LutaML — Language reference

RS 3001:2025, Version 1.0

Ronald Tse

Unrestricted
March 10, 2024
RS 3001:2025, Version 1.0
LutaML

Ribose

Contents

Ribose Inc.

167-169 Great Portland Street
5th Floor
London
W1W 5PF
United Kingdom

copyright@ribose.com
www.ribose.com

Introduction

LutaML is a language for defining information models and instances of those information models.

Traditionally, different languages have been developed to define information models for disparate purposes.

- Data modeling languages are used to define and describe information models in a structured and formalized manner to document and interchange such model designs. Models defined in these formats are not meant to be directly usable in programming languages or serialization formats.
- Schema languages are used to define serialization formats that can be used to serialize and deserialize data in a specific format. These formats are not meant to be directly usable in programming languages or information models.
- Serialization formats are used to serialize and deserialize data according to some schema or in a "schema-less" model. These formats are not directly useable as information models.
- Programming language constructs are used to define data structures that can be used in software applications. These constructs are not directly useable as information models or serialization formats nor communicated across systems.

EXAMPLE: Traditionally, multiple layers of languages and tools are used to work with information models:

- UML for defining information models
- XML Schema or JSON Schema for defining serialization formats
- Programming language implementations for both the models and their serialization mechanisms

LutaML is a single language that incorporates all the above roles of information modeling.

The goal of LutaML is to provide a single language that can be used to:

- define information models that are human-readable and interchangeable
- specify serialization formats of information models
- parse, validate, and utilize those information models in programming languages

1. Scope

This document defines the core syntax and semantics of the Luta Modeling Language (LutaML), a domain-specific language for defining data models and instances of those data models.

Other aspects of the LutaML language are defined in separate documents.

2. Normative references

There are no normative references in this document.

3. Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1. information model PREFERRED

representation of information in a structured manner

3.2. model PREFERRED

abstract representation of a category of information

Note 1 to entry: Commonly, a collection of classes, enums, and relationships defining a data structure, facilitating the organization of information and its metadata.

3.3. class PREFERRED

defined characteristics of a type of object, including attributes and constraints

3.4. enum PREFERRED

predefined set of values

3.5. instance PREFERRED

data conforming to a class definition

3.6. attribute PREFERRED

characteristic of a property of a *class* (3.3)

Note 1 to entry: Commonly defined by a name, type, and cardinality

3.7. reference PREFERRED

relationship between instances

Note 1 to entry: Can be implemented as pointer to another instance or attribute.

3.8. cardinality PREFERRED

allowed number of values for an attribute

Note 1 to entry: Commonly expressed as a range (e.g., 0..1, 1..n).

4. Purpose

4.1. Data modeling

Information is stored and used as data in digital systems. Digital systems rely on defined structures to store, retrieve and manipulate data in a consistent manner.

A data model is a structured representation of the data that describes the properties of the data, the relationships between different data elements, and any rules or constraints that govern the data.

Data modeling is the process of creating a data model.

Many data modeling languages exist, each with its own syntax and semantics.

EXAMPLE 1: UML (Unified Modeling Language) is a general-purpose modeling language that can be used to model a wide variety of information.

EXAMPLE 2: EXPRESS is a data modeling language designed to represent information requirements.

Commonly languages that define data models are called “conceptual modeling languages”.

While many data modeling languages are used to define data models in a way that is independent of underlying technologies used, most data modeling languages do not provide a way to define how data is represented in an information system.

For example, information models in UML are mostly used for diagramming and documentation purposes, and do not provide a way to define how the data can be accessed or used in programming languages.

4.2. Serialization

Serialization is the process of converting data structures or objects into a format that can be stored or transmitted between systems. Typically, this involves converting the data into a string of bytes that can be written to a file or sent over a network.

Serialization formats are very important in allowing systems to exchange structured information in an interoperable manner without loss of semantics.

EXAMPLE 1: Common serialization formats include JSON, XML, and YAML.

Serialization formats often come with a schema language that allows for the definition of particular data structures that data needs to conform to.

EXAMPLE 2: XML Schema, JSON Schema, and YAML Schema represent the schema languages for XML, JSON, and YAML, respectively.

Typically, languages that define serialization formats are called “schema languages”.

A conceptual model may be realized (“implemented”) in multiple serialization formats. For example, some standards define a conceptual model in UML and provide manually-defined transformation rules to XML and JSON.

4.3. Access in programming languages

Information defined by conceptual data models and serialization schema languages are not directly usable in a native manner in programming languages or APIs.

At the conceptual data model level, there is no direct mapping to programming language constructs. For example, a UML class diagram does not directly map to a Java class.

There are tools that attempt to generate code from UML models, yet the generated code will require manual integration and intervention to be usable in software.

