LutaML — Model Collections

RS 3005:2025, Version 1.0

Ronald Tse

Unrestricted
February 20, 2025
RS 3005:2025, Version 1.0
LutaML

Ribose

Contents

1. Scope 5

2. Normative references 5

3. Terms and definitions 6

4. Collection types 6

4.1. General 6

4.2. Configuration 6

4.3. Root collections 7

4.4. Named collections 9

4.5. Nested collections 11

4.6. Keyed collections (serialization formats only) 12

4.7. Keyed value collections 15

5. Collection serialization 17

5.1. General 17

5.2. XML serialization 17

5.3. YAML serialization 18

5.4. JSON serialization 18

6. Collection mapping rules 19

6.1. General 19

6.2. Root element override 19

7. Collection behaviors 20
7.1. Enumerable interface 20
7.2. Collection validation 21
7.3. Collection initialization 21

8. Advanced collection features 22
8.1. Ordered collections 22

9. Operations 22
9.1. Collection-level operations 22
9.2. Enumerable methods 23

Annex A **(normative)** Tutorial: Building a car parts database 26
A.1. Step 1: Basic collection 26
A.2. Step 2: Adding serialization 27
A.3. Step 3: Collection operations 28
A.4. Step 4: Nested collections 29
A.5. Step 5: Keyed collections 30
A.6. Summary 32

Ribose Inc.

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

copyright@ribose.com
www.ribose.com

1. Scope

This document specifies the collection capabilities in LutaML Model, which enable:

- Definition of collections within models
- Serialization of collections to different formats
- Mapping of collection elements to different representations

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. collection PREFERRED

specialized class that defines a container for multiple instances of a model type

3.2. collection element PREFERRED

individual instance within a collection

3.3. collection mapping PREFERRED

rules that define how collection elements are serialized and deserialized

4. Collection types

4.1. General

LutaML Model provides the `Lutaml::Model::Collection` class for defining collections of model instances.

4.2. Configuration

4.2.1. `instances {attribute}, {ModelType}`

Defined at the class level.

Defines the collection attribute and the model type of the collection elements.

4.2.2. `map_instances to: {attribute}`

Defined within the `key_value` block. In the `xml` block, the `map_attribute` and `map_element` directives are used instead.

This directive maps individual array elements to the defined `instances` attribute. These are the items considered part of the Collection and reflected as Enumerable elements.

NOTE The `map_instances` directive is only used in the `key_value` block.

4.3. Root collections

These are simple collections store multiple instances of the same model type, placed at the root level of the serialization format.

FIGURE 1: Simple collection in XML with models each containing an attribute `name`

```
<name>Item One</name>
<name>Item Two</name>
<name>Item Three</name>
```

FIGURE 2: Simple collection in YAML with models each containing an attribute `name`

```
---
- name: Item One
- name: Item Two
- name: Item Three
```

Syntax:

FIGURE 3

```
class MyCollection < Lutaml::Model::Collection
  instances :items, ModelType
end

class ModelType < Lutaml::Model::Serializable
  attribute :name, :string
```

```
    end
```

EXAMPLE

```
class Title < Lutaml::Model::Serializable
  attribute :content, :string
end

class TitleCollection < Lutaml::Model::Collection
  instances :titles, Title

  xml do
    no_root # default
    map_element "title", to: :titles
  end

  key_value do
    no_root # default
    map_instances to: :titles
  end
end
```

```
<title>
  <content>Title One</content>
</title>
<title>
  <content>Title Two</content>
</title>
<title>
  <content>Title Three</content>
</title>
```

```
---
- content: Title One
- content: Title Two
- content: Title Three
```

```
[
  {"content": "Title One"},
  {"content": "Title Two"},
  {"content": "Title Three"}
]
```

```
titles = TitleCollection.from_yaml(yaml_data)
titles.count
# => 3
titles.first.content
```

```
# => "Title One"
```

4.4. Named collections

Named collections are collections wrapped inside a name or a key.

