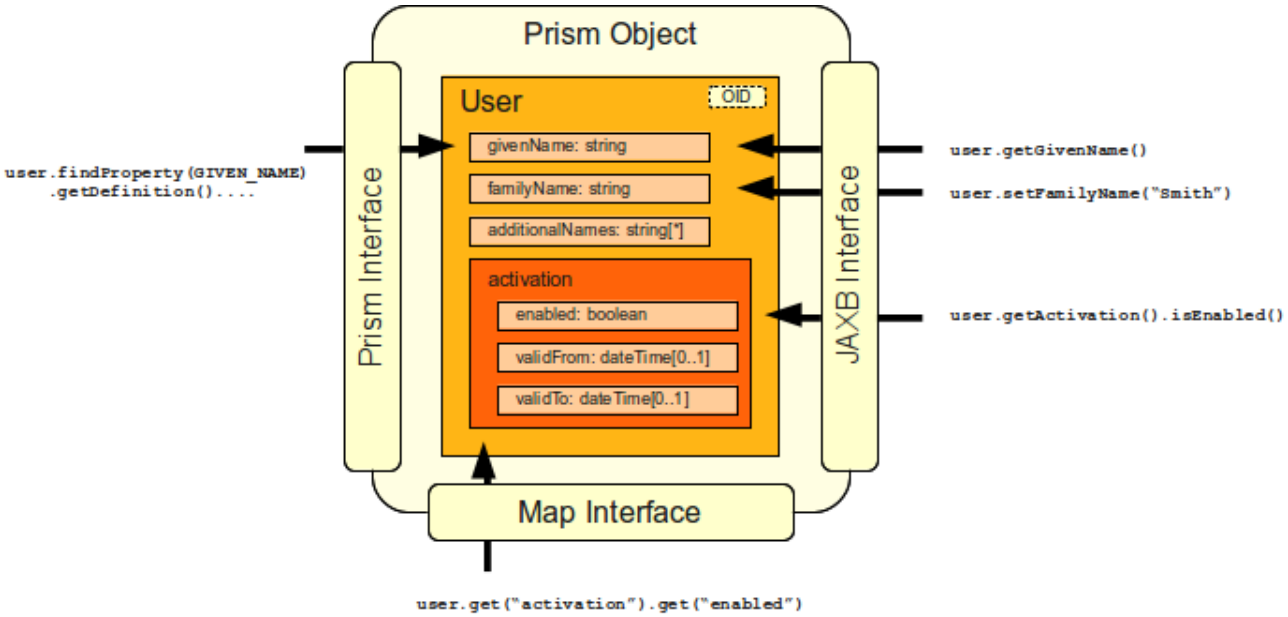
Prism Objects

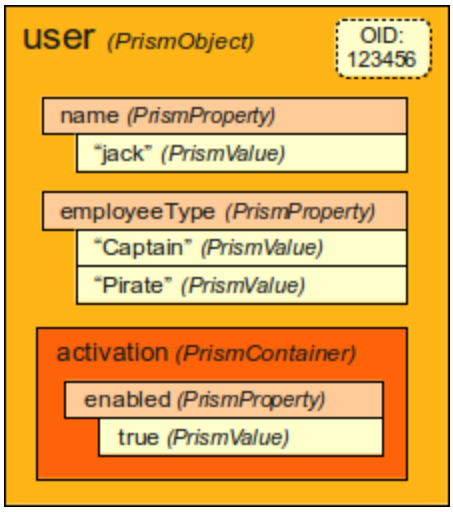
Introduction

Prism is a data representation mechanisms that allows to present data in several formats and the same time. Prism objects have support for full schema (definitions) for a reliable data conversion. It includes deltas and diff/patch mechanism to support relative change model. Prisms can be transformed to compile-time classes and therefore considerably speed up the development cycle.



The goal of prism is to represent the data in a format that is usable in a broad variety of situations. While the are represented in a structure of ordinary Java objects, they can be represented as DOM tree, Java Maps, serialized to and parsed from XML and JSON, etc.

Native prism representation (Java)



XML representation

```
<user oid="123456">
  <name>jack</name>
  <employeeType>Captain</employeeType>
  <employeeType>Pirate</employeeType>
  <activation>
    <enabled>true</enabled>
  <activation>
</user>
```

JSON representation

```
{
  "_oid": "123456",
  "name": "jack",
  "employeeType": [ "Captain", "Pirate" ],
  "activation": {
    "enabled": true,
  }
}
```

Live Data

Prism objects are more or less ordinary data structures in Java. They can be accessed using a native Prism interface (API), but the same data can also be accessed by variety of other interfaces (JAXB, DOM, ...). The unique feature of prism is that the data are not transformed to these formats but real live data are presented. For example if a bean property is set by invoking JAXB setter the change is immediately reflected to the underlying prism data model. The next call to DOM interface will return the new value of the property.

