler);
```

EventRemove, used to remove a handler for an event

```
Pattern: void remove_<EventName> (<DelegateType> handler)
Example: void remove_Click (EventHandler handler);
```

EventRaise, used to signal that an event has occurred

```
Pattern: void family raise_<EventName> (Event e)
```

For Properties:

An individual property is created by deciding on the type returned by its getter method and the types of the getter's parameters (if any). Then, two methods are created with names based on the name of the property and these types. For the examples below we define two properties: **Name** takes no parameters and returns a System.String, while **Item** takes a System.Object parameter and returns a System.Object. Item is referred to as an indexed property, meaning that it takes parameters and thus may appear to the user as though it were an array with indices

PropertyGet, used to read the value of the property

Pattern: <PropType> get_<PropName> (<Indices>)

Example: System.String get_Name ();

Example: System.Object get_Item (System.Object key);

PropertySet, used to modify the value of the property

Pattern: void set_<PropName> (<Indices>, <PropType>)

Example: void set_Name (System.String name);

Example: void set_Item (System.Object key, System.Object value);

9.6 Collected CLS Rules

The complete set of CLS rules are collected here for reference. Recall that these rules apply only to “externally visible” items – types that are visible outside of their own assembly and members of those types that have **public**, **family**, or **family-or-assembly** accessibility. Furthermore, items may be explicitly marked as CLS-compliant or not using the **System.CLSCompliantAttribute**.

0. CLS rules apply only to those parts of a type that are exposed outside of the defining assembly.
1. Boxed value types are not part of the CLS. Instead, use System.Object, System.ValueType or System.Enum, as appropriate.
2. Languages must follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-61633-5) governing the set of characters permitted to start and be included in identifiers, available on-line at <http://www.unicode.org/unicode/reports/tr15/tr15-18.html>. Identifiers must be persisted in their original case so that case sensitive comparisons work as expected. For CLS purposes, however, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they must differ in more than simply their case
3. Languages must provide a mechanism to reference identifiers that happen to be keywords in their own language. In addition, extender languages must provide a mechanism for defining and overriding virtual methods with names that are keywords in their language.
4. All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not
5. Fields and nested types must be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) must differ by more than just the return type (although there is a special exception discussed in Conversion Operators).
6. The underlying type of an enum must be a built-in CLS integer type.
7. There are two distinct kinds of enums, indicated by the presence or absence of the System.FlagsAttribute custom attribute. One represents named integer values, the other named bit flags that can be combined together to generate an unnamed value. While languages are encouraged to distinguish these, they need not do so.
8. Literal static fields of an enum must have the type of the enum itself.
9. Accessibility must not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility family or assembly. In this case the override must have accessibility family.

10. Extenders need not allow authoring nested types.
11. All types appearing in a signature must be CLS-compliant.
12. The visibility and accessibility of types and members must be such that types in the signature of any member must be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly must not have an argument whose type is visible only within the assembly.
13. Typed references are not CLS-compliant.
14. The varargs constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.
15. Arrays elements must have a CLS-compliant type, a fixed number of dimensions, and all dimensions of the array must have zero lower bounds. While the abstract type `System.Array` is considered CLS-compliant it does have types that inherit from it but are not CLS-compliant.
16. Unmanaged pointer types are not CLS-compliant.
17. CLS-compliant tools must deal with the fact that a single type may implement two interfaces and those interfaces may each require the definition of a method of the same name and signature. These methods are considered distinct and need not have the same implementation.
18. CLS-compliant interfaces must not require the definition of non-CLS compliant methods in order to implement them.
19. CLS-compliant interfaces do not define static or instance methods, nor do they define fields. They may define properties, events, and virtual methods.
20. CLS-compliant classes, value types, and interfaces must not require the implementation of non-CLS-compliant interfaces.
21. An object constructor must call some class constructor of its base class before any access occurs to inherited instance data. Note that this does not apply to value types, which are not even required to have constructors.
22. For reference types, an object constructor shall not be called except as part of the creation of an object, and no object should be initialized twice.
23. A CLS-compliant class must inherit from a CLS-compliant class. `System.Object` is CLS-compliant.
24. The methods that implement the getter and setter methods of a property must be marked `mdSpecialName` in the metadata.
25. The accessibility of the property and of its accessors must be identical.
26. The property and its accessors must all be static, all be virtual, or all be instance.
27. The type of the property must be the return type of the **getter** and the type of the last argument of the **setter**. The types of the parameters of the property must be the types of the parameters to the **getter** and the types of all but the final parameter of the **setter**. All of these types must be CLS-compliant, and must not be managed pointers (i.e. cannot be passed by reference).
28. Properties must adhere to a specific naming pattern. See [Naming Patterns](#).
29. The methods that implement the event must be marked `mdSpecialName` in the metadata.
30. The **add** and **remove** methods for an event must both either be present or absent.
31. The **add** and **remove** methods for an event must each take one parameter whose type defines the type of the event and that must be derived from **System.Delegate**
32. Events must adhere to a specific naming pattern. See [Naming Patterns](#).
33. CLS-compliant tools are only required to deal with a subset of the encodings of custom attributes. The only types that can appear in these encodings are: **System.Type**, **System.String**, **System.Char**, **System.Boolean**, **System.Byte**, **System.Int16**, **System.Int32**, **System.Int64**, **System.Single**,

System.Double, any enumeration type based on a CLS-compliant base integer type, as well as single-dimensional, zero-based arrays made of any type that can be encoded in this way (“closed under vector-of operation”).

34. CLS-compliant tools shall not generate publicly visible required modifiers, and shall ignore optional modifiers they do not understand. They need not emit either kind of modifier, although they must be able to copy both kinds of modifiers should they exist in metadata that they import.
35. Global static fields and methods are not CLS-compliant.
36. Types whose CLS-compliance differs from that of the assembly in which they are defined must be so marked with the `System.CLSCompliantAttribute`. Similarly, members whose CLS-compliance differs from that of their class must also be so marked.
37. Only properties and methods may be overloaded.
38. Properties, instance methods, and virtual methods may be overloaded based only on the number and types of their parameters, except the conversion operators named **op_Explicit** and **op_Implicit** which may also be overloaded based on their return type.
39. If either **op_Explicit** or **op_Implicit** is overloaded on its return type, an alternate means of providing the coercion must be provided.
40. Objects that are thrown must be of type **System.Exception** or inherit from **System.Exception**.

10 Supported Data Types

The CLI directly supports the data types shown in Table 6: Data Types Directly Supported by the CLI. That is, these data types can be manipulated using the CIL instruction set.

Table 6: Data Types Directly Supported by the CLI

Data Type	Description
I1	8-bit 2’s complement signed value
U1	8-bit unsigned binary value
I2	16-bit 2’s complement signed value
U2	16-bit unsigned binary value
I4	32-bit 2’s complement signed value
U4	32-bit unsigned binary value
I8	64-bit 2’s complement signed value
U8	64-bit unsigned binary value
R4	32-bit IEEE 754 floating point value
R8	64-bit IEEE 754 floating point value
I	natural size 2’s complement signed value
U	natural size unsigned binary value, also unmanaged pointer
F	natural size floating point number (internal to VES, not user visible)
O	natural size object reference to managed memory
&	natural size managed pointer (may point into managed memory)

The CLI model uses an evaluation stack. Instructions that copy values from memory to the evaluation stack we call “loads”; instructions that copy values from the stack back to memory we call “stores”. The full set of data types in Table 6: Data Types Directly Supported by the CLI can be represented in memory. However, the CLI supports only a subset of these types in its operations upon values stored on its evaluation stack – I4,

I8, I. In addition the CLI supports an internal data type, F, to represent floating point values on the internal evaluation stack. The F type can be thought of as starting at the size of values loaded from memory and then expanded to a size convenient for the underlying hardware when combined with higher-precision values. Shorter values (I1, I2, U1, U2) are widened when loaded (memory-to-stack) and narrowed when stored (stack-to-memory). This reflects a computer model that assumes, for numeric and object references, memory cells are 1, 2, 4, or 8 bytes wide but stack locations are either 4 or 8 bytes wide. User-defined value types can appear in memory locations or on the stack and have no size limitation; the only built-in operations on them are those that compute their address and copy them between the stack and memory.

The support for short numeric values consists of:

- Load and store instructions to/from memory: **ldelem, ldind, stind, stelem**
- Arithmetic with overflow detection: **add.ovf, mul.ovf, sub.ovf**
- Data conversion: **conv, conv.ovf**
- Loading constants: **ldc**
- Array creation: **newarr**

The signed integer (I1, I2, I4, I8, and I) and unsigned integer (U1, U2, U4, U8, and U) types differ only in how the bits of the integer are interpreted. For those operations where an unsigned integer is treated differently from a signed integer (e.g. comparisons or arithmetic with overflow) there are separate instructions for treating an integer as unsigned (e.g. **cgt.un** and **add.ovf.u**).