FIGURE 4: Named collections in XML with models each containing an attribute `name`

```xml
<names>
  <name>Item One</name>
  <name>Item Two</name>
  <name>Item Three</name>
</names>
```

FIGURE 5: Named collections in YAML with models each containing an attribute `name`

```yaml
---
names:
- name: Item One
- name: Item Two
- name: Item Three
```

Syntax:

FIGURE 6

```
class MyCollection < Lutaml::Model::Collection
  instances :items, ModelType

  xml do
    root "name-of-xml-container-element"
  end

  key_value do
    root "name-of-key-value-container-element"
  end
end

class ModelType < Lutaml::Model::Serializable
```

```
      attribute :name, :string
    end
```

A direct element collection can be alternatively modeled as a “Model class with an attribute” that contains the collection of instances. In this case, the attribute will be an Array object, which does not contain additional attributes and methods.

EXAMPLE 1

```
class DirectTitleCollection < Lutaml::Model::Collection
  instances :items, Title

  xml do
    root "titles"
    map_instances to: :items
  end
end

class Title < Lutaml::Model::Serializable
  attribute :title, :string
end
```

```
<titles>
  <title>Title One</title>
  <title>Title Two</title>
  <title>Title Three</title>
</titles>
```

```
---
titles:
- title: Title One
- title: Title Two
- title: Title Three
```

```
{
  "titles": [
    {"title": "Title One"},
    {"title": "Title Two"},
    {"title": "Title Three"}
  ]
}
```

```
titles = DirectTitleCollection.from_yaml(yaml_data)
titles.count
# => 3
titles.first.title
# => "Title One"
titles.last.title
# => "Title Three"
```

EXAMPLE 2

```
class NameType < Lutaml::Model::Serializable
  attribute :name, :string
end

class NamedCollection < Lutaml::Model::Collection
```

```
  instances :names, NameType

  xml do
    root_name "names"
    map_element "item", to: :names
  end

  key_value do
    root_name "names"
    map_instances to: :names
  end
end
```

```
<names>
  <item>
    <name>Item One</name>
  </item>
  <item>
    <name>Item Two</name>
  </item>
  <item>
    <name>Item Three</name>
  </item>
</names>
```

```
---
names:
  - name: Item One
  - name: Item Two
  - name: Item Three
```

```
{
  "names": [
    {"name": "Item One"},
    {"name": "Item Two"},
    {"name": "Item Three"}
  ]
}
```

4.5. Nested collections

Collections can be nested within other models and define their own serialization rules.

EXAMPLE

```
class Title < Lutaml::Model::Serializable
  attribute :title, :string
end

class TitleCollection < Lutaml::Model::Collection
  instances :items, Title

  xml do
    root "title-group"
    map_element "artifact", to: :items
  end
end
```

```
class BibItem < Lutaml::Model::Serializable
  attribute :titles, TitleCollection

  xml do
    root "bibitem"
    # This overrides the collection's root "title-group"
    map_element "titles", to: :titles
  end
end
```

```
<bibitem>
  <titles>
    <title>Title One</title>
    <title>Title Two</title>
    <title>Title Three</title>
  </titles>
</bibitem>
```

4.6. Keyed collections (serialization formats only)

4.6.1. General

Keyed collections store instances with unique keys.

> *WARNING*
>
> *Keyed collections were previously handled through Collection with keyed elements (keyed collection). However, this new mechanism is much easier to understand.*

In key-value serialization formats, a key can be used to uniquely identify each instance. This usage allows for enforcing uniqueness in the collection.

NOTE The concept of keyed collections does not typically apply to XML collections.

There are two types of values in a keyed collection:

1. When the value is a “model instance”. This is called the “keyed model collection”. Refer to 4.6.3 for more information.
2. When the value is a “primitive type”. This is called the “keyed value collection”. Refer to 4.7 for more information.

The mechanism for defining keyed collections is slightly different for both types.