At the serialization schema language level, there is again no direct access from the programming language to the schema language constructs. For example, a JSON Schema does not directly map to a Java class. Typically developers will write code that parses the schema and/or implement code to read and write data in the format defined by the schema.

The lack of direct access to data models and serialization formats in programming languages makes it difficult to work with structured data in a consistent manner across different systems and technologies.

4.4. Interleaving conceptual modeling and serialization format modeling

Conceptual modeling focuses on the definition of the existence and relationships of data elements.

Serialization data modeling focuses on low-level details of how data is represented in a particular format.

For example, a conceptual model may define that a `Date` object has `year`, `month`, `day` attributes, and each of them are of an `Integer` type and potentially the numbering constraints for each attribute, and that there are certain restrictions to what the attributes can be at an attribute level (valid values of `month` range from 1 to 12), or at the object level (only a leap year is allowed to have 29 days in February). It does

not specify how many bytes are to be used to represent an `Integer` or how the `Date` object is to be written in a file.

A serialization data model (using a schema language) may define that a `Date` object is represented as a sequence of 3 Integers, each representing the `year`, the `month`, and the day, respectively, and that the `Integer` type is represented as a 4-byte unsigned integer. While the schema language may support value constraints that the `month` attribute must be between 1 and 12, it is unlikely that the schema language supports object-level complex constraints such as "only a leap year is allowed to have 29 days in February". Schema languages typically do not have the expressive power to define such complex constraints.

This presents a conundrum for the data modeler, who needs to define the entire structure and constraints, but also needs to decide where and how to define constraints that may not be definable in a schema language. This complexity explodes exponentially as the need for the same data to be represented in additional schema languages arises.

How much of the constraints should be defined in the conceptual model? How much of the data model constraints should be defined in the serialization schema language? Do we need to define duplicated constraints in both the conceptual model and the serialization schema language if allowed? This is a question that haunts developers and data modelers alike.

4.5. Challenges in data interchange

4.5.1. General

Interchange of data between systems of a particular data model can be challenging due to multiple factors.

4.5.2. Derivation of serialization formats from conceptual models

A successful, modern conceptual model is expected to support multiple serialization formats, given the variety of modern systems and tech stack components.

While it is possible to define a conceptual model in a data modeling language, and then manually define transformation rules to convert the model into serialization formats, this is a time-consuming and error-prone process.

This issue is exacerbated by the fact that the transformation rules are not always well-defined and may not be easily maintained. A change in the conceptual model may require changes in multiple serialization formats, and it is difficult to maintain synchronicity between the conceptual model and the serialization formats.

The solution to this problem is to define a single source of truth for the data model, and then automatically derive the serialization formats from the data model.

Even with automatic derivation, there are additional concerns (addressed in this document):

- Versioning: How do we version the data model and the serialization formats?

- Migration: How do we migrate data from one version of the data model to another?
- Interoperability: How do we ensure that data can be exchanged between systems that use different versions of the data model?
- Access: How do we work with different versions of the data model in a programming language?

4.5.3. Complexity

Modern technical stacks are composed of multiple systems that are built using different technologies and data modeling languages.

Browser-based applications may use JSON to represent data, while server-side applications may use XML. Databases may use SQL to define data structures, while APIs may use JSON Schema to define data structures.

4.5.4. Model definition language incompatibility

Different data modeling languages have different syntax and semantics. This makes it difficult to interchange data between systems that are defined using different data modeling languages.

For example, a data model defined in UML may not be directly convertible to a data model defined in EXPRESS.

When a schema language is used to define data structures, the schema language itself may not be directly convertible to another schema language.

For example, an XML Schema may not be directly convertible to a JSON Schema.

Typical approaches used today include:

- create a transformation toolchain that convert serialized formats to a defined serialization format, then use a codebase that can parse the defined serialization format
- create a transformation toolchain that convert serialized formats to an intermediary representation, and then use a codebase that reads that intermediary representation.

Imagine that there are 5 serialization formats that need to be supported.

Neither of these approaches is ideal, as they require additional work to be done, and may introduce errors in the transformation process.

4.5.5. Semantic incompatibility of conceptual modeling languages

4.5.5.1. General

Some modeling languages are at their core, incompatible with each other due to their different foundational principles and structures.

4.5.5.2. Comparing UML and EXPRESS

Both UML and EXPRESS are conceptual modeling languages intended for general use.

However, some UML concepts are not directly mappable to EXPRESS concepts:

- A UML class can have "methods", but an EXPRESS `ENTITY` allows only validation logic (via `WHERE`, `RULES`, or `FUNCTION`/`PROCEDURE`), which is declarative and stateless.
- EXPRESS `SELECT` types (flexible polymorphic unions) have no direct UML equivalent. UML generalizations or interfaces approximate `SELECT` but lose EXPRESS's runtime validation semantics.