This instruction set design simplifies IL-to-native code (eg. JIT) compilers and interpreters of CIL by allowing them to internally track a smaller number of data types. See [The Evaluation Stack](#).

As described below, CIL instructions do not specify their operand types. Instead, the CLI keeps track of operand types based on data flow and aided by a stack consistency requirement described below. For example, the single **add** instruction will add two integers or two floats from the stack.

10.1 Natural Size: I, U, O and &

The natural-size, or generic, types (I, U, O, and &) are a mechanism in the CLI for deferring the choice of a value's size. These data types exist as CIL types. But the CLI maps each to the natural size for a specific processor. (For example, data type I would map to I4 on a Pentium processor, but to I8 on an IA64 processor). So, the choice of size is deferred until JIT compilation or runtime, when the CLI has been initialized and the architecture is known. This implies that field and stack frame offsets are also not known at compile time. For languages like Visual Basic, where field offsets are not computed early anyway, this is not a hardship. In languages like C or C++, where sizes must be known when source code is compiled, a conservative assumption that they occupy 8 bytes is sometimes acceptable (for example, when laying out compile-time storage).

10.1.1 Unmanaged Pointers as Type U

For languages like C, when compiling all the way to native code, where the size of a pointer is known at compile time and there are no managed objects, the fixed-size unsigned integer types (U4 or U8) can serve as pointers. However choosing pointer size at compile time has its disadvantages. If pointers were chosen to be 32 bit quantities at compile time, the code would be restricted to 4gig of address space, even if it were run on a 64 bit machine. Moreover, a 64 bit CLI would need to take special care so those pointers passed back to 32-bit code could always fit in 32 bits. If pointers were chosen at compile time to be 64 bits, the code could be run on a 32 bit machine, but pointers in **every** data structure would be twice as large as necessary on that CLI.

For other languages, where the size of a data type need not be known at compile time, it is desirable to defer the choice of pointer size from compile time to CLI initialization time. In that way, the *same CIL code* can handle large address spaces for those applications that need them, while also being able to reap the size benefit of 32 bit pointers for those applications that do not need a large address space.

For these reasons, the U type is used to represent unmanaged pointers with the VES. The metadata allows unmanaged pointers to be represented in a strongly typed manner, but these types are translated into type U for use by the VES.

10.1.2 Managed Pointer Types: **O** and **&**

The **O** datatype represents an object reference that is managed by the CLI. As such, the number of specified operations is severely limited. In particular, references can only be used on operations that indicate that they operate on reference types (e.g. **ceq** and **ldind.ref**), or on operations whose metadata indicates that references are allowed (e.g. **call**, **ldsfd**, and **stfd**).

The **&** datatype (managed pointer) is similar to the **O** type, but points to the interior of an object. That is, a managed pointer is allowed to point to a field within an object or an element within an array, rather than to point to the ‘start’ of object or array.

Object references (**O**) and managed pointers (**&**) must be reported to the CLI memory manager so that it can update their values as the items they point to are moved during garbage collection.

In summary, object references, or **O** types, refer to the ‘outside’ of an object, or to an object as-a-whole. But managed pointers, or **&** types, refer to the interior of an object. The **&** types are sometimes called “by-ref types” in source languages, since passing a field of an object by reference is represented in the VES by using an **&** type to represent the type of the parameter.

In order to allow managed pointers to be used more flexibly, they are also permitted to point to areas that aren’t under the control of the CLI garbage collector, such as the evaluation stack, static variables, and unmanaged memory. This allows them to be used in many of the same ways that unmanaged pointers (**U**) are used. As a result, however, managed pointers are allowed to appear only as parameters or local variables; this guarantees that a managed pointer to a value on the evaluation stack doesn’t outlast the life of location to which it points.

10.1.3 Portability: Storing Pointers in Memory

Several instructions, including **calli**, **cpblk**, **initblk**, **ldind.***, and **stind.***, expect an address on the top of the stack. If this address is derived from a pointer stored in memory, there is an important portability consideration.

1. Code that stores pointers in a natural sized integer or pointer location (types **I**, **O**, **U**, or **&**) is always fully portable.
2. Code that stores pointers in an 8 byte integer (type **I8** or **U8**) *can* be portable. But this requires that a **conv.ovf.u** instruction be used to convert the pointer from its memory format before its use as a pointer. This may cause a runtime exception if run on a 32-bit machine.
3. Code that uses any smaller integer type to store a pointer in memory (**I1**, **U1**, **I2**, **U2**, **I4**, **U4**) is *never* portable, even though the use of a **U4** or **I4** will work correctly on a 32-bit machine.

10.2 Handling of Short Integer Data Types

The CLI defines an evaluation stack that contains either 4-byte or 8-byte integers, but a memory model that encompasses in addition 1-byte and 2-byte integers. To be more precise, the following rules are part of the CLI model:

- Loading from 1-byte or 2-byte locations (arguments, locals, fields, statics, pointers) expands to 4-byte values. For locations with a known type (e.g. local variables) the type being accessed determines whether the load sign-extends (signed locations) or zero-extends (unsigned locations). For pointer dereference (**ldind.***), the instruction itself identifies the type of the location (e.g. **ldind.u1** indicates an unsigned location, while **ldind.i1** indicates a signed location).
- Storing into a 1-byte or 2-byte location truncates to fit and will not generate an overflow error. Specific instructions (**conv.ovf.***) can be used to test for overflow before storing.
- Calling a method in essence assigns values from the evaluation stack to the arguments for the method, hence it truncates just as any other store would.
- Returning from a method in essence assigns a value to an invisible return variable, so it also truncates as a store would. Since the value of this return variable is then placed on the evaluation stack, it is then sign-extended or zero-extended as would any other load. Notice that this truncation followed by extending is *not* identical to simply leaving the computed value unchanged.

It is the responsibility of any translator from CIL to native machine instructions to make sure that these rules are faithfully modeled through the native conventions of the target machine. The CLI does not specify, for example, whether truncation of short integer arguments occurs at the call site or in the target method.

10.3 Handling of Floating Point Datatypes

The CLI assumes floating-point calculations are handled as described in the IEEE 754 standard, “IEEE Standard for Binary Floating-point Arithmetic”. This standard describes encoding of floating point numbers, definitions of the basic operations and conversion, rounding control, and exception handling.

The standard defines special values, **NaN**, (not a number), **+infinity**, and **-infinity**. These values are returned on overflow conditions. A general principle is that operations that have a value in the limit return an appropriate infinity while those that have no limiting value return **NaN**, but see the standard for details. The following examples show the most commonly encountered cases:

```
X rem 0 = NaN
0 * +infinity = 0 * -infinity = NaN
(X / 0) = +infinity, if X>0
NaN, if X=0
• infinity, if X < 0
NaN op X = X op NaN = NaN for all operations
(+infinity) + (+infinity) = (+infinity)
X / (+infinity) = 0
X mod (-infinity) = -X
(+infinity) - (+infinity) = NaN
```

For purposes of comparison, infinite values act like a number of the correct sign but with a very large magnitude when compared with finite values. **NaN** is ‘unordered’ for comparisons (see **clt**, **clt.un**).

While the IEEE 754 spec also allows for exceptions to be thrown under unusual conditions (overflow, invalid operand, ...), the CLI does not generate these exceptions. Instead, the CLI uses the **NaN** return values and provides the instruction **ckfinite** to allow users to generate an exception if a result is **NaN**, **+infinity**, or **-infinity**.

The rounding mode defined in IEEE 754 shall be set by the CLI to “round to the nearest number,” and neither the CIL nor the class library provide a mechanism for modifying this setting. Conforming implementations of the CLI need not be resilient to external interference with this setting. That is, they need not restore the mode prior to performing floating-point operations, but rather may rely on it having been set as part of their initialization.

For conversion to integers, the default operation supplied by the CIL is “truncate towards zero”. There are class libraries supplied to allow floating-point numbers to be converted to integers using any of the other three traditional operations (**round** to nearest integer, **floor** (truncate towards **-infinity**), **ceiling** (truncate towards **+infinity**)).

Storage locations for floating point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are **R4** and **R8**. Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables) floating point numbers are represented using the internal **F** type. This type can be thought of as starting at the size of value loaded from storage and then expanding as needed, typically to the natural size for the hardware or as required for efficient implementation of an operation.

Rationale: This design allows the CLI to choose a platform-specific high-performance representation for floating point numbers until they are placed in storage locations. For example, it may be able to leave floating point variables in hardware registers that provide more precision than a user has requested. At the same time, CIL generators can force operations to respect language-specific rules for representations through the use of conversion instructions.

When a value of type **F** is put in a storage location it is automatically coerced to the required size, which may involve a loss of precision or the creation of an out-of-range marker (a **NaN**). To detect values that cannot be converted to a particular storage type, use a conversion instruction (**conv.r4**, or **conv.r8**) and then check for a non-finite value using **ckfinite**. To detect underflow when converting to a particular storage type, a comparison to zero is required before and after the conversion.