4.6.2. `map_key` and `map_value`

The `map_key` method specifies that the unique key is to be moved into an attribute belonging to the instance model.

Syntax:

FIGURE 7

```
key_value do
  map_key to_instance: {instance-attribute-name}
end
```

Where,

`to_instance` Refers to the attribute name in the instance that contains the key.

`{key_attribute}` The attribute name in the instance that contains the key.

The `map_value` method specifies that the value (the object referenced by the unique key) is to be moved into an attribute belonging to the instance model.

Syntax:

FIGURE 8

```
key_value do
  # basic pattern
  map_value {operation}: [*argument]

  # to_instance
  map_value to_instance: {instance-attribute-name}

  # as_instance
  map_value as_attribute: {instance-attribute-name}
end
```

Where,

`{operation}` The operation to be performed on the key-referenced value. Accepts the following values.

`to_instance` Each value includes multiple attributes. Map all those attributes into one attribute belonging to the instance model.

`as_attribute` Each value is of a primitive

type. Map that primitive type value into an attribute belonging to the instance model.

`{instance-attribute-name}` — The attribute name in the instance that will contain the value.

4.6.3. Keyed model collections

In keyed model collections, the collection contains multiple model instances. Within the collection, the unique key identifies individual models.

This can be thought as the case where the unique key is moved into the model instance as a model attribute.

EXAMPLE 1 — Sample of a keyed model collection

```
---
author_01:
  name: Author One
author_02:
  name: Author Two
author_03:
  name: Author Three
```

Here we only need the `map_key` method to define the key attribute in the instance.

Syntax:

FIGURE 9

```
class ModelType < Lutaml::Model::Serializable
  attribute :key_attribute, :string
  # ... additional attributes
end

class KeyedCollection < Lutaml::Model::Collection
  instances :items, ModelType

  key_value do
    map_key to_instance: :key_attribute <1>
    map_instances to: :items <2>
  end
end
```

Key

<1> — The `:key_attribute` is the attribute name inside the `ModelType` used to uniquely identify each instance.

<2> The `:items` attribute is the collection attribute containing the instances.

EXAMPLE 2

```
class Author < Lutaml::Model::Serializable
  attribute :id, :string
  attribute :name, :string
end

class AuthorCollection < Lutaml::Model::Collection
  instances :authors, Author

  key_value do
    map_key to_instance: :id # This refers to 'authors[].id'
    map_instances to: :authors
  end
end
```

```
---
author_01:
  name: Author One
author_02:
  name: Author Two
author_03:
  name: Author Three
```

```
{
  "author_01": {"name": "Author One"},
  "author_02": {"name": "Author Two"},
  "author_03": {"name": "Author Three"}
}
```

```
authors = AuthorCollection.from_yaml(yaml_data)
authors.first.id
# => "author_01"
authors.first.name
# => "Author One"
```

4.7. Keyed value collections

A keyed value collection is a collection of primitive values (not models) that are keyed.

EXAMPLE 1

```
---
author_01: true
author_02: false
author_03: true
```

Here we need to use both `map_key` and `map_value` methods to define the key attribute in the instance.

Syntax:

FIGURE 10

```
class ModelType < Lutaml::Model::Serializable
  attribute :key_attribute, :string
  # ... additional attributes
end

class KeyedCollection < Lutaml::Model::Collection
  instances :items, ModelType

  key_value do
    map_key to_instance: :key_attribute <1>
    map_value as_attribute: :value_attribute <2>
    map_instances to: :items <3>
  end
end
```

Key

<1>	The `:key_attribute` is the attribute name inside the `ModelType` used to uniquely identify each instance.
<2>	The `:value_attribute` is the attribute name inside the `ModelType` used to uniquely identify each instance.
<3>	The `:items` attribute is the collection attribute containing the instances.