Live data example

```
// Set the "fullName" property using a JAXB interface
UserType userType = ....
userType.setFullName("Jack Sparrow");

// Invoking native Prism interface to read fullName, returns "Jack Sparrow"
PrismObject<UserType> userPrism = ....
String fullName = userPrism.findProperty(new QName(NS_C, "fullName").getValue().getValue();

// Invoking DOM interface to read the fullName, returns "Jack Sparrow"
Element fullnameElement = ....
String fullName = fullnameElement.getTextContent()
```

Prism Native Data Structures

Full article: Prism Data Structures

Each prism data structure is constructed from *items*. The items can be of three types:

- **Property** is an item that contains useful value. It usually contains primitive data types such as string, integer or date. But it may also contain complex data structures. Property is the "atomic" unit of information, it cannot be divided to smaller parts. Property values can be added, deleted or replaced, but only a complete value. Prism does not care for anything smaller than a property.
- **Reference** points to a prism object. It is used to represent relationship between prism objects.
- **Container** contains other items: properties, references or other containers. Containers build the structure of prism data.

Prism objects are special cases of prism containers. Although prism data can be represented without objects, objects give a kind of order to the data. Objects are supposed to be stand-alone meaningful sets of prism data. Each object has its unique identifier called simple *object identifier* or OID for short. Prism *references* are used to point to objects which is a method to create complex data structures.

Any prism item (except object) may have multiple values. Such values are unordered. Limiting the values to unordered makes it much easier to work with relative changes.

Definition and Schema

Full article: Prism Schema

Each prism item has a definition. The definition is defined by the schema or is determined at runtime. The definition tells the type of the item, e.g. whether it is string, integer or a date. This is necessary to present the same data in various formats. E.g. to be able to correctly process write of the data using a DOM interface the prism must know that it is date to correctly parse it from the DOM string representation.

The presence of a schema at runtime has yet another advantage: code can be generated from the schema. This is used by the prism JAXB code generator which generates data representation that provide a simple Java Bean interface. Having a data model generated and compiled is a huge advantage for most systems. E.g. compiler checks are making the development cycle much shorter and it makes refactoring much more reliable.

Presence of full object schema at runtime is a priceless help for processing and especially displaying the objects. As each item has a proper definition associated to it the GUI code can use it to render an object without even knowing how the object looks like. The code can iterate over items and use the schema to display appropriate form fields, e.g. simple text field for string items and date picker for date fields. The schema also allows annotations that make the presentation even more attractive such as specification of field display names, field ordering, etc. This feature is ideal for objects that can be extended in the runtime or parts of their schema is not just not known at compile time (see *Dynamic Schema* below).

Currently Prism Schema is defined in XML Schema Defenition (XSD) language with custom annotations (but that may change in the future).

Dynamic Schemas

On of the unique features of prism is support for runtime and dynamic schemas.

Runtime schema is a static schema definition that is not available at compile time. It may be available as assembly time or even at runtime. It usually takes form of XSD files that are loaded at system start. As such schema is not available as compile time it cannot be used to generate compiled code. But except for that the runtime schema behaves almost the same as compile time schema: it is used to parse and validate data when parsing prism items from XML, to serialize data, etc.

Dynamic schema is schema that is not fixed, not even in runtime. E.g. there may be a container that can store items of any type, even each instance of the container may have different items of different types. Prism supports such schema and still correctly maintains definitions for such items end-to-end. E.g. it parses the data from the source format and creates appropriate definition. The definition travels with the item though the system. If the data are serialized, the definition is serialized with the data (if possible), e.g. by using the XML `xsi:type` runtime type specification.

Dynamic schema is tricky. Complete schema may not be available at the time that the data are parsed. Therefore prism allows dynamic items to be in the *raw* parsing state. It means that prism only parses basic data structure but the data are left as they were in the source file. E.g. when parsing data from XML prism determines what is a property and what is container but the actual data types are not yet determined. All the dynamic data remain internally in prism objects in the form of un-parsed DOM elements until the proper schema is applied. When the schema is applied the parsing is completed and the data are converted from the raw elements to the proper native data types. This process is mostly transparent to the user but there may be some gotchas (e. g. comparing parsed and raw items).