10.4 CIL Instructions and Numeric Types

Most CIL instructions that deal with numbers take their operands from the evaluation stack (see [The Evaluation Stack](#)), and these inputs have an associated type that is known to the VES. As a result, a single operation like **add** can have inputs of any numeric data type, although not all instructions can deal with all combinations of operand types. Binary operations other than addition and subtraction require that both operands be of the same type. Addition and subtraction allow an integer to be added to or subtracted from a managed pointer (types **&** and **O**). Details are specified in Part 2.

Instructions fall into the following categories:

Numeric: These instructions deal with both integers and floating point numbers, and consider integers to be signed. Simple arithmetic, conditional branch, and comparison instructions fit in this category.

Integer: These instructions deal only with integers. Bit operations and unsigned integer division/remainder fit in this category.

Floating point: These instructions deal only with floating point numbers.

Specific: These instructions deal with integer and/or floating point numbers, but have variants that deal specially with different sizes and unsigned integers. Integer operations with overflow detection, data conversion instructions, and operations that transfer data between the evaluation stack and other parts of memory (see [Method State](#)) fit into this category.

Unsigned/unordered: There are special comparison and branch instructions that treat integers as unsigned and consider unordered floating point numbers specially (as in “branch if greater than or unordered”):

Load constant: The load constant (**ldc.***) instructions can be used to load constants of type I4, I8, R4 or R8. Natural size constants (type I) must be created by conversion from I4 (conversion from I8 would not be portable) using **conv.i** or **conv.u**.

[Table 7: CIL Instructions by Numeric Category](#) shows the CIL instructions that deal with numeric values, along with the category to which they belong. Instructions that end in “.*” indicate all variants of the instruction (based on size of data and whether the data is treated as signed or unsigned).

Table 7: CIL Instructions by Numeric Category

add	Numeric	div	Numeric
add.ovf.*	Specific	div.un	Integer
and	Integer	ldc.*	Load constant
beq[.s]	Numeric	ldelem.*	Specific
bge[.s]	Numeric	ldind.*	Specific
bge.un[.s]	Unsigned/unordered	mul	Numeric
bgt[.s]	Numeric	mul.ovf.*	Specific
bgt.un[.s]	Unsigned/unordered	neg	Integer
ble[.s]	Numeric	newarr.*	Specific
ble.un[.s]	Unsigned/unordered	not	Integer
blt[.s]	Numeric	or	Integer
blt.un[.s]	Unsigned/unordered	rem	Numeric
bne.un[.s]	Unsigned/unordered	rem.un	Integer
ceq	Numeric	shl	Integer
cgt	Numeric	shr	Integer
cgt.un	Unsigned/unordered	shr.un	Specific
ckfinite	Floating point	stelem.*	Specific
clt	Numeric	stind.*	Specific
clt.un	Unsigned/unordered	sub	Numeric
conv.*	Specific	sub.ovf.*	Specific
conv.ovf.*	Specific	xor	Integer

10.5 CIL Instructions and Pointer Types

The CLI has the ability to track pointers to objects and to collect objects that are no longer reachable (memory management by “garbage collection”). This process moves objects in order to reduce the working set and thus must modify all pointers to those objects as they move. For this to work correctly, pointers to objects must only be used in certain ways. The **O** (object reference) and **&** (managed pointer) datatypes are the formalization of these restrictions.

The use of object references is tightly restricted in the CIL. They are used almost exclusively with the “virtual object system” instructions, which are specifically designed to deal with objects. In addition, a few of the base instructions of the CIL can handle object references. In particular, object references can be:

1. Loaded onto the evaluation stack to be passed as arguments to methods (**ldloc**, **ldarg**), and stored from the stack to their home locations (**stloc**, **starg**)
2. Duplicated or popped off the evaluation stack (**dup**, **pop**)
3. Tested for equality with one another, but not other data types (**beq**, **beq.s**, **bne**, **bne.s**, **ceq**)
4. Loaded-from / stored-into unmanaged memory, in type unmanaged code only (**ldind.ref**, **stind.ref**)
5. Created as a null reference (**ldnull**)

6. Returned as a value (**ret**)

Managed pointers have several additional base operations.

1. Addition and subtraction of integers, in units of *bytes*, returning a managed pointer (**add**, **add.ovf.u**, **sub**, **sub.ovf.u**)
2. Subtraction of two managed pointers to elements of the same array, returning the number of *bytes* between them (**sub**, **sub.ovf.u**)
3. Unsigned comparison and conditional branches based on two managed pointers (**bge.un**, **bge.un.s**, **bgt.un**, **bgt.un.s**, **ble.un**, **ble.un.s**, **blt.un**, **blt.un.s**, **cgt.un**, **clt.un**)

Since the memory manager runs asynchronously with respect to programs and updates managed pointers, both the distance between distinct objects and their relative position can change. Arithmetic operations upon managed pointers are intended *only* for use on pointers to elements of the same array. Other uses of arithmetic on managed pointers is unspecified.

10.6 Aggregate Data

The CLI supports *aggregate data*, that is, data items that have sub-components (arrays, structures, or object instances) but are passed by copying the value. The sub-components can include references to managed memory. Aggregate data is represented using a *value type*, which can be instantiated in two different ways:

- **Boxed**: as an Object, carrying full type information at runtime, and typically allocated on the heap by the CLI memory manager.
- **Unboxed**: as a “value type instance” which does *not* carry type information at runtime and which is never allocated directly on the heap. It can be part of a larger structure on the heap – a field of a class, a field of a boxed value type, or an element of an array. Or it can be in the local variables or incoming arguments array (see [Method State](#)). Or it can be allocated as a static variable or static member of a class or a static member of another value type.

- 2 Because value type instances, specified as method arguments, are copied on method call, they do not have “identity” in the sense that Objects (boxed instances of classes) have.

10.6.1 Homes for Values

The **home** of a data value is where it is stored for possible reuse. The CLI directly supports the following home locations:

1. An incoming **argument**
2. A **local variable** of a method
3. An instance **field** of an object or value type
4. A **static** field of a class, interface, or module
5. An **array element**

For each home location, there is a means to compute (at runtime) the address of the home location and a means to determine (at JIT compile time) the type of a home location. These are summarized in [Table 8: Address and Type of Home Locations](#).

Table 8: Address and Type of Home Locations

Type of Home	Runtime Address Computation	JITtime Type Determination
Argument	ldarga for by-value arguments or ldarg for by-reference arguments	Method signature
Local Variable	ldloca for by-value locals or ldloc for by-reference locals	Locals signature in method header
Field	ldflda	Type of field in the class, interface, or module

Static	ldslfda	Type of field in the class, interface, or module
Array Element	ldlema for single-dimensional zero-based arrays or call the instance method Address	Element type of array

In addition to homes, built-in values can exist in two additional ways (i.e. without homes):

1. as constant values (typically embedded in the CIL instruction stream using **ldc.*** instructions)
2. as an intermediate value on the evaluation stack, when returned by a method or CIL instruction.

10.6.2 Operations on Value Type Instances

Value type instances can be created, passed as arguments, returned as values, and stored into and extracted from locals, fields, and elements of arrays (i.e., copied). Like classes, value types can have both static and non-static members (methods and fields). But, because they carry no type information at runtime, value type instances are not substitutable for items of type `Object`; in this respect, they act like the primitive types `int`, `long`, and so forth. There are two operations, box and unbox, that convert between value type instances and `Objects`.

10.6.2.1 Initializing Instances of Value Types

There are three options for initializing the home of a value type instance. You can zero it by loading the address of the home (see Table 8: Address and Type of Home Locations) and using the **initobj** instruction (for local variables this can also be accomplished by setting the **zero initialize** bit in the method's header). You can call a user-defined constructor by loading the address of the home (see Table 8: Address and Type of Home Locations) and then calling the constructor directly. Or you can copy an existing instance into the home, as described in Loading and Storing Instances of Value Types.

10.6.2.2 Loading and Storing Instances of Value Types

There are two ways to load a value type onto the evaluation stack:

- Directly load the value from a home that has the appropriate type, using an **ldarg**, **ldloc**, **ldfld**, or **ldsfld** instruction
- Compute the address of the value type, then use an **ldobj** instruction

Similarly, there are two ways to store a value type from the evaluation stack:

- Directly store the value into a home of the appropriate type, using a **starg**, **stloc**, **stfld**, or **stsfld** instruction
- Compute the address of the value type, then use a **stobj** instruction

10.6.2.3 Passing and Returning Value Types

Value types are treated just as any other value would be treated:

- **To pass a value type by value**, simply load it onto the stack as you would any other argument: use **ldloc**, **ldarg**, etc., or call a method which returns a value type. To access a value type parameter that has been passed by value use the **ldarga** instruction to compute its address or the **ldarg** instruction to load the value onto the evaluation stack.