4.5.5.3. Comparing UML and RDF

UML is a class-based modeling language, while RDF is a graph-based modeling language.

- The concept of encapsulation (private/public attributes) in UML is not supported in RDF.
- The property multiplicity constraints (e.g., 1..*) cannot be enforced in RDF.
- RDF applies graph-based relationships (arbitrary connections between resources) that are incompatible with the hierarchical class/object structure used in UML.

4.5.5.4. Approach

It is clear that a direct mapping between certain conceptual model languages is impossible without losing some semantics.

Some communities have attempted to define mappings between conceptual model languages, which often become rather complex as re-implementations of the modeling language in another language. The result is a loss of semantics, expressivity, and deviation from the conventions, benefits, and usability of the modeling language.

The solution to this problem is to take a common approach that works across different conceptual modeling languages, and then define additional rules for each language to supplement the common approach.

4.5.6. Conversion difficulties between serialization formats

4.5.6.1. General

Some serialization formats are not directly mappable to each other, due to the differences in syntax and semantics between the formats.

This means that a document in one serialization format cannot be directly converted to another serialization format without modification.

These features are critical in data representation and serialization contexts.

4.5.6.2. Comparing YAML Schema and JSON Schema

Though YAML Schema is derived from JSON Schema, there are differences between the two serialization formats that make direct conversion difficult without additional handling. These include:

- Content-level semantic differences
  - Data type mismatches. YAML supports data types not native to JSON, such as date-time, symbols, and binary data. As a result, a defined transform needs to be implemented to round-trip these data types between YAML and JSON.

    EXAMPLE 1: A value that is of date-time type in YAML does not have a direct equivalent in JSON. Such value is likely to be represented as a string in JSON. In a round-trip operation, the JSON-to-YAML transform needs to detect that the string is a date-time value and convert it back to a date-time value in YAML.

- Serialization-level semantic differences
  - YAML supports multi-line strings, while JSON does not. A multi-line string in YAML can be represented using modifier symbols on keys (such as the | and > characters), but these are serialization-level specifics that are not part of the content model, and hence JSON would have no representation for them.

    EXAMPLE 2: Given YAML content that uses one of these multi-line structures, a round-trip between JSON and YAML would lose the multi-line structure and end up with a single-line string in the YAML serialization.

  - YAML supports anchors and references, while JSON does not. In YAML, it is possible to define an anchor for a value and then reference that anchor in another part of the YAML document in order to reuse the value. JSON does not have a direct equivalent for this feature.

    EXAMPLE 3: Given a YAML document that uses anchors and references, a round-trip between JSON and YAML would lose the anchor and reference structure and end up with a repeated value in the YAML serialization.

4.5.6.3. Comparing XML Schema (XSD) and JSON Schema

XML Schema (XSD) is also not directly mappable to JSON Schema as XSD provides features that JSON Schema does not include, such as:

- Content-level semantic differences
  - Data type mismatches. JSON Schema supports the data types of string, number, object, array, boolean, and null, while XSD supports these data types and many additional data types such as date, time, and duration.

- Namespace support. XSD supports the concept of namespaces, while JSON Schema does not.
- Composing complex types. XSD offers the ability to define complex composite types and allows those to be reused in multiple places in the schema. JSON Schema does not have a direct equivalent for this feature.
- Value constraints. XSD supports placing constraints on string values using regular expressions, but JSON Schema does not have a direct equivalent for this feature.
- Serialization order and structural constraints. XSD allows for the definition of element sequences, groups, choices, and unions. JSON Schema does not have a direct equivalent for these features.

- Serialization-level semantic differences
  - XML attributes. XSD allows for the definition of attributes on elements, while JSON Schema does not have a direct equivalent for this feature.
  - XML comments. XML allows comments in the schema, while JSON does not.
  - XML processing instructions. XML allows processing instructions in the schema, while JSON does not.
  - XML entities. XML allows entities in the schema, while JSON does not. This means that an XML file with entities will likely be represented as a string in JSON, causing a round-trip operation from XML to lose the entity structure.
  - XML CDATA sections. XML allows CDATA sections in the schema, while JSON only has one type of string. This means that an XML file with CDATA sections will likely be represented as a string in JSON, causing a round-trip operation from XML to lose the CDATA structure.

4.5.6.4. Approach

The solution to this problem is to define a common approach that works across different serialization formats, and then define additional rules for each format to supplement the common approach.

4.6. Goals

LutaML is a system that aims to simplify users to easily define data models and work with information that conforms to those data models.