EXAMPLE 2

```
class AuthorAvailability < Lutaml::Model::Serializable
  attribute :id, :string
  attribute :available, :boolean
end

class AuthorCollection < Lutaml::Model::Collection
  instances :authors, AuthorAvailability

  key_value do
    map_key to_instance: :id # This refers to 'authors[].id'
    map_value as_attribute: :available # This refers to 'authors[].
available'
    map_instances to: :authors
  end
end
```

```
---
author_01: true
author_02: false
author_03: true
```

```
{
```

```
  "author_01": true,
  "author_02": false,
  "author_03": true
}

authors = AuthorCollection.from_yaml(yaml_data)
authors.first.id
# => "author_01"
authors.first.available
# => true
```

5. Collection serialization

5.1. General

Collections support multiple serialization formats through format-specific mapping rules.

5.2. XML serialization

XML serialization defines how collection elements are represented in XML.

The `xml` block is used to define XML serialization rules for the collection.

In a collection, the following directives are available:

- `root` — Specifies the XML container element name
- `no_root` — Disables the root element for the collection
- `map_element` — Specifies how individual elements are represented
- `map_attribute` — Maps an attribute to the XML output
- `map_instances` — Maps the collection instances to the XML output

EXAMPLE 1: The `root` directive specifies the XML container element name.

```
xml do
  root "container-name"
end
```

EXAMPLE 2: The `map_element` directive specifies how individual elements are represented.

```
xml do
  map_element "element-name", to: :collection_attribute
end
```

EXAMPLE 3: The `map_attribute` directive maps an attribute to the XML output.

```
xml do
  map_attribute "attribute-name", to: :attribute_name
end
```

5.3. YAML serialization

YAML serialization defines how collection elements are represented in YAML.

EXAMPLE

```
class AuthorCollection < Lutaml::Model::Collection
  instances :items, Author

  yaml do
    map_instances :items
  end
end
```

Produces:

FIGURE 11

```
---
authors:
  - name: Author 1
    bio: Bio 1
  - name: Author 2
    bio: Bio 2
```

5.4. JSON serialization

JSON serialization defines how collection elements are represented in JSON.

EXAMPLE

```
class AuthorCollection < Lutaml::Model::Collection
  instances :items, Author

  json do
    root_key "authors"
    map_instances :items
  end
end
```

Produces:

```
{
  "authors": [
    {"name": "Author 1", "bio": "Bio 1"},
    {"name": "Author 2", "bio": "Bio 2"}
  ]
```

```
}
```

6. Collection mapping rules

6.1. General

Collection mapping rules determine how collection elements are serialized and deserialized.

6.2. Root element override

The root element name can be overridden at the collection usage point.

EXAMPLE 1

```
class Title < Lutaml::Model::Serializable
  attribute :title, :string
end

class TitleCollection < Lutaml::Model::Collection
  instances :items, Title

  xml do
    root "title-group"
    map_element "artifact", to: :items
  end
end

class BibItem < Lutaml::Model::Serializable
  attribute :titles, TitleCollection

  xml do
    root "bibitem"
    # This overrides the collection's root "title-group"
    map_element "titles", to: :titles
  end
end

<bibitem>
  <titles>
    <title>Title One</title>
    <title>Title Two</title>
    <title>Title Three</title>
  </titles>
</bibitem>
```

EXAMPLE 2

```
class Title < Lutaml::Model::Serializable
```

```
  attribute :title, :string
end

class TitleCollection < Lutaml::Model::Collection
  instances :items, Title

  xml do
    root "title-group"
    # This overrides the element's root "title"
    map_element "artifact", to: :items
  end
end

class BibItem < Lutaml::Model::Serializable
  attribute :titles, TitleCollection

  xml do
    root "bibitem"
    map_element "title-group", to: :titles
  end
end
```

```
<bibitem>
  <title-group>
    <artifact>Title One</artifact>
    <artifact>Title Two</artifact>
    <artifact>Title Three</artifact>
  </title-group>
</bibitem>
```

7. Collection behaviors

7.1. Enumerable interface

Collections implement the Ruby `Enumerable` interface, providing standard collection operations.