- **To pass a value type by reference**, load the address of the value type as you normally would (see Table 8: Address and Type of Home Locations). To access a value type parameter that has been passed by reference use the **ldarg** instruction to load the address of the value type and then the **ldobj** instruction to load the value type onto the evaluation stack.
- **To return a value type**, just load the value onto an otherwise empty evaluation stack and then issue a **ret** instruction.

10.6.2.4 Calling Methods

Static methods on value types are handled no differently from static methods on an ordinary class: use a **call** instruction with a metadata token specifying the value type as the class of the method. Non-static methods (i.e. instance and virtual methods) are supported on value types, but they must be given special treatment. A non-static method on a class (rather than a value type) expects a **this** pointer which is an instance of that class. This makes sense for classes, since they have identity and the **this** pointer represents that identity. Value types, however, have identity only when boxed. To address this issue, the **this** pointer on a non-static method of a value type is a by-ref parameter of the value type rather than an ordinary by-value parameter.

A non-static method on a value type may be called in the following ways:

- Given an unboxed instance of a value type, the **call** instruction can be used to invoke the function, passing as the first parameter (the **this** pointer) the address of the instance. The metadata token used with the **call** instruction must specify the value type itself as the class of the method.
- Given a boxed instance of a value type, the **call** instruction can be used to invoke the function, passing the boxed instance as the first parameter (the **this** pointer). The metadata token used must specify **System.Object** as the class of the method.

To call a non-static method of an interface that is implemented by a value type or a virtual method inherited from **System.Object** you must box the value type and use a **callvirt** instruction. For a method on an interface, the metadata token must specify the interface as the type of the method, and for an inherited method it must specify **System.Object** as the class of the method.

10.6.2.5 Boxing and Unboxing

Box and **unbox** are conceptually equivalent to (and may be seen in higher-level languages as) casting between a value type instance and **System.Object**¹. Because they change data representations, however, boxing and unboxing are like the widening and narrowing of various sizes of integers (the **conv** and **conv.ovf** instructions) rather than the casting of reference types (the **isinst** and **castclass** instructions). The **box** instruction is a widening (always typesafe) operation that converts a value type instance to **System.Object** by making a copy of the instance and embedding it in a newly allocated object. **Unbox** is a narrowing (runtime exception may be generated) operation that converts a **System.Object** (whose runtime type must be a value type) to a value type instance. This is done by computing the address of the embedded value type instance without making a copy of the instance.

10.6.2.6 Castclass and IsInst on Value Types

Casting to and from value type instances isn't possible (the equivalent operations are **box** and **unbox**). When boxed, however, it is possible to use the **isinst** instruction to see whether a value of type **System.Object** is the boxed representation of a particular class.

10.6.3 Opaque Classes

Some languages provide multi-byte data structures whose contents are manipulated directly by address arithmetic and indirection operations. To support this feature, the CLI allows value types to be created with a specified size but no information about their data members. Instances of these "opaque classes" are handled in precisely the same way as instances of any other class, but the **ldfld**, **stfld**, **ldflda**, **ldsfd**, and **stsfld** instructions cannot be used to access their contents.

11 Executable Image Information

Part 2 provides details of the CLI PE file format. The CLI relies on the following information about each method defined in a PE file:

- The *instructions* composing the method body, including all exception handlers.

¹ Actually, to the boxed type corresponding to the value type in languages that support strongly typed boxed value types.

- The *signature* of the method, which specifies the return type and the number, order, parameter passing convention, and primitive data type of each of the arguments. It also specifies the native calling convention (this does *not* affect the CIL virtual calling convention, just the native code).
- The *exception handling array*. This array holds information delineating the ranges over which exceptions are filtered and caught. See [Exception Handling](#).
- The size of evaluation stack that the method will require.
- The size of the locals array that the method will require.
- A “zero init flag” that indicates whether the local variables and memory pool should be initialized by the CLI (see also **localloc**).
- Type of each local variable in the form of a signature of the local variable array (called the “locals signature”).

In addition, the file format is capable of indicating the degree of portability of the file. There are two kinds of restrictions that can be described:

- Restriction to a specific (32-bit or 64-bit) natural size for integers.
- Restriction to a specific “endian-ness” (i.e. whether bytes are stored left-to-right or right-to-left within a machine word).

By stating what restrictions are placed on executing the code, the CLI class loader can prevent non-portable code from running on an architecture that it cannot support.

12 Machine State

One of the design goals of the CLI is to hide the details of a method call frame from the CIL code generator. This allows the CLI (and not the CIL code generator) to choose the most efficient calling convention and stack layout. To achieve this abstraction, the call frame is integrated into the CLI. The machine state definitions below reflect these design choices, where machine state consists primarily of global state and method state.

12.1 The Global State

The CLI manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system), multiple managed heaps, and a shared memory address space. A thread of control can be thought of, somewhat simplistically, as a singly linked list of *method states*, where a new state is created and linked back to the current state by a method call instruction – the traditional model of a stack-based calling sequence. Notice that this model of the thread of control doesn’t correctly explain the operation of **tail**, **jmp**, or **throw** instructions.

[Figure 2: Machine State Model](#) illustrates the machine state model which includes threads of control, method states, and multiple heaps in a shared address space. Method state, shown separately in [Figure 3: Method State](#), is an abstraction of the stack frame. Arguments and local variables are part of the method state, but they can contain Object References that refer to data stored in any of the managed heaps.

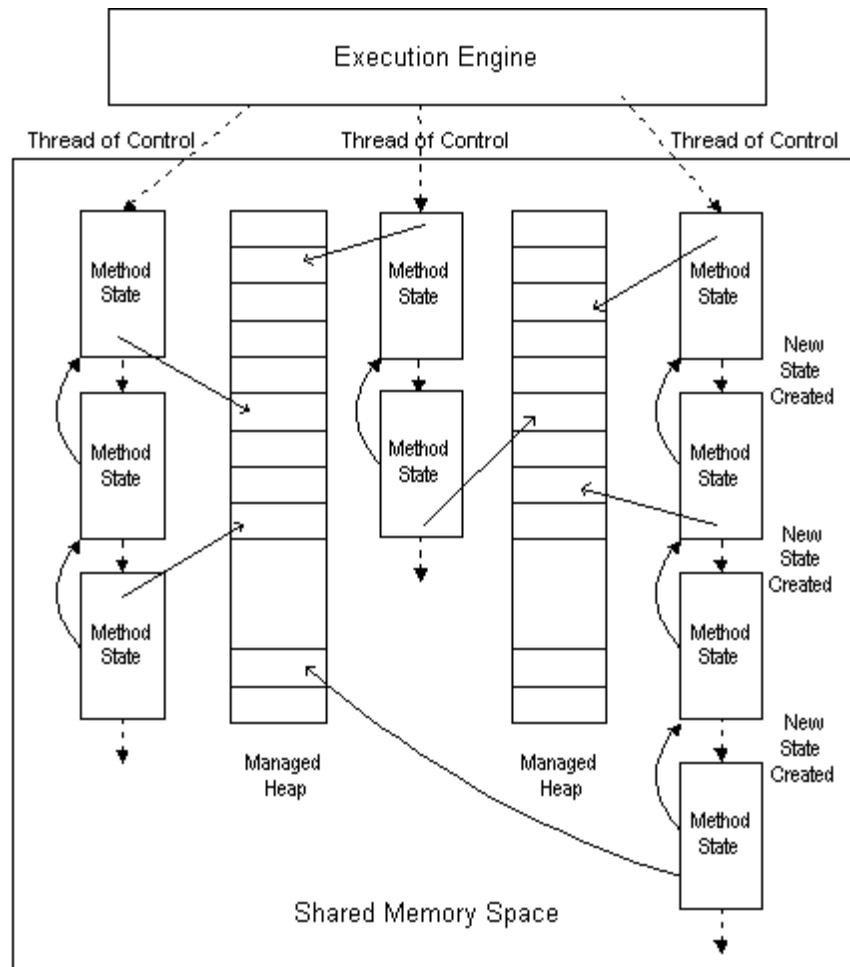


Figure 2: Machine State Model

12.2 The Memory Store

By “memory store” we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a data object. The CLI accesses data objects in the memory store via the **ldind.*** and **stind.*** instructions.

12.2.1 Alignment

Alignment of datatypes larger than 1 byte is dependent on the target CPU. It is strongly recommended that primitive datatypes be aligned to the size of that datatype. That is I2 and U2 start on even address; I4, U4, and R4 start on an address divisible by 4; and I8, U8, and R8 start on an address divisible by 8. The natural size types (I, U, and &) are always generated by the CLI aligned to their natural size (4 bytes or 8 bytes, depending on architecture). When generated externally, these should also be aligned to their natural size, but portable code may choose to enforce the stronger restriction of 8 byte alignment which is guaranteed to be architecture independent.

There is a special prefix instruction, **unaligned.**, that can immediately precede a **ldind**, **stind**, **in itblk**, or **cpblk** instruction. It indicates that the data may not be fully aligned and requires that the JIT generate code that will not cause unaligned memory faults.