The goal of LutaML is to allow these steps to flow seamlessly:

1. Define a platform-independent data model in a human-readable format. (LutaML language)
2. Allow the data model to be directly useable in a programming language. (LutaML runtime)

3. Allow the data model to be expressed in a serialization schema language. (LutaML language)
4. Allow the data model to be directly serializable to a serialization format. (LutaML runtime)
5. Support the loading of data from a serialization format into a programming language. (LutaML runtime)
6. Allow importing a serialization schema language to define a data model. (LutaML runtime)
7. Allow the conversion of serialization formats to another through the data model. (LutaML runtime)

The LutaML language provides a way to represent structured data in a human-readable format.

5. Principles

The principles of LutaML are:

1. Provide a human-readable syntax for defining data models.
2. Consistent language constructs.
3. The language should be simple and easy to understand, and does not require specialized knowledge for particular serialization formats unless advance features are needed.
4. Support modular reuse of definitions across files.
5. Support instance representation of data models.

6. Core syntax

The core syntax of LutaML is based on the following constructs:

- primitive values
- declaration
- argument
- block

- comment

6.1. Primitive values

Primitive values are the basic building blocks of LutaML.

All primitive types have names that start with uppercase.

LutaML supports the following primitive values:

- `String` (e.g., `"high-fire"`),
- `Integer` (e.g., `42`).
- `Float` (e.g., `3.5`),
- `Boolean` (e.g., `true`),
- `TimeWithoutDate` (e.g., `12:00:00`),
- `DateTime` (e.g., `2024-01-01T12:00:00+00:00`),
- `Time` (e.g., `2024-01-01T12:00:00+00:00`),
- `Decimal` (e.g., `3.14159`),
- `Hash` (e.g., `{ "key": "value" }`)

6.2. Declaration, arguments and blocks

A declaration is a keyword that represents an action or a statement.

Syntax:

FIGURE 1

```
{declaration} {*argument} {*block}
```

Where,

`{declaration}`	the declaration keyword
`{*argument}`	zero or more arguments
`{*block}`	an optional blocks

An argument is a string, which can be a primitive value or a reference to another object.

A block is delimited by curly braces and provides a context for inner declarations.

EXAMPLE 1: This example shows the `class` declaration that takes an argument `Studio` to represent the name of the class.

```
class Studio
// or
class Studio {
}
```

EXAMPLE 2: This example shows the `class` declaration that takes an argument `Studio` as its name with a block given, where it contains an attribute.

```
class Studio {
  attribute ...
}
```

6.3. Comment

6.3.1. General

A comment is a line of text that is ignored by the processor.

6.3.2. Single-line comment

A single-line comment is also called a “tail comment”, a comment that appears at the end of a line.

The `//` character sequence can appear at the beginning of a line or after whitespace.

Syntax:

FIGURE 2

```
// {text}
or
{expression} // {text}
```

Where,

{text} the comment text

EXAMPLE 1

```
// This is a comment
```

EXAMPLE 2

```
class Studio { // This is a comment
  attribute location, String
}
```

EXAMPLE 3

```
// TODO: attributes <1>
class Ceramic {
  // Write docs <2>
  + glazeType: GlazeType // Glaze type <3>
```

```
}
```

Key

<1>	A line beginning with `//` is a comment.
<2>	A line beginning with whitespaces followed by `//` is a comment.
<3>	A line with `//` contains a tail comment: the text after `//` is a comment.

6.3.3. Multi-line comment

Comments can also be multi-line.

Syntax:

FIGURE 3

```
/* {text} */
```

Where,

{text}	the comment text

EXAMPLE

```
/* This is a
multi-line comment */
/*
  This is also a
  multi-line comment
*/
```

6.3.4. Definition

The `definition` block is used to define a multiline description for a class.

Syntax:

FIGURE 4

```
definition {text}
```

Where,

{text}	the description text
{block}	description text block

EXAMPLE 1

```
definition "Non-reflective finish"
```

Content within the block is considered to be multi-line textual content. No declarations are available within the block.

FIGURE 5

```
definition {block}
```

Where,

{text}	the description text
{block}	description text block

EXAMPLE 2

```
definition {
  Non-reflective finish.

  A matte finish is a non-reflective finish that is often used in
ceramics.
}
```

EXAMPLE 3 — Declaring a class named “Ceramic” with a description

```
class Ceramic {
  definition {
    This class represents a ceramic object.
  }
}
```

7. Representable objects

7.1. General

LutaML supports the following constructs.

7.2. Package

The package construct is used to define a collection of classes and enums.

