Goja - Generator for Oracle/Java-Applications

Michael Löwe[1]

Abstract

Goja supports model driven development of Java-applications whose persistence is provided by an Oracle relational database. Given a textual model in a Java-like language, Goja generates (1) the Oracle persistence, (2) an Java-application-server, and (3) a standard Java-Swing-Client. The application server communicates with the database via JDBC. The client accesses the application server via XML-RPC over HTTP. Goja enforces a fixed system architecture, which cannot be adjusted or influenced by the user. Therefore, the user is relieved from technical issues and can concentrate on functional aspects of the application. Even newcomers in information system design can develop complex multi-user applications quickly using Goja. Goja is especially useful in teaching system design, since, on the one hand, it generates pure Java- and PL/SQL-code without any hidden interpretation of comments or annotations and, on the other hand, realizes many design and enterprise patterns (from proxy to active object) in instructive contexts.

Contents

Quick Reference

This chapter provides a short reference to all features of Goja, especially its major design decisions, the underlying architecture, and the model language.

1 michael.loewe@fhdw.de

Architecture

Goja generates a three-tier-architecture. The persistence is created as a Oracle relational database. The application server is realized by a XML-RPC-web-server, which delegates each client call to a generated Java-framework. The client is generated as a Swing-application with XML-RPC-interfaces to the application server.

Persistence

Goja provides a canonical persistence of class models in relational databases. It supports single-inheritance of classes and multi-specialization of interfaces. It uses the pattern "one table for each inheritance hierarchy" for mapping object to rows in relational tables. Each relational table row is decorated with an information about the type of the object in this row in order to control type conformance of association within the database itself. For this purpose, the inheritance hierarchy of the class model is reflected in the relational model and checked prior to any update of associations.

Due to multi-specialization of interfaces, associations between objects cannot be mapped to foreign keys. Instead each association is realized by a pair (classId, objectId) which points to an object (objectId) in the right table (classId). Single-valued associations are mapped to column pairs in the table of the source class. Multi-valued associations are realized by separate reference tables. For simple set-valued associations, Goja generates a link table between the table of the source class and the tables of the target classes[2]. For hash table associations, Goja generates a (3-valued) link table between the table of the source class, the key table (resp. the key value, if the key is of base type) and the tables of the target classes.

In order to guarantee consistence of the database w.r.t. the underlying class model, there is no direct access to the tables in the database. Instead each database update has to be performed by a call to one of the Goja-generated Oracle-packages. For each class a facade package is generated, which provides the necessary getter and setter operations for single-valued attributes, the necessary adder and remover operations for set-valued attributes as well as the operation that creates new objects of this class.

Application Server

The application server is a simple web-server which understands XML-RPC-calls only. Each user which connects to the server obtains a separate "Server"-object, which is responsible for his calls. There is only one connection possible per user. Thus, the "Server"-object provides a single-user environment on the application server.

Nevertheless, all users of the application server work on the same model objects. This is supported by a common object cache for all connections. Every update to objects in the server is made persistent in the Oracle database immediately. For this purpose, the server uses generated JDBC-calls to the Oracle-facade-packages.

Every user connection employs an auto-commit connection to the database. Thus, transaction support is not provided for these connections. In order to achieve transactional behavior, explicit commands have to be given to some user-defined "Active Objects", which use their own, non-auto-commit connection to the database.[3] Therefore, Goja supports the explicit distinction of a single-user area (standard user connections) and a multi-user area (active objects) on the server.

2 There can be several target tables due to multi-specialization of interfaces.
3 Active objects can be defined in the Goja-modeling-language.

The server uses a lazy-loading mechanism. Objects are only fetched from the database when needed. Objects that have not been fetched from the database are represented as proxy-objects. Goja uses a two-level proxy mechanism. Each object-valued reference is given by a proxy of the right type. These proxies point to so-called in-cache proxies (ICProxy) which handle different copies of the same object in different non-committed transactions. Finally the in-cache proxies point to the real objects.

The server does not know the IP-addresses of the clients. The server only reacts on client calls. Therefore, the event mechanism between server and clients uses client polling. The server disconnects a client, when the client is inactive for a specified period (time-out).

Client

Since all servers run as web-servers, a client can connect to a server by just using its URL via http (XML-RPC). Once connected, a client can (1) inspect his "Server"-object by just navigating its associations, (2) call all operations of the "Server-object" and (3) connect to any other accessible object whose class has been declared to be a server (see language description below).