Collections allows the following sample `Enumerable` methods:

- `each` — Iterate over collection items
- `map` — Transform collection items
- `select` — Filter collection items
- `find` — Find items matching criteria
- `reduce` — Aggregate collection items

EXAMPLE

```
class AuthorCollection < Lutaml::Model::Collection
  instances :items, Author
end

authors = AuthorCollection.new

# Iterate
authors.each { |author| puts author.name }

# Transform
author_names = authors.map { |author| author.name }

# Filter
active_authors = authors.select { |author| author.active? }
```

7.2. Collection validation

Collections can define validation rules for their elements.

Syntax:

FIGURE 12

```
    class ValidatedCollection < Lutaml::Model::Collection
      instances :items, ModelType do
        validates :attribute, presence: true
        validate :custom_validation
      end
    end
```

EXAMPLE

```
class PublicationCollection < Lutaml::Model::Collection
  instances :items, Publication do
    validates :title, presence: true
    validates :year, numericality: { greater_than: 1900 }

    validate :must_have_author

    def must_have_author
      errors.add(:base, "Publication must have an author") unless
author.present?
    end
  end
end
```

7.3. Collection initialization

Collections can be initialized with an array of items or through individual item addition.

EXAMPLE 1

```
class AuthorCollection < Lutaml::Model::Collection
  instances :items, Author
end

authors = AuthorCollection.new([
  Author.new(name: "Author 1"),
  Author.new(name: "Author 2")
])
```

EXAMPLE 2

```
authors = AuthorCollection.new
authors << Author.new(name: "Author 1")
authors.push(Author.new(name: "Author 2"))
```

8. Advanced collection features

8.1. Ordered collections

Collections that maintain a specific ordering of elements.

EXAMPLE

```
class OrderedCollection < Lutaml::Model::Collection
  instances :items, ModelType
  ordered by: "date", order: :desc
end

class ModelType < Lutaml::Model::Serializable
  attribute :date, :datetime
end
```

9. Operations

9.1. Collection-level operations

Collections can be combined using set operations to create new collections.

Syntax:

FIGURE 13

```
    # Union
    collection1.union(collection2)

    # Intersection
    collection1.intersection(collection2)

    # Difference
    collection1.difference(collection2)
```

EXAMPLE

```
class AuthorCollection < Lutaml::Model::Collection
  instances :items, Author
end

# Usage
authors1 = AuthorCollection.new([
  Author.new(name: "Author 1"),
  Author.new(name: "Author 2")
])

authors2 = AuthorCollection.new([
  Author.new(name: "Author 2"),
  Author.new(name: "Author 3")
])

combined = authors1.union(authors2)
common = authors1.intersection(authors2)
unique = authors1.difference(authors2)
```

9.2. Enumerable methods

Collections inherit from Enumerable and support standard enumeration methods.

Collections can be filtered using predicate methods to create new collections.

Syntax:

FIGURE 14

```
    collection.filter(predicate)
    collection.reject(predicate)
    collection.select(predicate)
```

EXAMPLE 1

```
class PublicationCollection < Lutaml::Model::Collection
  instances :items, Publication

  def published
    self.class.new(
      items.select { |item| item.status == 'published' }
    )
  end

  def by_year(year)
    self.class.new(
      items.select { |item| item.year == year }
    )
  end

  def by_author(author_name)
    self.class.new(
      items.select { |item| item.author == author_name }
    )
  end
end

# Usage
publications = PublicationCollection.new(items)
published_2023 = publications.published.by_year(2023)
```

Collections can be transformed using mapping methods to create new collections.

Syntax:

FIGURE 15

```
    collection.count
    collection.sum(attribute)
    collection.average(attribute)
    collection.group_by(attribute)
```

EXAMPLE 2

```
class PublicationCollection < Lutaml::Model::Collection
  instances :items, Publication

  def total_citations
    items.sum(&:citation_count)
  end

  def average_rating
    items.sum(&:rating).to_f / items.count
  end

  def by_category
    items.group_by(&:category)
  end

  def statistics
```

```
    {
      total_items: count,
      total_citations: total_citations,
      average_rating: average_rating,
      by_status: items.group_by(&:status).transform_values(&:count)
    }
  end
end
```

Collections can be transformed while maintaining their collection nature.