Syntax:

FIGURE 6

```
package {name} {block}
```

Where,

{name}	the name of the package
{*block}	an optional block

EXAMPLE 1

```
package Ceramics {
  enum FiringProfile { values { "low", "medium", "high" } }

  class CeramicTile {
    attribute dimensions, float { cardinality 2 }  // [length, width]
    attribute firing_profile, FiringProfile
  }
}
```

The package construct can be nested. You can also define other packages within a package, which helps in organizing related classes and enums.

EXAMPLE 2

```
class Dimension {
  attribute length, float
  attribute width, float
  attribute height, float
}

package Ceramics {
  class Ceramic {
    attribute volume, float
    attribute color, String

    // Refers to outermost "root package"
    attribute dimensions, Dimension

    // Refers to "Material" package
    attribute tiles, Material::CeramicTile { cardinality 0..n }
  }

  package Materials {
    enum FiringProfile { values { "low", "medium", "high" } }

    class CeramicTile {
      // Refers to outermost "root package"
      attribute dimensions, Dimension

      // Refers to local "Materials" package
      attribute firing_profile, FiringProfile
```

```
    }
  }
}
```

Within the block, the following declarations are available:

- `class`
- `enum`
- `definition`
- `package`

7.3. Class

The `class` construct is used to define a class which is a collection of attributes.

Syntax:

FIGURE 7

```
class {name} {block}
```

Where,

`{name}`	the name of the class
`{*block}`	an optional block

EXAMPLE 1

```
class Studio {
  attribute location, String {
    definition "Location of the studio"
  }
  attribute potter, String {
    definition "Name of the potter"
  }
  attribute kiln, String {
    definition "Type of kiln used"
  }
}
```

EXAMPLE 2

```
class TemperatureWithUnit {
  attribute value, Float {
    definition "Temperature value"
  }
  attribute unit, String {
    definition "Unit of temperature"
  }
```

```
}
```

Class attributes can be restricted to a specific set of values using the `values` declaration.

EXAMPLE 3

```
class GlazeTechnique {
  attribute name, String {
    values { "Celadon", "Raku", "Majolica" }
  }
}
```

Or

```
enum GlazeTechniqueEnum {
  value "Celadon"
  value "Raku"
  value "Majolica"
}
class GlazeTechnique {
  attribute name, String {
    values GlazeTechniqueEnum
  }
}
```

It is possible to also define values with model instances. See [instance_representation] for syntax to declare model instances.

EXAMPLE 4: Given this class:

```
class Ceramic {
  attribute type, String {
    definition "Type of ceramic material"
  }
  attribute firing_temperature, Integer {
    definition "Temperature at which the ceramic is fired"
  }
}
```

The values can be defined as:

```
class CeramicCollection {
  attribute featured_piece, Ceramic {
    definition "Featured ceramic piece"

    values {
      instance Ceramic {
        type = "Porcelain"
        firing_temperature = 1300
        definition {
          Porcelain is a ceramic material made from kaolin clay.
        }
      }
      instance Ceramic {
        type = "Stoneware"
        firing_temperature = 1200
```

```
        definition {
          Stoneware is a durable ceramic that is fired at high
temperatures.
        }
      }
      instance Ceramic {
        type = "Earthenware"
        firing_temperature = 1000
        definition {
          Earthenware is a porous ceramic that is fired at lower
temperatures.
        }
      }
    }
  }
}
```

Or

```
class CeramicCollection {
  attribute featured_piece, Ceramic {
    definition "Featured ceramic piece"

    value {
      instance Ceramic {
        type = "Porcelain"
        firing_temperature = 1300
        definition {
          Porcelain is a ceramic material made from kaolin clay.
        }
      }
    }
    value {
      instance Ceramic {
        type = "Stoneware"
        firing_temperature = 1200

        definition {
          Stoneware is a durable ceramic that is fired at high
temperatures.
        }
      }
    }
    value {
      instance Ceramic {
        type = "Earthenware"
        firing_temperature = 1000
        definition {
          Earthenware is a porous ceramic that is fired at lower
temperatures.
        }
      }
    }
  }
}
```

Within the block, the following declarations are available:

- `attribute`
- `definition`
- `values` / `value`

7.4. Attribute

The `attribute` construct is used to define an attribute of a class.

Syntax:

FIGURE 8

```
attribute {name}, {type} {block}
```

Where,

{name}	the name of the attribute
{type}	the type of the attribute
{*block}	an optional block

EXAMPLE 1

```
attribute location, String {
  definition "Location of the studio"
}
```

EXAMPLE 2

```
attribute dimensions, Float {
  definition "Dimensions of the ceramic piece"
}
```

An attribute can have a cardinality constraint when it is a collection.