Also the client employs a lazy loading mechanism. But there is no cache, since the environment is single-user. Thus, there is no need for transactions and, consequently, the proxy mechanism is simple: Each association is represented by a proxy of the right type which, when loaded, directly points to the real object. Due to the absence of a cache, the same object may be loaded and instantiated twice on the client (use "equals" instead of "==").

Objects on the client do not propagate their changes to the server. All changes of objects on the server have to be stipulated by explicit calls to the operations of some "server-object". The setter operations on the client are only used within the standard implementation of the Swing-Table-Model.

The client polls the server by calling "hasChanged" periodically. If the server answers "true", the operation "handleRefresh" is called on the client side. This provides a simple event mechanism. A client can disconnect by simply stopping the polling.

Each client object implements the Swing interfaces "TreeModel" and "TableModel" in order to simplify the GUI-design. Appropriate "Listeners" can be registered and get notified, if GUI-edits have changed the client object (note that the corresponding object on the server has not been changed yet). These notifications can be propagated to the server using some explicit calls via XML-RPC. In a tree model each object shows all its associations to other objects as child nodes. The table model for an object has two columns. The first column contains the names of the object's base type attributes and the second column contains the current value for that attribute.

Language

This section provides a short description of all Goja modeling language features. The language syntax is similar to Java. Wherever concepts in Java and in Goja coincide, the syntax is the same.[4] A Goja-model consists of interfaces and classes. There are special interfaces, called hierarchies, that model aggregations. And there are special classes, namely singletons, exceptions, and servers, which model classes with exactly one instance, exceptional results that can be "thrown" by operations resp. classes that publish there interface in the internet.

4 For the time being, Goja does not provide any syntax for statements and expressions. Methods have to be programmed in the target code after the Goja-generation.

The syntax of a Goja-model is given by the following rule, the components of which are explained in more detail in the following[5]:

```
S1: model := model <name> {<interface>|<aggregation>|<class>|<singleton>|
      <exception>|<server>|<string-subtype>}*[{||}]
```

Classes and Interfaces

As Java, Goja distinguishes between classes and interfaces. Interfaces are abstract and contain operations only. Classes can be abstract or not and can contain arbitrary fields, i. e. attributes and operations.

Interfaces

The syntax for interfaces is as follows:

```
S2: interface := [{active|transient}{server-only}]{string-factory}interface
      <name> {extends <interface name>+[|,|]} <operation>*[{||}]
```

The only difference to Java-syntax are the optional modifiers (1) “active”, (2) “server-only”, (3) transient, and (4) string-factory:

1. Each class that implements an active interface supports the active object pattern for all operations of the interface. For each operation in the interface, there is an additional operation with one additional parameter of type “Invoker” and the return type normalized to “void”. Calling this operation results in an asynchronous call of the original operation in a separate thread-of-control of an active object “CommandExecuter”. The “CommandExecuter” performs its JDBC-calls to the database via its own connection. The connection's auto-commit is switched off. The thread commits after each call and returns the result to the invoker asynchronously (via “handleResult” resp. “handleException”). The thread rolls back if an exception occurs. (Also individual operations in a non-active interface or class can be declared “active”, see below.)
2. Interfaces that are “server-only”, are not generated in the client. Therefore all attributes that have a “server-only” type are also suppressed in the client.
3. The modifier “transient” declares all objects of this type to be not persistent.[6] The modifiers “active” and “transient” exclude each other, since the generated command-objects for the asynchronous call are persistent and need a persistent receiver.
4. The modifier “string-factory” generates a string subtype as a choice of the names (view-name, see below) of all concrete subtypes of the interface. Additionally a static string factory method is generated (in the meta package) that provides the necessary case distinction for the names.

The specialization hierarchy shall be consistent with respect to the transient specification:

C1: Transient interfaces shall only extend transient interfaces and non-transient interfaces shall only extend non-transient interfaces.

---

5 Elements in {}-brackets are optional. The notations $x_1 \mid x_2 \mid \ldots \mid x_n$ and $\{x_1 \mid x_2 \mid \ldots \mid x_n\}$ mean that exactly one resp. at most one of the given options is possible. Elements in []-brackets can be given in any order. Elements in <>-brackets are non-terminals. The symbol “*” means possible empty list and the symbol “+” means non-empty list. The decoration [*begin*|*separator*|*end*] of “*” and “+” provides the symbol in front of the list “*begin*”, the separator symbol between consecutive elements in the list “*separator*”, and the symbol behind the list “*end*”.