12.2.2 Byte Ordering

For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering may not run on all platforms. The PE file format (see the Executable Image Information section) allows the file to be marked to indicate that it depends on a particular type ordering.

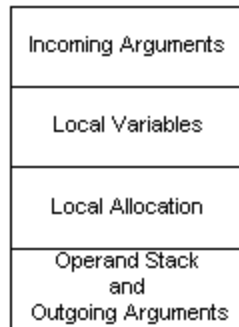


Figure 3: Method State

12.3 Method State

Method state describes the environment within which a method executes. (In conventional compiler terminology, it corresponds to a superset of the information captured in the “invocation stack frame”). The CLI method state consists of the following items:

- An instruction pointer (**IP**). This points to the next CIL instruction to be executed by the CLI in the present method.
- An *evaluation stack*. The stack is empty upon method entry. Its contents are entirely local to the method and are preserved across call instructions (that’s to say, if this method calls another, once that other method returns, our evaluation stack contents are “still there”). The evaluation stack is not addressable. At all times it is possible to deduce which one of a reduced set of types is stored in any stack location (see the Evaluation Stack section).
- A *local variable array* (starting at index 0). Values of local variables are preserved across calls (in the same sense as for the evaluation stack). A local variable can hold any data type. However, a particular slot must be used in a type consistent way (where the type system is the one described in the Evaluation Stack section). Local variables are initialized to 0 before entry if the initialize flag for the method is set (see the Opaque Classes section). The address of an individual local variable can be taken using the **ldloca** instruction.
- An *argument array*. The values of the current method’s incoming arguments (starting at index 0). These can be read and written by logical index. The address of an argument can be taken using the **ldarga** instruction. The address of an argument is also implicitly taken by the **arglist** instruction for use in conjunction with typesafe iteration through variable-length argument lists.
- A *methodInfo* handle. This contains read-only information about the method. In particular it holds the signature of the method, the types of its local variables, and data about its exception handlers.
- A *local memory pool*. The CLI includes instructions for dynamic allocation of objects from the local memory pool (**localloc**). Memory allocated in the local memory pool is *addressable*. The memory allocated in the local memory pool is reclaimed upon method context termination.
- A *return state* handle. This handle is used to restore the method state on return from the current method. Typically, this would be the state of the method’s caller. This corresponds to what in conventional compiler terminology would be the *dynamic link*.
- A *security descriptor*. This descriptor is not directly accessible to managed code but is used by the CLI security system to record security overrides (**assert**, **permit-only**, and **deny**).

Note that we describe the four areas of the method state – incoming arguments array, local variables array, local memory pool and evaluation stack – as if logically distinct areas. This is important, since this is a specification of the CLI architecture. However, in practice, an implementation of the CLI may map these areas into one contiguous array of memory, held as a conventional stack frame on the underlying target architecture.

12.3.1 The Evaluation Stack

Associated with each method state is an evaluation stack. Most CLI instructions retrieve their arguments from the evaluation stack and place their return values on the stack. Arguments to other methods and their return values are also placed on the evaluation stack. When a procedure call is made the arguments to the called methods become the incoming arguments array (see [Local Variables and Arguments](#)) to the method. This may require a memory copy, or simply a sharing of these two areas by the two methods.

The evaluation stack is made up of slots that can hold any data type, including an unboxed instance of a value type. The type state of the stack (the stack depth and types of each element on the stack) at any given point in a program must be identical for all possible control flow paths. For example, a program that loops an unknown number of times and pushes a new element on the stack at each iteration would be prohibited.

While the CLI, in general, supports the full set of types described in [Supported Data Types](#), the CLI treats the evaluation stack in a special way. While some JIT compilers may track the types on the stack in more detail, the CLI only requires that values be one of:

- I8, an 8-byte signed integer
- I4, a 4-byte signed integer
- I, a signed integer of either 4 or 8 bytes, whichever is more convenient for the target architecture
- F, a floating point value (R4, R8, or other representation supported by the underlying hardware)
- &, a managed pointer
- O, an object reference
- *, a “transient pointer,” which can be used only within the body of a single method, that points to a value known to be in unmanaged memory (see the CIL Instruction Set specification for more details. * types are generated internally within the CLI; they are not created by the user).
- A user-defined value type

The other types are synthesized through a combination of techniques:

- Shorter integer types in other memory locations are zero-extended or sign-extended when loaded onto the evaluation stack; these values are truncated when stored back to their home location.
- Special instructions perform numeric conversions, with or without overflow detection, between different sizes and between signed and unsigned integers.
- Special instructions treat an integer on the stack as though it were unsigned.
- Instructions that create pointers which are guaranteed not to point into the memory manager’s heaps (e.g. **ldloca**, **ldarga**, and **ldsflda**) produce transient pointers (type *****) which can be used wherever a managed pointer (type **&**) or unmanaged pointer (type **U**) is expected.
- When a method is called, an unmanaged pointer (type **U** or *****) is permitted to match a parameter that requires a managed pointer (type **&**). The reverse, however, is *not* permitted since it would allow a managed pointer to be “lost” by the memory manager.
- A managed pointer (type **&**) can be explicitly converted to an unmanaged pointer (type **U**), although this is not verifiable and may produce a runtime exception.

12.3.2 Local Variables and Arguments

Part of each method state is an array that holds local variables and an array that holds arguments. Like the evaluation stack, each element of these arrays can hold any single data type or an instance of a value type. Both arrays start at 0 (that is, the first argument or local variable is numbered 0). The address of a local variable can be computed using the **ldloca** instruction, and the address of an argument using the **ldarga** instruction.

Associated with each method is metadata that specifies:

- whether the local variables and memory pool memory must be initialized when the method is entered
- the type of each argument and the length of the argument array (but see below for variable argument lists)
- the type of each local variable and the length of the local variable array.

The CLI inserts padding as appropriate for the target architecture. That is, on some 64-bit architectures all local variables may be 64-bit aligned, while on others they may be 8-, 16-, or 32-bit aligned. The CIL generator must make no assumptions about the offsets of local variables within the array. In fact, the CLI is free to reorder the elements in the local variable array, and different JITters may choose to order them in different ways.

12.3.3 Variable Argument Lists

The CLI works in conjunction with the class library to implement methods that accept argument lists of unknown length and type (“varargs methods”). Access to these arguments is through a typesafe iterator in the Class Library, called **System.ArgIterator**.

The CIL includes one instruction provided specifically to support the argument iterator, **arglist**. This instruction can be used only within a method that is declared to take a variable number of arguments. It returns a value that is needed by the constructor for a **System.ArgIterator** object. Basically, the value created by **arglist** provides access both to the address of the argument list that was passed to the method and a runtime data structure that specifies the number and type of the arguments that were provided. This is sufficient for the class library to implement the user visible iteration mechanism.

From the CLI point of view, varargs methods have an array of arguments like other methods. But only the initial portion of the array has a fixed set of types and only these can be accessed directly using the **ldarg**, **starg**, and **ldarga** instructions. The argument iterator allows access to both this initial segment and the remaining entries in the array.

12.3.4 Local Memory Pool

Part of each method state is a local memory pool. Memory can be explicitly allocated from the local memory pool using the **localloc** instruction. All memory in the local memory pool is reclaimed on method exit, and that is the only way local memory pool memory is reclaimed (there is no instruction provided to *free* local memory that was allocated during this method invocation). The local memory pool is used to allocate objects whose type or size is not known at compile time and which the programmer does not wish to allocate in the managed heap.

Because the local memory pool cannot be shrunk during the lifetime of the method, a language implementation cannot use the local memory pool for general-purpose memory allocation .

13 Control Flow

The CIL instruction set provides a rich set of instructions to alter the normal flow of control from one CIL instruction to the next.

- **Conditional and Unconditional Branch** instructions for use within a method, provided the transfer doesn’t cross a protected region boundary (see the Exception Handling section).
- **Method call** instructions to compute new arguments, transfer them and control to a known or computed destination method (see the Method Calls section).
- **Tail call** prefix to indicate that a method should relinquish its stack frame before executing a method call (see the Method Calls section).
- **Return** from a method, returning a value if necessary.
- **Method jump** instructions to transfer the current method’s arguments to a known or computed destination method (see the Method Calls section).

- **Exception-related** instructions (see the Exception Handling section). These include instructions to initiate an exception, transfer control out of a protected region, and end a filter, catch clause, or finally clause.

While the CLI supports arbitrary control transfers within a method, there are several restrictions that must be observed:

1. Control transfer is never permitted to enter a catch handler or finally clause (see the Exception Handling section) except through the exception handling mechanism.
2. Control transfer out of a protected region (see the Exception Handling section) is only permitted through an exception instruction (**leave**, **end.filter**, **end.catch**, or **end.finally**).
3. The evaluation stack must be empty after the return value is popped by a **ret** instruction.
4. All slots on the stack must have the same data type at every point within the method body, regardless of the control flow that allows execution to arrive there.