Syntax:

FIGURE 9

```
attribute {name}, {type} { cardinality {min}..{max} }
```

Where,

{min}	the minimum number of values
{max}	the maximum number of values

EXAMPLE 3

```
attribute batch_ids, String { cardinality 0..n }
```

An attribute can have a values constraint.

Syntax:

FIGURE 10

```
attribute {name}, {type} { values {value1, value2, ...} }
```

Where,

{value1, value2, …}	the values that the attribute can take, or an Enum.

EXAMPLE 4

```
attribute firing_profile, FiringProfile {
  values { "low", "medium", "high" }
}
```

EXAMPLE 5

```
enum FiringProfileEnum {
  value "low"
  value "medium"
  value "high"
}
class FiringProfile {
  attribute firing_profile, FiringProfile {
    values FiringProfileEnum
  }
}
```

An attribute that accepts a string value accepts value validation using regular expressions.

Syntax:

FIGURE 11

```
attribute {name}, String { pattern {regex} }
```

Where,

{regex}	the regular expression to validate the string value

EXAMPLE 6: In this example, the `color` attribute takes hex color values such as `#ccddee`.
A regular expression can be used to validate values assigned to the attribute. In this case, it is `/^#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})$/`.

```
attribute color, String {
  pattern /\A#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})\z/
}
```

An attribute can have a default value using the `default` option. The `default` option can be set to a value or a lambda that returns a value.

Syntax:

FIGURE 12

```
attribute {name}, {type} { default: {value} }
```

EXAMPLE 7 — Using the `default` option to set a default value for an attribute

```
class Glaze {
  attribute color, String { default: "Clear" }
  attribute temperature, Integer { default: 1050 }
}
```

Within the block, the following declarations are available:

- `definition`
- `values`
- `cardinality`

7.5. Value

The `value` construct is used to define a value.

Syntax:

FIGURE 13

```
value {name} {block}
```

Where,

`{name}` the name of the value

{*block}	an optional block

EXAMPLE

```
value "Porcelain" {
  definition "Ceramic material made from kaolin clay"
}
```

7.6. Enums

An enum construct is used to defined a named collection of values. Objects inside an Enum may be primitive values or instances of objects.

Syntax:

FIGURE 14

```
enum {name} { // enum block
  value {value-name} {value-block}
}
```

Where,

{name}	the name of the enum
{*block}	an optional block
{value-name}	the name of the value
{*value-block}	an optional block

EXAMPLE 1

```
enum GlazeTechnique {
  definition "Techniques for glazing ceramics"

  value "Celadon" {
    definition "Technique that creates glaze in a pale green color"
  }
  value "Raku" {
    definition "Technique that creates a crackled glaze"
  }
  value "Majolica" {
    definition "Technique that creates a white glaze"
  }
}
```

EXAMPLE 2

```
enum GlazeType {
  value "matte" { description "Non-reflective finish" }
  value "gloss" { description "Reflective finish" }
```

```
}
```

Enums accept instances as values as well.

EXAMPLE 3

```
class GlazeType {
  attribute name, String
  attribute description, String
}

enum GlazeTypes {
  definition "Types of glaze finishes"

  value "matte" {
    instance {
      name = "matte"
      definition = "Non-reflective finish"
    }
  }
  value "gloss" {
    instance {
      name = "gloss"
      definition = "Reflective finish"
    }
  }
}
```

Within the block, the following declarations are available:

- `value`
- `definition`

8. Serialization

8.1. General

LutaML supports the following constructs for serialization.

8.2. XML

8.2.1. General

LutaML supports the serialization of data models to XML.

Syntax:

FIGURE 15

```
xml {block}
```

Where,

{block}	an XML block

EXAMPLE

```
class Studio {
  attribute location, String
  attribute potter, String
  attribute kiln, String

  xml {
    map_element "location" { attribute location }
    map_element "potter" { attribute potter }
    map_attribute "kiln" { attribute kiln }
  }
}
```

An instance of the class `Studio` will be serialized to XML as:

```
<studio>
  <location>Paris</location>
  <potter>Marie</potter>
  <kiln>Electric</kiln>
</studio>
```

Within the block, the following declarations are available:

- `map_element`
- `map_attribute`
- `root`
- `no_root`

8.2.2. Setting root element name

By default, the root name is set to a lowercased version of the class name.

FIGURE 16

```
xml {
  root "xml_element_name" <1>
```

```
    }
```

Key

<1> The name of the root element.

EXAMPLE

```
class Studio {
  attribute location, String
  attribute potter, String
  attribute kiln, String

  xml {
    root "local-studios"
    map_element "location" { attribute location }
    map_element "potter" { attribute potter }
    map_attribute "kiln" { attribute kiln }
  }
}
```

An instance of the class `Studio` will be serialized to XML as:

```
<local-studios>
  <location>Paris</location>
  <potter>Marie</potter>
  <kiln>Electric</kiln>
</local-studios>
```

8.2.3. Omitting root element

By default, there exists a root element in a model represented in XML. In cases where the root element is not to be present, it can be omitted.