Extensibility

Prism data structures are designed to be extensible while also maintain a reasonable degree of backward compatibility. The primary mechanism to allow this is namespacing. Name and type of every prism item is a QName (name in a namespace). data formats that nativelly support QNames (such as XML) can take natural advantage of that. For simpler data formats (such as JSON) we create a dialect for prism data. Namespaces provide isolation of custom extensions from various sources and also isolate the extensions form a future evolutionary changes in primary prism data model. Namespaces also provide data model versioning: change in the primary namespace indicated non-compatible change.

Each prism object has a pre-defined optional `extension` container. This container is supposed to hold custom extensions to each object. The `extension` container is expected to be dynamic by default therefore allowing any item to be stored there. However, there are mechanisms in prism to provide a (run-time) schema for this container. This constraints the possibilities of which items may be stored there and therefore creates a solid schema for the entire object.

Object with extension (XML representation)

```
<user oid="123456">
  <name>jack</name>
  <extension>
    <pir:shipName>Black Pearl</pir:shipName>
  </extension>
  <employeeType>Captain</employeeType>
  <employeeType>Pirate</employeeType>
</user>
```

Deltas

Prism data structures were designed with the relative change model in mind. Prism library contains integral support for *deltas*: objects that describe relative change of prism items and objects. Deltas can be applied (*patch*) to prisms to get the resulting state. Prisms can be compared (*diff*) to generate deltas that describe the difference. As prism items have proper definitions this process is much more reliable as compared to e.g. direct XML or JSON diffs.

Prism Interfaces

Prism object are accessible using several interfaces. The details are described in following sections. Yet there are few concepts that are common to all interfaces.

Prism object can be *switched* from one interface to the other. Methods to do that start with prefix `as`, e.g. `asObjectable()`. Such method return the same object represented by the mechanism of the other interface. The two objects still refer to the same data, therefore changing the content of one object is immediately reflected to the other.

Interface switching example

```
PrismObject<UserType> userPrism = ....;                                    // Native Prism API
UserType userType = userPrism.asObjectable();                              // Switch to JAXB generated class

userType.setFullName("Jack Sparrow");                                      // Sets the "fullName" property

String fullName =
  userPrism.findProperty(new QName(NS_C, "fullName").getValue().getValue();    // Returns "Jack Sparrow"
```

Prism context (`PrismContext`) is an umbrella class that maintains the instances needed for operation of the whole prism library. E.g. the prism context maintains registered schemas, initialized JAXB context (if compile-time classes are used) and so on. Prism context is present in all prism items (as `transient` member), therefore it usually do not need to be provided for most of the operations. However, some operations require prism context as an explicit parameter or are invoked directly on a prism context. This include mostly parsing and serialization operations and construction of a new prisms.

Native Prism API

Native Prism API is ideal for dynamic manipulation of the data structures. It is also the only API that exposes all the functions of Prism library.

TODO

Most prism classes implement the `clone()` method, therefore the prism data objects are easily clonnable.

Prism items, values and definitions are `Serializable`. However, there is a catch. Prism context is not serializable and it is a `transient` member of prism items and definitions. Therefore deserialized prism items and definitions may not be fully operational. The call to `revive(...)` method is required after deserialization to make the prisms work as expected.

JAXB Generated Beans

Prism JAXB interface maintains the same looks and feel as usual JAXB generated classes. But instead of plain annotated Java Beans the Prism JAXB compiler plug-in generates classes that are just proxies for real Prism classes. This implements the immediate propagation of changes: the data are always stored *only* in native prism data structures. Any JAXB classes are just relaying the calls to the native classes. Any `get` is getting the data from prisms, any `set` is applied directly to prisms.

TODO: schema-full and schema-less JAXB operation

DOM

The DOM Prism API has now only a very rough implementation in a form of proof of concept. It is not yet finished.

Parsing and Serialization

In addition to live data interfaces prism data can be parsed from and serialized to a variety of data formats.

Data Format	Parsing	Serialization
XML	Yes (DOM, StAX planned)	Yes (DOM)
JSON	Planned	Planned

Prism in midPoint

Prism is used as a universal data representation through the entire midPoint system. It is used from end to end, from user interface to the database. Every midPoint component works with prism objects. It means that midPoint is **not** using XML or JSON directly. Change of representation format is possible in the future.

See Also

- Prism Data Structures
- Prism Schema
- Using Prism Objects