DML Reference

This document is just the beginning of what should become a complete reference to the DML language.

Currently, it contains only a few elements (the new ones that are being added to the language).

Entity type names

Package declarations

Using the keyword "package" it is possible to define a common prefix for all the entity-type names that appear after that declaration. After a package declaration such as

```
package some.qualified.id;
```

all the *relative* entity type names will be prefixed with "some.qualified.id.".

For instance, the following class declarations are interpreted as shown (assuming no previous package declaration):

```
class name;           =>   class some.qualified.id.name;
class another.name;   =>   class some.qualified.id.another.name;
```

A package declaration remains in effect until another package declaration is found.

An empty package declaration is written as:

```
package;
```

and its effect is to make all the following entity type names as absolute.

Absolute entity type names

When a package declaration is in effect, it is still possible to use an *absolute* entity-type name, by starting the name with a ".". The name of the entity-type, however, will not have the dot, as shown below:

```
package foo;

class bar;        =>   class foo.bar;
class baz.quux;   =>   class foo.baz.quux;
class .abs.name;  =>   class abs.name;
```

Value types

The slots of an entity type must always be of a certain value-type.

But, even though there are a few value-types built into the DML, typically, each domain model will use a different set of value-types.

So, in the DML it is possible to declare new value-types (only the declaration that the type exists, rather than its definition, which is, at least for now, out of the scope of the DML).

There are two possible value-type declarations: one for enums and another one for other kinds of value-types. In the first case, the externalization of the values of the value-type are handled automatically by the DML compiler. In the latter case, it must be specified in the value-type declaration how should the values of that type be externalized.

The syntax for enums is the following:

```
enum value-type-fully-qualified-name  [ as alias-identifier ] ;
```

The syntax for non-enum value-types, however, is this:

```
valueType value-type-fully-qualified-name [ as alias-identifier ] {
   <externalization-clause>
   [ <internalization-clause> ]
}
```

where <externalization-clause> must be present and has the following form:

```
externalizeWith {
  valueType1 methodName1();
  valueType2 methodName2();
  ...
```

```
}
```

For ease of reference in the text that follows, we will call each of the "valueType methodName();" declarations that appears within the externalization-clause block delimited by braces an "externalization element".

The value-type of an externalization element must have been previously declared as a valid value-type (and, therefore, it cannot be the same value-type of the current value-type declaration). Note, also, that this restriction prevents cycles in the externalization relations.

The semantics of an externalization-clause for a value-type VT is that a value of VT should be externalized as a series of one or more values of other (more primitive) value-types, in the order specified by the externalization elements within the externalization-clause. Each externalization element describes the type of one of those values and what method should be called to get the externalization value.

The current code generator assumes that if the method name of an externalization element contains a dot, then it is a static method with a single argument type assignable to a value of the value-type VT. Otherwise, it is a no-argument method of the class implementing the value-type.

Unlike the <externalization-clause>, the <internalization-clause> is optional. If no <internalization-clause> is specified, a value is reconstructed by calling the constructor of the value-type with the values corresponding to the external representation of the value, in the same order as they were specified in the <externalization-clause>.

If, however, an <internalization-clause> is specified, it must have the following form:

```
internalizeWith methodName();
```

In this case, the method called "methodName" should be used to reconstruct a value of the value-type VT.

For instance, to externalize a java.math.BigDecimal as a String we may use this:

```
valueType java.math.BigDecimal {
  externalizeWith {
    String toString();
  }
}
```

because the java.math.BigDecimal.toString() method creates an external representation of the BigDecimal that may be used to reconstruct the BigDecimal value again with

```
new BigDecimal(external-representation-as-string)
```

If, on the other hand, we have a class that has no pair getter/constructor that allows us to do this, such as the java.util.Currency class, but that has a static method that allows us to reconstruct a value of the class, then we may write it like this:

```
valueType java.util.Currency {
  externalizeWith {
    String toString();
  }

 internalizeWith getInstance();
}
```

Also, besides these simple cases where a given value is externalized as another single value, it is possible to have value-types that may be externalized as more than one value. For instance, consider the case of a Money value-type that represents a given amount on some currency. If the amount and currency are sufficient to completely describe a value of Money, and assuming the appropriate methods and constructor on the Money class, we may use this in the DML specification:

```
valueType Money {
  externalizeWith {
    long getAmount();
    Currency getCurrency();
  }
}
```

Finally, if a given value-type is implemented by a class that does not provide either a method to create an external representation of it, nor a method to reconstruct it from the external representation, we may still create static methods in some other class and use those methods in the externalization and internalization clauses. For instance, assume that we want to externalize a Money as a String in the form "amount currency", and that there is no appropriate methods in class Money. The skeleton of a possible solution is the following:

```
class MoneyExternalization {
  public static String moneyAsString(Money money) {
    return String.valueOf(money.getAmount()) + " " + money.getCurrency().toString();
```

```
  }

 public static Money moneyFromString(String moneyExtRep) {
   String[] parts = moneyExtRep.split(" ", 2);
   long amount = Long.parseLong(parts[0]);
   Currency currency = Currency.getInstance(parts[1]);
   return new Money(amount, currency);
  }
}

valueType Money {
  externalizeWith {
    String MoneyExternalization.moneyAsString();
  }

 internalizeWith MoneyExternalization.moneyFromString();
}
```

Parametric value-types

It is possible to use parametric value-types in the DML, both when declaring that a value-type exists (in a value-type declaration), and when specifying the properties of an entity-type.