Syntax:

FIGURE 17

```
    xml {
      no_root
    }
```

EXAMPLE

```
class Studio {
  attribute location, String
  attribute potter, String
  attribute kiln, String

  xml {
    no_root
    map_element "location" { attribute location }
    map_element "potter" { attribute potter }
```

```
    map_attribute "kiln" { attribute kiln }
  }
}
```

An instance of the class `Studio` will be serialized to XML as:

```
<location>Paris</location>
<potter>Marie</potter>
<kiln>Electric</kiln>
```

8.2.4. map_element

The `map_element` construct is used to map an XML element to a model attribute.

Syntax:

FIGURE 18

```
map_element {name} {block}
```

Where,

{name}	the name of the element
{*block}	an optional block

EXAMPLE 1

```
class Studio {
  attribute location, String

  xml {
    map_element "geo-location" { attribute location }
  }
}
```

```
<studio>
  <geo-location>Paris</geo-location>
</studio>
```

In `map_element`, if the target attribute is a LutaML model, the newly specified element name overrides the defined `root` name of the model.

EXAMPLE 2

```
class LocationElement {
  attribute location, String

  xml do
    root "location"
    map_element "place" { attribute location }
  end
}

class Studio {
```

```
  attribute location, LocationElement

  xml {
    root "studio"
    map_element "geo" { attribute location }
  }
}
```

```
<studio>
  <geo><place>Paris</place></geo>
</studio>
```

8.2.5. map_attribute

The `map_attribute` construct is used to map an XML attribute to a model attribute.

Syntax:

FIGURE 19

```
map_attribute {name} {block}
```

Where,

{name}	the name of the attribute
{*block}	an optional block

EXAMPLE

```
class Studio {
  attribute location, String
  attribute identifier, String
  attribute kiln, String

  xml {
    map_element "geo-location" { attribute location }
    map_attribute "identifier" { attribute identifier }
    map_attribute "kiln" { attribute kiln }
  }
}
```

```
<studio identifier="1234" klin="Electric">
  <geo-location>Paris</geo-location>
</studio>
```

8.2.6. map_content

The `map_content` construct is used to map the content of an XML element to a model attribute.

Syntax:

FIGURE 20

```
map_content {block}
```

Where,

{*block}	an optional block

EXAMPLE

```
class Studio {
  attribute location, String

  xml {
    map_content { attribute location }
  }
}
```

```
<studio>Paris</studio>
```

8.2.7. Namespaces

Namespaces can be defined in the XML serialization.

The `namespace` construct is used to define a namespace in the element.

- In XML, this namespace declares the “default namespace” for all unnamespaced elements within this element.
- Attributes are not affected by the namespace.
- A namespace URI is optional, allowing for flexibility in definition.

Syntax:

FIGURE 21

```
namespace {prefix} {uri}
```

Where,

{prefix}	the namespace prefix
{uri}	the namespace URI (optional)

EXAMPLE

```
class Studio {
  attribute location, String
```

```
  xml {
    namespace "geo" "http://example.com/geo"
    map_element "location" { attribute location }
  }
}

<studio xmlns:geo="http://example.com/geo">
  <location>Paris</location>
</studio>
```

8.3. Key-value formats

Key-value formats are serialization formats that represent data as a collection of key-value pairs.

These include YAML, JSON, TOML and others.

8.3.1. map

The `map` construct is used to map a collection of elements to a model attribute.

Syntax:

FIGURE 22

```
    map {block}
```

Where,

`{*block}`	an optional block

EXAMPLE

```
class Studio {
  attribute location, String

  yaml {
    map "location" { attribute location }
  }
}

- location: Paris
- location: Berlin
```

```
- location: London
```

8.4. Collection mappings

8.4.1. map_key

> *WARNING*
>
> *Only for key-value formats.*

The map_key construct is used to map the key of a serialization object to a model attribute.

Syntax:

FIGURE 23

```
map_key {name} {block}
```

Where,

{name}	the name of the key
{*block}	an optional block

EXAMPLE

```
class Studio {
  attribute location, String
}

class StudioCollection < Collection {
  instances Studio

  yaml {
    map_key "location" { attribute location }
  }
}
```

```
- 01_paris: Paris
- 02_berlin: Berlin
- 03_london: London
```

8.4.2. map_value

> *WARNING*
>
> *Only for key-value formats.*

The `map_value` construct is used to map the value of a serialization object to a model attribute.