5. In order for the JIT compilers to efficiently track the data types stored on the stack, the stack must normally be empty at the instruction following an unconditional control transfer instruction (**br**, **br.s**, **ret**, **jmp**, **throw**, **end.filter**, **end.catch**, or **end.finally**). The stack is allowed to be non-empty at this point only if at some earlier location within the method there has been a forward branch to that location.
6. Control is not permitted to simply “fall through” the end of a method. All paths must terminate with one of these instructions: **ret**, **throw**, **jmp**, or (**tail.** followed by **call**, **calli**, or **callvirt**).

14 Method Calls

An important design goal of the CLI is to abstract the layout of native method frames, including calling convention. That is, instructions emitted by the CIL code generator contain sufficient information for different implementations of the CLI to use different native calling convention. All method calls initialize the method state areas (see the Method State section) as follows:

1. The incoming arguments array is set by the caller to the desired values.
2. The local variables array always has **null** for Object types and for fields within value types that hold objects. In addition, if the “zero init flag” is set in the method header, then it is initialized to 0 for all integer types and 0.0 for all floating point types. Value Types are not initialized by the CLI, but verified code will supply a call to an initializer as part of the method’s entry point code.
3. If the “zero init flag” is set in the method header the local memory pool is initialized to all zeros.
4. The evaluation stack is empty.

14.1 Call Site Descriptors

To support this flexibility, call sites need additional information that enables an interpreter or JIT compiler to synthesize any native calling convention. All CIL calling instructions (**call**, **calli**, and **callvirt**) include as part of the instruction a description of the call site. This description can take one of two forms. The simpler form, used with the **calli** instruction, is a “call site description” (represented as a metadata token for a stand-alone call signature) that provides:

- The number of arguments being passed.
- The data type of each argument.
- The order in which they have been placed on the call stack.
- The native calling convention to be used

The more complicated form, used for the **call** and **callvirt** instructions, is a “method reference” (a metadata **methodref** token) that augments the call site description with an identifier for the target of the call instruction.

14.2 Calling Instructions

The CIL has three call instructions that are used to transfer new argument values to a destination method. Under normal circumstances, the called method will terminate and return control to the calling method.

- **call** is designed to be used when the destination address is fixed at the time the CIL is linked. In this case, a method reference is placed directly in the instruction. This is comparable to a direct call to a static function in C. It can be used to call static or instance methods or the (statically known) superclass method within an instance method body.
- **calli** is designed for use when the destination address is calculated at run time. A method pointer is passed on the stack and the instruction contains only the call site description.
- **callvirt**, part of the CIL common type system instruction set, uses the class of an object (known only at runtime) to determine the method to be called. The instruction includes a method reference, but the particular method isn't computed until the call actually occurs. This allows an instance of a subclass to be supplied and the method appropriate for that subclass to be invoked. The **callvirt** instruction is used both for instance methods and methods on interfaces. For further details, see the Common Type System specification and the CIL Instruction Set specification.

In addition, each of these instructions can be immediately preceded by a **tail.** instruction prefix. This specifies that the calling method terminates with this method call (and returns whatever value is returned by the called method). The **tail.** prefix instructs the JIT compiler to discard the caller's method state prior to making the call (if the call is from untrusted code to trusted code the frame cannot be fully discarded for security reasons). When the called method executes a **ret** instruction, control returns not to the calling method but rather to wherever that method would itself have returned (typically, return to caller's caller). Notice that the **tail.** instruction shortens the lifetime of the caller's frame so it is unsafe to pass managed pointers (type **&**) as arguments.

Finally, there are two instructions that indicate an optimization of the **tail.** case:

- **jmp** is followed by a **methodref** or **methoddef** token and indicates that the current method's state should be discarded, its arguments should be transferred intact to the destination method, and control should be transferred to the destination. The signature of the calling method must exactly match the signature of the destination method.

14.3 Computed Destinations

The destination of a method call can be either encoded directly in the CIL instruction stream (the **call** and **jmp** instructions) or computed (the **callvirt**, and **calli** instructions). The destination address for a **callvirt** instruction is automatically computed by the CLI based on the method token and the value of the first argument (the **this** pointer). The method token must refer to a virtual method on a class that is a direct ancestor of the class of the first argument. The CLI computes the correct destination by, effectively, locating the nearest ancestor of the first argument's class that supplies an implementation of the desired method (the implementation can be assumed to be more efficient than the linear search implied here).

For the **calli** instruction the CIL code is responsible for computing a destination address and pushing it on the stack. This is typically done through the use of a **ldftn** or **ldvirtfn** instruction at some earlier time. The **ldftn** instruction includes a metadata token in the CIL stream that specifies a method, and the instruction pushes the address of that method. The **ldvirtfn** instruction takes a metadata token for a virtual method in the CIL stream and an object on the stack. It performs the same computation described above for the **callvirt** instruction but pushes the resulting destination on the stack rather than calling the method.

The **calli** instruction includes a call site description that includes information about the native calling convention that should be used to invoke the method. Correct CIL code shall specify a calling convention specified in the **calli** instruction that matches the calling convention for the method that is being called.

14.4 Virtual Calling Convention

The CIL provides a "virtual calling convention" that is converted by the JIT into a native calling convention. The JIT determines the optimal native calling convention for the target architecture. This allows the native calling convention to differ from machine to machine, including details of register usage, local variable homes, copying conventions for large call-by-value objects (as well as deciding, based on

the target machine, what is considered “large”). This also allows the JIT to reorder the values placed on the CIL virtual stack to match the location and order of arguments passed in the native calling convention.

The CLI uses a single uniform calling convention for all method calls. It is the responsibility of the JITters to convert this into the appropriate native calling convention. The virtual calling convention is:

1. If the method being called is an instance method (class or interface) or a virtual method, first push the **this** pointer. For methods on Objects (including boxed value types), the **this** pointer is of type **O** (object reference). For methods on value types, the **this** pointer is provided as a by-ref parameter; that is, the value is a pointer (managed, **&**, or unmanaged, ***** or **I**) to the instance.
2. Push the remaining arguments in left-to-right order (that is, push the lexically, leftmost argument first). The Parameter Passing section describes how each of the three parameter passing conventions (by-value, by-reference, and typed reference) should be implemented.
3. Execute the appropriate call instruction (**call**, **calli**, or **callvirt** any of which may be preceded by **tail.**).

14.5 Parameter Passing

The CLI supports three kinds of parameter passing, all indicated in metadata as part of the signature of the method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed by-value while all others are passed by-ref). Parameter may be passed as follows:

- **By-value** parameters, where the **value** of an object is passed from the caller to the callee.
- **By-ref** parameters, where the **address** of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.
- **Typed reference** parameters, where a runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.

It is the responsibility of the CIL generator to follow these conventions. The verifier checks that the types of parameters match the types of values passed, but is otherwise unaware of the details of the calling convention.

14.5.1 By-Value Parameters

For primitive types (integers, floats, etc.) the caller copies the value onto the stack before the call. For Objects the object reference (type **O**) is pushed on the stack. For managed pointers (type **&**) or unmanaged pointers (type **U**), the address is passed from the caller to the callee. For value types, the protocol described in the Operations on Value Type Instances section is used.

14.5.2 By-Ref Parameters

By-Ref Parameters are the equivalent of C++ reference parameters or PASCAL **var** parameters: instead of passing as an argument the value of a variable, field, or array element, its address is passed instead; and any assignment to the corresponding parameter actually modifies the corresponding caller’s variable, field, or array element. Much of this work is done by the higher-level language, which hides from the user the need to compute addresses to pass a value and the use of indirection to reference or update values.

Passing a value by reference requires that the value have a home (see the Homes for Values section) and it is the address of this home that is passed. Constants, and intermediate values on the evaluation stack, cannot be passed as by-ref parameters because they have no home.

The CLI provides instructions to support by-ref parameters:

- calculate addresses of home locations (see [Table 8: Address and Type of Home Locations](#))
- load and store primitive data types through these address pointers (**ldind.***, **stind.***, **ldfld**, etc.)
- copy value types (**ldobj** and **cpobj**).

Some addresses (e.g., local variables and arguments) have lifetimes tied to that method invocation. These cannot be referenced outside their lifetimes, and so they should not be stored in locations that last

beyond their lifetime. The CIL does not (and cannot) enforce this restriction, so the CIL generator must enforce this restriction or the resulting CIL will not work correctly. For code to be verifiable (see the Verification section) by-ref parameters may **only** be passed to other methods or referenced via the appropriate **stind** or **ldind** instructions.

14.5.3 Typed Reference Parameters

By-ref parameters and value types are sufficient to support statically typed languages (C++, Pascal, etc.). They also support dynamically typed languages that pay a performance penalty to box value types before passing them to polymorphic methods (Lisp, Scheme, SmallTalk, etc.). Unfortunately, they are not sufficient to support languages like Visual Basic that require by-reference passing of unboxed data to methods that are not statically restricted as to the type of data they accept. These languages require a way of passing *both* the address of the home of the data *and* the static type of the home. This is exactly the information that would be provided if the data were boxed, but without the heap allocation required of a box operation.