Syntax:

FIGURE 16

```
collection.map(transform)
collection.flat_map(transform)
```

EXAMPLE 3

```
class CitationCollection < Lutaml::Model::Collection
  instances :items, Citation

  def to_references
    ReferenceCollection.new(
      items.map { |citation| citation.to_reference }
    )
  end

  def normalize
    items.each do |citation|
      citation.normalize!
    end
    self
  end
end
```

Annex A
(normative)

Tutorial: Building a car parts database

This tutorial demonstrates building a car parts database using LutaML collections, progressively adding features to show different collection capabilities.

A.1. Step 1: Basic collection

- Create a basic LutaML model
- Instantiate a simple collection
- Use basic enumeration methods

EXAMPLE: First, define the basic model:

```
class CarPart < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :description, :string
  attribute :price, :float
end
```

Then create a simple collection:

```
class PartsCollection < Lutaml::Model::Collection
  instances :parts, CarPart
end

# Create and use the collection
parts = PartsCollection.new([
  CarPart.new(name: "Engine Block", price: 1500.0),
  CarPart.new(name: "Brake Pad", price: 50.0)
])

# Basic enumeration
parts.each { |part| puts part.name }
parts.count  # => 2
```

- Collections inherit from Enumerable
- Collections maintain type safety through the `instances` declaration

- Basic enumeration methods are available out of the box

A.2. Step 2: Adding serialization

- Configure XML serialization
- Configure YAML serialization
- Understand format mapping differences

EXAMPLE: Enhance the collection with serialization rules:

```
class PartsCollection < Lutaml::Model::Collection
  instances :parts, CarPart

  xml do
    root "parts-catalog"
    map_element "part", to: :parts
  end

  yaml do
    sequence "parts"
  end
end

# Create some parts
parts = PartsCollection.new([
  CarPart.new(name: "Engine Block", price: 1500.0),
  CarPart.new(name: "Brake Pad", price: 50.0)
])

# XML output
parts.to_xml
```

Produces:

```
<parts-catalog>
  <part>
    <name>Engine Block</name>
    <price>1500.0</price>
  </part>
  <part>
    <name>Brake Pad</name>
    <price>50.0</price>
  </part>
</parts-catalog>
```

```
parts:
  - name: Engine Block
    price: 1500.0
  - name: Brake Pad
    price: 50.0
```

- Collections can support multiple serialization formats simultaneously

- Each format can have its own mapping rules
- Root elements and naming can be customized per format

A.3. Step 3: Collection operations

- Implement filtering methods
- Add aggregation calculations
- Chain collection operations

EXAMPLE: Add operation methods to the collection:

```
class PartsCollection < Lutaml::Model::Collection
  instances :parts, CarPart

  # ... existing serialization code ...

  def expensive_parts(threshold = 1000.0)
    self.class.new(
      parts.select { |part| part.price > threshold }
    )
  end

  def total_value
    parts.sum(&:price)
  end

  def price_stats
    {
      total: total_value,
      average: total_value / count,
      max: parts.map(&:price).max,
      min: parts.map(&:price).min
    }
  end
end

# Usage
parts = PartsCollection.new([
  CarPart.new(name: "Engine Block", price: 1500.0),
  CarPart.new(name: "Brake Pad", price: 50.0),
  CarPart.new(name: "Transmission", price: 1200.0)
])

expensive = parts.expensive_parts
puts parts.price_stats
```

- Collection operations return new collection instances
- Operations can be chained
- Statistical operations are easily implemented using Enumerable methods

A.4. Step 4: Nested collections

- Create hierarchical data structures
- Configure nested serialization
- Handle complex data relationships