For instance, given a value-type declaration such as this:

```
valueType my.package.MyType as MyType {
  ...
}
```

it is possible to use this in an entity-type declaration as shown below:

```
class MyEntity {
  MyType<A,B> prop1;
}
```

It is also possible, though, to declare the parametric type in the value-type declaration:

```
valueType my.package.MyType<A,B> as MyType {
  ...
}
```

and then use it like this:

```
class MyEntity {
  MyType prop1;
}
```

The result in this case will be the same as above.

This is useful when MyType is most often used with the type parameters A and B, even if in some cases it may be changed. For instance, given the last declaration for MyType, it is still possible to use like this:

```
class MyEntity {
  MyType prop1;
  MyType<X,Y> prop2;
  MyType<Z,W<C,F>> prop3;
}
```

The Java class generated for MyEntity will have three properties, prop1 to prop3, with the following types:

```
my.package.MyType<A,B>
my.package.MyType<X,Y>
my.package.MyType<Z,W<C,F>>
```

Restrictions imposed of value types

As the name implies, value-types must be immutable. Actually, the backing implementation of a value type may be that of a mutable class. However, in such cases the programmer must take care not to change the contents of the value types created, so that they behave as immutable. The typical approach when 'changing a slot typed with a value type that is actually mutable' is to create a new value type instance, rather than to change the existing instance, which would lead to incorrect behaviour of this framework.

Should a programmer decide to create a value type that can internally hold an [Entity][Entity type names], the programmer should invoke that entity's externalization mechanism when externalizing the value type. This is to ensure

that no automatic serialization occurs on entities, which could be the case if the externalization of a value type simply returned an object that actually contained a reference to an entity.

Slot indexation and query

Usage:

Changes in the Domain Modeling Language

DML now supports metadata associated with each slot of a domain class. This way, creating an index over a slot boils down to adding the metadata to the respective slot using a JSON structure. Example:

```
class Person {
    {"unique":true} String email;
    String name;
}
```

This newly introduced metadata states that the email of a Person is considered to be unique. This domain information is then used to create an index of instances of Person using their emails.

The usage of JSON (as a universal and widely known language) to compose the metadata is the start of plans that shall lead to adding arbitrary metadata to slots in DML. For now, the DML compiler only supports the usage described above.

Compilation step

After enhancing the application's DML files with some unique slots, you must recompile the application so that the newly generated domain classes (the *_Base.java ones) are upgraded with index search methods.

Changing the application code

For each "unique" slot in some domain class, the previous step generated a static method in the respective domain class. The convention used is that the search by index queries are provided in methods:

```
TypeOfDomainClass DomainClassName.findByNameOfSlot(TypeOfSlot)
```

This means that the search methods receive an argument of the type of the indexed slot, and the return is of the type of the domain class that contains the indexed slot.

Using the example presented above, it would generate code as follows:

```
public class Person_Base {
  (...)
  public static Person findByEmail(String email) {
    (...)
  }
}
```

This allows a search of a given Person by its email. The typical usage is to refactor code similar to the following:

```
String email = (...) // some email
Person p = null;
for (Person person : MyApplication.getRoot().getUsers()) {
  if (person.getEmail().equals(email)) {
     p = person;
     break;
  }
}
(...) // use Person p for business logic
```

...into:

```
String email = (...) // some email
Person p = Person.findByEmail(email);
(...) // use Person p for business logic
```

Not only the code becomes clearer, but also the implementation of the index is such that it performs better than the naive iterations/search. Benchmarking in TPC-W showed 30% to 70% better performance throughout the different workloads.

Limitations

1. If a slot is declared as unique, we rely on the domain model programmer to be accounted for that action. This means that we do not perform consistency verifications over that predicate (at least for now). Having said that, if

such statement is violated at runtime, the behavior of a search query for a non-unique key K is to return the instance that had its K slot last changed.

2. We do not index objects whose indexed slots have had their values set to 'null'. This means that 'null' is not considered a unique value.

3. Assume that your application was already working for some time, and thus has contents in its persistence. Now you add metadata to your DML to declare some slots as unique, and change some code to use the newly added index searches. The next time you bootstrap the application, we detect the newly added unique slots' declarations, and create the indexes. Yet, we do not populate them. This poses a limitation/problem if your persistence already has contents (as stated in this example)! As a workaround, using the above example, when you add the new unique slot 'email' to 'Person', and your database already contains instances of 'Person', you may run a script after booting the application that sets the email of every 'Person' instance to its own email (basically doing p.setEmail(p.getEmail())). In the future, we may provide an automatization of this process.