Typed reference parameters address this requirement. A typed reference parameter is very similar to a standard by-ref parameter but the static data type is passed as well as the address of the data. Like by-ref parameters, the argument corresponding to a typed reference parameter must have a home. If it were not for the fact that the verifier and the memory manager must be aware of the data type and the corresponding address, a by-ref parameter could be implemented as a standard value type with two fields: the address of the data and the type of the data. Like a regular by-ref parameter, a typed reference parameter can refer to a home that is on the stack, and that home will have a lifetime limited by the call stack. Thus, the CIL generator must apply appropriate checks on the lifetime of by-ref parameters; and the verifier imposes the same restrictions on the use of typed reference parameters as it does on by-ref parameters (see the By-Ref Parameters section).

A typed reference is passed by either creating a new typed reference (using the **mkrefany** instruction) or by copying an existing typed reference. Given a typed reference argument, the address to which it refers can be extracted using the **refanyval** instruction; the type to which it refers can be extracted using the **refanytype** instruction.

14.5.4 A Note on Interactions

A given parameter can be passed using any one of the parameter passing conventions: by-value, by-ref, or typed reference. No combination of these is allowed for a single parameter, although a method may have different parameters with different calling mechanisms.

There are a pair of non-obvious facts about the parameter passing convention:

A parameter that has been passed in as typed reference cannot be passed on as by-ref or by-value without a runtime type check and (in the case of by-value) a copy.

A by-ref parameter can be passed on as a typed reference by attaching the static type.

Table 9: Parameter Passing Conventions illustrates the parameter passing convention used for each data type.

Table 9: Parameter Passing Conventions

Type of data	Pass By	How data is sent
Built-in value type (int, float, etc.)	Value	Copied to called method, type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method
User-defined value type	Value	Called method receives a copy; type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method

Object	Value	Reference to data sent to called method, type statically known and class available from reference
	Reference	Address of reference sent to called method, type statically known and class available from reference
	Typed reference	Address of reference sent to called method along with static type information, class (i.e. dynamic type) available from reference

15 Exception Handling

The CLI supports an exception handling model based on the idea of exception objects and protected blocks of code. When an exception occurs, an object is created to represent the exception. All exceptions objects are instances of some class (i.e. they can be boxed value types, but not pointers, unboxed value types, etc.). Users can create their own exception classes, typically by subclassing **System.Exception**.

There are four kinds of handlers for protected blocks. A single protected block can have exactly one handler associated with it:

1. A **finally handler** which must be executed whenever the block exits, regardless of whether that occurs by normal control flow or by an unhandled exception.
2. A **fault handler** which must be executed if an exception occurs, but not on completion of normal control flow.
3. A **type-filtered handler** that handles any exception of a specified class or any of its sub-classes.
4. A **user-filtered handler** that runs a user-specified set of CIL instructions to determine whether the exception should be ignored (i.e. execution should resume), handled by the associated handler, or passed on to the next protected block.

Protected regions, the type of the associated handler, and the location of the associated handler and (if needed) user-supplied filter code are described through an Exception Handler Table associated with each method. The exact format of the Exception Handler Table is specified in detail in the [File Format Specification](#). Details of the exception handling mechanism are specified in the [Exception Specification](#).

15.1 Exceptions Thrown by the CLI Itself

CLI instructions can throw the following exceptions as part of executing individual instructions. The documentation on a particular instruction will list all the exceptions the instruction can throw (except for the general purpose ExecutionEngineException described below that can be generated by all instructions).

Base Instructions

ArithmeticException
 DivideByZeroException
 ExecutionEngineException
 InvalidAddressException
 OverflowException
 SecurityException
 StackOverflowException

Object Model Instructions

TypeLoadException
 IndexOutOfRangeException
 InvalidAddressException
 InvalidCastException

MissingFieldException
MissingMethodException
NullReferenceException
OutOfMemoryException
SecurityException
StackOverflowException

The `ExecutionEngineException` is special. It can be thrown by any instruction and indicates an unexpected inconsistency in the CLI. Code that has been passed through the code verifier should never throw this exception (it is a defect in either the verifier or the CLI if it does). However, unverified code can cause this error if the code is corrupt or inconsistent in some way.

Note that, because of the verifier, there are no exceptions for things like ‘`MetaDataTokenNotFound`.’ The verifier can detect this inconsistency before the instruction is ever executed (the code is then considered unverified). If the code has not been verified, this type of inconsistency would raise the generic `ExecutionEngineException`.

Exceptions can also be thrown by the CLI, as well as by user code, using the **throw** instruction. The handing of an exception is identical, regardless of the source.

15.2 Overview of Exception Handling

See the [Exception Handling specification](#) for details.

Each method in an executable has associated with it a (possibly empty) array of exception handling information. Each entry in the array describes a protected block, its filter, and its handler (which may be a **catch** handler, a **finally** handler, or a **fault** handler). When an exception occurs, the CLI searches the array for the first protected block that

- Protects a region including the current instruction pointer *and*
- Is a catch handler block *and*
- Whose filter wishes to handle the exception

If a match is not found in the current method, the calling method is searched, and so on. If no match is found the CLI will dump a stack trace and abort the program. If a match is found, the CLI walks the stack back to the point just located, but this time calling the **finally** and **fault** handlers. It then starts the corresponding exception handler. Stack frames are discarded either as this second walk occurs or after the handler completes, depending on information in the exception handler array entry associated with the handling block.

Some things to notice are:

- The ordering of the exception clauses in the Exception Handler Table is important. If handlers are nested, the most deeply nested try blocks must come before the try blocks that enclose them.
- Exception handlers can access the local variables and the local memory pool of the routine that catches the exception, but any intermediate results on the evaluation stack at the time the exception was thrown are lost.
- An exception object describing the exception is automatically created by the CLI and pushed onto the evaluation stack as the first item upon entry of a filter or catch clause.
- Execution cannot be resumed at the location of the exception. This restriction may be relaxed in the future.

15.3 CIL Support for Exceptions

The CIL has special instructions to:

- **Throw** and **rethrow** a user-defined exception.

- **Leave** a protected block and execute the appropriate **finally** clauses within a method, without throwing an exception. This is also used to exit a **catch** clause. Notice that leaving a protected block does **not** cause the fault clauses to be called.
- End a user-supplied filter clause (**endfilter**) and return a value indicating whether to handle the exception.
- End a finally clause (**endfinally**) and continue unwinding the stack.

15.4 Lexical Nesting of Protected Blocks

This section summarizes restrictions that are described in detail in the [Exception Handling Specification](#).

The following restrictions below refer to the *lexical* nesting of **try** blocks and their associated handlers:

1. A **try** block may have associated with it any *one* of the following:
 - a **catch** block (with an implied filter based on the type of the exception)
 - a **filter** block and a **catch** block
 - a **finally** block
 - a **fault** block
2. A single **try** block, **filter** block, **catch** block, **fault** block, or **finally** block must constitute a contiguous block of CIL instructions.
3. The exception table fully specifies the range of the **try**, **catch**, **fault** and **finally** blocks, but only specifies the entry point for the **filter** block. The **filter** block lexically ends with the (required and unique) **endfilter** instruction for that block.
4. Multiple **try** blocks that specify precisely the same range of instructions are considered to be a single try block with multiple associated handler blocks. The associated handler blocks must be either **catch** blocks or **filter** and **catch** blocks, in any combination. There cannot be any **finally** or **fault** blocks associate with these **try** blocks. (To model a source-level construct that has, for example, a **try** with two associated **catch** handlers and a **finally**, you must have three entries: two **try/catch** entries specifying the same region of instructions, and an enclosing **try/finally** that covers both the other **try** block *and* their handlers.)
5. The addresses included in **catch** blocks, **filter** blocks, **fault** blocks, and **finally** blocks must not overlap one another, nor can any handler block be shared between multiple **try** blocks.
6. The region of CIL instructions associated with a **try** block cannot include its own **filter**, **fault**, **finally**, or **catch** block (i.e. it is just the protected code, not the handlers that are associated with it).
 7. A block of any kind (except the **try** block associated with a **fault** block) that encloses a **try** block must include all of the code associated with the inner **try** as well as the handlers associated with the inner **try**.
 8. A **try** block that has an associated **fault** block may overlap another **try** block that has a **fault** block (but the **fault** blocks themselves may not overlap one another).
 9. A **try** block cannot appear within a **filter** block.

15.5 Control Flow Restrictions on Protected Blocks

The following restrictions are about the control flow into, out of, and between **try** blocks and their associated handlers.

1. Correct CIL code must not enter a **filter**, **catch**, **fault** or **finally** block except through the CLI exception handling mechanism.