Syntax:

FIGURE 24

```
map_value {name} {block}
```

Where,

{name}	the name of the value
{*block}	an optional block

EXAMPLE

```
class Studio {
  attribute location, String
}

class StudioCollection < Collection {
  instances Studio

  yaml {
    map_key "location" { attribute location }
    map_value "location" { attribute location }
  }
}
```

```
- 01_paris:
    location: Paris
- 02_berlin:
    location: Berlin
- 03_london:
    location: London
```

9. Instance representation

9.1. Instance

An `instance` construct is used to define an instance of a class.

To assign values to attributes, the attribute name is followed by an equal sign and the value.

Syntax:

FIGURE 25

```
instance {name} {block}
```

Where,

{name}	the name of the instance
{*block}	an optional block

EXAMPLE 1

```
class Studio {
  attribute location, String
  attribute potter, String
  attribute kiln, String
}

instance "Studio" {
  location = "Paris"
  potter = "Marie"
  kiln = "Electric"
  definition {
    Marie's studio in Paris.
  }
}
```

EXAMPLE 2

```
class CeramicTile {
  attribute dimensions, Float { cardinality 2 }
  attribute firing_profile, FiringProfile
  definition {
    Types of ceramic tiles.
  }
}

instance "square" CeramicTile {
  dimensions = [30.5, 30.5]
  firing_profile = "high"

  definition {
    A square ceramic tile used for various applications.
  }
}
```

Within the block, the following declarations are available:

- attributes of the class
- definition
- values

9.2. Instance collection

Instances are representations of classes with values assigned to their attributes as a collection.

Syntax:

FIGURE 26

```
instances {name} {block}
```

Where,

{name}	the name of the instance
{*block}	an optional block

EXAMPLE 1

```
instances "Studios" {
  instance Studio {
    location = "Paris"
    potter = "Marie"
    kiln = "Electric"
  }

  instance Studio {
    location = "Berlin"
    potter = "Hans"
    kiln = "Gas"
  }
}
```

EXAMPLE 2

```
instances "Tiles" {
  instance "square" CeramicTile {
    dimensions = [30.5, 30.5]
    firing_profile = "high"
  }

  instance "circle" CeramicTile {
    dimensions = [20.5, 20.5]
    firing_profile = "medium"
  }
}
```

Within the block, the following declarations are available:

- `instance`

9.3. Value type

9.3.1. References

References are used to link instances together.

Syntax:

FIGURE 27

```
ref:(path)
```

Where,

{path}	the path to the instance

EXAMPLE

```
instances Glazes {
  GlazeFormula "blue_matte" {
    components = ["silica", "cobalt"]
  }

  CeramicTile "tile_002" {
    glaze = ref:(Glazes.blue_matte)
  }
}
```

10. Reuse and referencing

Lutaml supports modularity through external file inclusions.

10.1. External file inclusion

Use `require` to import definitions from other files.

FIGURE 28

```
require "materials.lml"
instances Tiles {
  Materials::CeramicTile "tile_001" {
    dimensions = [30.0, 30.0]
    firing_profile = "medium"
  }
```

```
  Materials::CeramicTile "tile_002" {
    dimensions = [40.0, 60.0]
    firing_profile = "high"
  }
}
```

10.2. Internal references

Link instances using `ref:`.

FIGURE 29

```
instances Production {
  GlazeFormula "gloss_blue" { ... }

  CeramicTile "tile_001" {
    glaze = ref:(GlazeFormula.gloss_blue)
  }
}
```

10.3. Namespacing

Class names are scoped to their model (e.g., `Materials::CeramicTile`).

11. Validation rules

11.1. Attribute constraints

1. Mandatory fields (`cardinality  1`) must be populated.
2. Values must match the declared type (e.g., `float` for `thickness`).

11.2. Class constraints

1. Unique names within a model.

2. Attribute names unique within a class.

11.3. Reference validity

- `ref`: must resolve to a valid instance path.
- Circular references are invalid.

12. Examples

12.1. Ceramic product definition

FIGURE 30

```
models Ceramics {
  enum FiringProfile { values { "low", "medium", "high" } }

  class CeramicTile {
    attribute dimensions, float { cardinality 2 }  //
[length, width]
    attribute firing_profile, FiringProfile
  }
}

instances Tiles {
  Ceramics::CeramicTile "tile_001" {
    dimensions = [30.5, 30.5]
    firing_profile = "high"
  }
}
```

12.2. Glaze formula reference

FIGURE 31

```
instances Glazes {
  GlazeFormula "blue_matte" {
    components = ["silica", "cobalt"]
```

```
    }

    CeramicTile "tile_002" {
      glaze = ref:(Glazes.blue_matte)
    }
  }
```