EXAMPLE: First, add a category model:

```
class Category < Lutaml::Model::Serializable
  attribute :name, :string
  attribute :description, :string
end

class PartCategory < Lutaml::Model::Serializable
  attribute :category, Category
  attribute :parts, PartsCollection
end

class CategorizedPartsCollection < Lutaml::Model::Collection
  instances :categories, PartCategory

  xml do
    root "parts-catalog"
    map_element "category", to: :categories do
      map_element "name", to: "category.name"
      map_element "parts", to: :parts
    end
  end
end
```

Usage:

```
engine_parts = PartsCollection.new([
  CarPart.new(name: "Engine Block", price: 1500.0),
  CarPart.new(name: "Piston", price: 100.0)
])

brake_parts = PartsCollection.new([
  CarPart.new(name: "Brake Pad", price: 50.0),
  CarPart.new(name: "Rotor", price: 75.0)
])

catalog = CategorizedPartsCollection.new([
  PartCategory.new(
    category: Category.new(name: "Engine"),
    parts: engine_parts
  ),
  PartCategory.new(
    category: Category.new(name: "Brakes"),
    parts: brake_parts
  )
```

```
])
```

This produces:

```
<parts-catalog>
  <category>
    <name>Engine</name>
    <parts>
      <part>
        <name>Engine Block</name>
        <price>1500.0</price>
      </part>
      <part>
        <name>Piston</name>
        <price>100.0</price>
      </part>
    </parts>
  </category>
  <category>
    <name>Brakes</name>
    <parts>
      <part>
        <name>Brake Pad</name>
        <price>50.0</price>
      </part>
      <part>
        <name>Rotor</name>
        <price>75.0</price>
      </part>
    </parts>
  </category>
</parts-catalog>
```

- Collections can contain other collections
- Serialization rules cascade through the hierarchy
- Complex relationships can be modeled naturally

A.5. Step 5: Keyed collections

- Implement unique identifiers
- Convert to keyed collection
- Add efficient lookup methods

EXAMPLE: Enhance the CarPart model with an identifier:

```
class CarPart < Lutaml::Model::Serializable
  attribute :id, :string
  attribute :name, :string
  attribute :price, :float
end
```

```ruby
class KeyedPartsCollection < Lutaml::Model::Collection
  instances :parts, CarPart

  key_value do
    map_key to: :id
    map_instances to: :parts
  end

  xml do
    root "parts-catalog"
    map_element "part", to: :parts do
      map_attribute "id", to: :id
    end
  end

  def find_part(id)
    parts.find { |part| part.id == id }
  end

  def find_parts_by_ids(ids)
    self.class.new(
      parts.select { |part| ids.include?(part.id) }
    )
  end
end
```

Usage:

```ruby
parts = KeyedPartsCollection.new([
  CarPart.new(id: "ENG001", name: "Engine Block", price: 1500.0),
  CarPart.new(id: "BRK002", name: "Brake Pad", price: 50.0)
])

# YAML representation
```

Produces:

```yaml
ENG001:
  name: Engine Block
  price: 1500.0
BRK002:
  name: Brake Pad
  price: 50.0
```

XML representation:

```xml
<parts-catalog>
  <part id="ENG001">
    <name>Engine Block</name>
    <price>1500.0</price>
  </part>
  <part id="BRK002">
    <name>Brake Pad</name>
    <price>50.0</price>
  </part>
```

```
</parts-catalog>
```

Example lookups:

```
# Find a single part
engine = parts.find_part("ENG001")

# Find multiple parts
brake_engine = parts.find_parts_by_ids(["BRK002", "ENG001"])
```

- Keys provide efficient lookup capabilities
- Different serialization formats can represent keys differently
- Keyed collections maintain referential integrity

A.6. Summary

This tutorial demonstrated:

- Basic collection creation and usage
- Multiple serialization format support
- Collection operations and aggregation
- Nested collection hierarchies
- Keyed collection lookup patterns

The progression from simple to complex features shows how LutaML collections can be used to build sophisticated data management systems.