2. There are only two ways to enter a **try** block from outside its lexical body:

- a) **Branching to or falling into the try block's first instruction.** The branch can be made using a conditional branch, an unconditional branch, or a **leave** instruction.
 - b) **Using a leave instruction within the catch block associated with the try.** In this case correct CIL code can branch to any address within the **try** block, not just its first instruction.
3. Upon entry to a **try** block the evaluation stack must be empty.
 4. The only ways correct CIL code can leave a **try**, **filter**, **catch** or **finally** block are as follows:
 - a. **throw** from any of them.
 - b. **leave** from the body of a **try** or **catch** (in this case the destination of the **leave** must have an empty evaluation stack and the **leave** instruction has the side-effect of emptying the evaluation stack).
 - c. **endfilter** may appear only as the lexically last instruction of a **filter** block, and it must always be present (even if it is immediately preceded by a **throw** or other unconditional control flow). If reached, the evaluation stack must contain an **I4** when the **endfilter** is executed, and the value is used to determine how exception handling should proceed.
 - d. **endfinally** from anywhere within a **finally**, with the side-effect of emptying the evaluation stack.
 - e. **rethrow** from within a **catch** block, with the side-effect of emptying the evaluation stack.
 - f. fall through the end of a **try** block (falling through the end of any other kind of block is not permitted).
 5. When the try block is exited with a leave instruction, the evaluation stack must be empty for correct CIL.
 6. When a catch or filter clause is exited with a leave instruction, the evaluation stack must be empty for correct CIL. This involves popping off the exception object from the evaluation stack which was automatically pushed onto the stack.
 7. Correct CIL code must not exit any block using a **ret** instruction.

16 Atomicity of Memory Accesses

The CLI makes several assumptions about atomicity of memory references, and these translate directly into rules required of either programmers or translators from high-level languages into CIL.

- Read and write access to word-length memory locations (types **I** and **U**) that are properly aligned is atomic. Correct translation from CIL to native code requires generation of native code sequences that supply this atomicity guarantee. Note that there is no guarantee about atomic update (read-modify-write) of memory.
- Read and write access to 4-byte data (**I4** and **U4**) that is aligned on a 4-byte boundary is atomic, even on a 64-bit machine. Again, there is no guarantee about atomic read-modify-write.
- One- and Two-byte data that does not cross a word boundary will be read atomically, but writing will write the entire word back to memory.
- No other memory references are performed atomically.

When the CLI controls the layout of managed data, it pads the data so that if an object starts at a word boundary all of the fields that require 4 or fewer bytes will be aligned so that reads will be atomic. The managed heap always aligns data that it allocates to maintain this rule, so heap references (type **O**) to data that does not have explicit layout will occur atomically where possible. Similarly, static variables of managed classes are allocated so that they, too, are aligned when possible. The CLI aligns stack frames to word boundaries, but does not attempt to align to an 8-byte boundary on 32-bit machines even if the frame contains 8-byte values.

17 **OptIL: An Instruction Set Within CIL**

A fundamental issue associated with generating CIL is how much of the work is done by the CIL generator and how much of the work is done by the CLI (via a JIT compiler). The CIL instruction set was designed to be easy for compilers to generate so that CIL can be generated quickly in rapid application development (RAD) environments, where compile speed and ease of debugging are at a premium.

On the other hand, in situations where load time is important, it is useful to do as much work as possible in the code generator, before the executable is loaded. In particular it is useful to do expensive optimizations like common sub-expression elimination, constant folding, loop restructuring, and even register allocation in the code generator (as would be done in a traditional compiler). The instruction set should be able to represent such optimized code as well.

Finally, in some environments it is important that the JITter be small and run in a nearly constant amount of memory, even for large methods. The instruction set should allow a compiler to compute information and pass it on to the JITter that will reduce the memory required by the JITter (e.g., register allocation and branch targets).

In the CLI environment, an optimizing compiler can best express many optimizations by generating OptIL. OptIL is optimized code represented using the same CIL instruction set; however, OptIL differs from non-OptIL code in the following ways

- Many transformations will have been done (e.g., loop restructuring, constant folding, CSE).
- The code will obey certain conventions (e.g., method calls are not nested).
- There will be additional annotations (e.g., exactly when each variable is used for the last time).

The exact restrictions an executable must satisfy to be of this form are described in the “[Opt-CIL Specification](#)”. The “CIL Instruction Set” specification contains a detailed description of each of the CIL instructions.

Note that an OptIL program is still a valid CIL program (it can be run by the normal CLI), but because it has been optimized by the code generator it can be compiled to native code very quickly and using little memory.

18 Index

abstract	18, 21, 24	CLS frameworks	8
abstract class	1	CLS-compliant	31
access	8	consumer tool	31
accessible	8	extender tool	32
.....	7	39
.....	7	39
application	29	39
application domain	29, 39	coercion	4
.....	39	COFF	See PE
array element	5	Common Language Specification	See CLS
array type	5, 15	component metadata	26
assembly	7, 9, 18, 26, 28, 39	compound type	5
assembly dependency	28	concrete	18
assembly scope	7	11
assignment compatibility	11	constructor	20
assignment compatible	4, 7	39
autolayout	28	contract	9, 10, 27
behavior	1	conversions	
box	3	explicit	4
boxed type	3	implicit	4
boxed value	3	39
byref	12	39
by-ref	41	39
cast	4	39
explicit	36	27
implicit	36	EAT (Export Address Table)	31
narrowing	36	27
widening	36	9
casting	4	enclosing type	10
class contract	10	enum	7
class definitions	17	enumeration type	7
.....	39	equal	3
class type	10, 17	equality	3, 4
CLS	6, 8, 31	event contract	11
CLS consumer languages	8	exact type	2, 5
CLS extender languages	8	27

- expect existing slot 23
- explicitlayout 28
- exportable..... 9
- family..... 9
- family and assembly 9
- family or assembly 9
- field..... 5
- final 22, 24
- fully describe 2
- garbage collection 7
- getter 25
- global statics 31
- hide..... 22, 23, 24, 30
 - by name 22
 - by name and signature 22
 - 39
- IAT (Import Address Table) 31
- identical..... 3
- identity 3, 4, 5
 - 39
- implicit type 14
- indexed property See property, indexed
- inherit 1, 10
- inheritance demand..... 10
- inherits 5
- init-only 12
- init-only constraint 12
- instance field 5
- interface contract 10
- interface definition 16
- interface type 1, 2, 10
- Intermediate Language..... See CIL
 -
 -
- kind..... 7
- layout 22
- layoutsequential 28
- literal 8, 12

- literal constraint..... 12
- local signature 12
- location..... 4
- location constraint..... 12
- location signature..... 12
- managed code 7
- managed data..... 7
- managed pointer 12
- manifest..... 26, 29
- marshalling 27
- member 5
- member scope 7
- member signatures..... 27
- messages 1
- metadata 26
- metadata token..... 27
- method 2
- method contract..... 11
- method signature 13
- Common Intermediate Language..... See CIL
 - 39
 - 39
- names
 - special 34
- narrowing..... 4, 36
- nested 9
- nested type 9, 10
- new slot 23
- null 5, 6
 - 27
- object 2
- object type 1, 2, 18
- OOP..... See Programming, Object-Oriented
- optional modifiers..... 30
- overriding 5, 22
- parameter signature 13
- partial description 1
- PE 26

PEVerify.....	27	serializable.....	24
pointer type	1, 16	setter	25
.....		signature	10, 11
.....	39	static.....	24
Private	9	static field.....	5
.....	39	static method	5, 6
.....	39	System.Enum.....	21
programming		System.TypedReference..	See typed reference
object-oriented.....	9	System.ValueType	21
property		this	24
indexed	37	this pointer	5, 6, 24
property contract	11	type	1
.....	39	type alias.....	7
public.....	9, 18	type definer.....	14
publicly accessible parts	31	type definition	7
qualified name.....	7	7
qualify	7	type name	7
.....	28	type safety.....	6, 14, 39
rank.....	15	type signature	11
.....		typed reference	13
reference	29	8
reference demand.....	10	typedref.....	See typed reference
reference type	1	typeless programming	1
referenced entity	8	typesafe	14
referent	8	unbox.....	3
.....	27	uniqueness	
Relative Virtual Address.....	See RVA	name.....	7
.....	39	validation	
representation	1	metadata	27
required modifiers.....	30	value type.....	1
RVA	28	varargs.....	13
scope	7, 18	28
sealed	21	28
.....	39	vector.....	15, 39
.....	39	verification	7, 14, 39
.....	39	39
self-describing.....	26	VES.....	See Virtual Execution System
		virtual	6, 24

virtual call	5, 6
Virtual Execution System	6
virtual method	6	
visibility	18	volatile constraint..... 12
visible.....	8	widening
		4, 36

Free printed copies can be ordered from:

ECMA

114 Rue du Rhône

CH-1204 Geneva

Switzerland

Fax: +41 22 849.60.01

Email: documents@ecma.ch

Files of this Standard can be freely downloaded from the ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.