6 A transient class has the same semantics as a class in which all attributes are declared transient, see below!

It is not an error if transient and non-transient interfaces are mixed within the same specialization hierarchy. But there is no chance for a class to be derived from the special interface in this situation, see below.

Aggregations

Goja provides special support for aggregations. A model file can contain several aggregations. And several object-valued attributes (single- or set-valued) can take part in an aggregation. It is also possible that an attribute is part of several different aggregations. The generated code takes care that all aggregations are *hierarchical*. (Paths along links which are typed in attributes of the same aggregation do not have cycles!) Due to the hierarchy property, aggregations are declared using the following syntax:

```
S3: aggregation := hierarchy <name>;
```

Aggregations are translated into special interfaces. The `setter`- and `adder`-Operations of the participating attributes (see below) throw a `CycleException` if their execution would result in a cycle along links of the same aggregation.

Classes

Goja classes have the following syntax:

```
S4: class := [{active|transient}{server-only}{abstract}]{string-factory}class
      <name> <singleton-subtype-name>*[(|,|)]{extends <class name>} {implements
      <interface name>+[|,|]} {covers <concrete class name>+[|,|]} <field>*[{||}]
```

The differences to Java are:

1. The modifier “active” has the same effect on classes as on interfaces, see above.
2. The modifier “server-only” has the same effect on classes as on interfaces, see above.
3. The modifier “transient” declares all objects of this class to be not persistent. (The modifiers “active” and “transient” exclude each other.)
4. The short way of declaring singleton subtypes in the `<singleton-subtype-name>*[(|,|)]` phrase right behind the class name.
5. The modifier `string-factory`, see interface above.
6. Goja supports multiple-inheritance by implementing the “Inheritance-by-Delegation”-pattern. Each class C can extend one class and can *cover* many classes. The class C becomes a subtype of the extended class and of all covered classes. With the cover-mechanism, it is possible to realize the “one table per class”-strategy of relational persistence for class models.

The inheritance hierarchy must be consistent with respect to the transient specification:

C2: Transient classes can only extend or cover transient classes and non-transient classes can only extend or cover non-transient classes.

C3: Transient classes can only implement transient interfaces and non-transient classes can only implement non-transient interfaces.

The inheritance hierarchy must be consistent with respect to the server-only specification:

C4: If a class is not server-only, all its direct superclasses (extends and covers) must not be server-only.

The “Inheritance-by-Delegation”-pattern requires object instantiation of the covered classes. Therefore a concrete class CONCXXX is added for each covered abstract class XXX.

Goja produces a static factory method for object creation in each concrete class C, i. e. `C.createC(...)`. This method shall be used for object creation (instead of "`new C...`") in order to get the database consistent.

A field is either a declaration of an operation or a declaration of an attribute, see below.

Goja supports special types of classes, which are describe in the rest of this section.

Singletons

Goja directly supports the singleton pattern:

```
S5: singleton := [{transient|active}{server-only}] singleton <name>{extends
      <class name>}{implements <interface name>+[|,|]} {covers <concrete class
      name>+[|,|]} <field>*[{||}]
```

A singleton class C does not get a factory method for object creation. Instead, it gets a static access operation to the one and only object, i. e. `C.getTheC()`. This operation generates the instance if it is not generated yet. Therefore, a singleton class cannot be abstract. Since coverage requires a private instance of the covered class for the subclass, we require:

C5: Singletons cannot be covered.

Exceptions

Exceptions are special *transient* classes in Goja.

```
S6: exception := [{abstract}{server-only}] exception <name> {extends <exception
      name>} <field>*[{||}]
```

All exceptions that are thrown by server operations and, possibly, sent via XML-RPC to clients must be declared as exception classes in the Goja model file.

Exceptions form their own inheritance hierarchy:

C6: Exceptions can extend exceptions and can be extended by exceptions, only.

C7: Exceptions cannot cover other types or be covered by other types.

Servers and The Server

Goja distinguishes between standard classes, whose operations can only be called on the server, and server classes, the operations of which can be called by clients via HTTP and XML-RPC. The server syntax is as follows:

```
S7: server := [{transient}{abstract}] server <name> {extends <class name>}
      {implements <interface name>+[|,|]} {covers <concrete class name>+[|,|]}
      <field>*[{||}]
```

Server classes, standard classes, and singletons share the same extends-hierarchy, i. e. servers, classes, and singletons can extend any server, class, or singleton. Coverage is restricted in the following sense:

C8: A server cannot directly or indirectly cover another server.

In order for clients to be able to connect to the application server, the Goja model must contain a server named "Server". This class - besides the user-defined fields and operations - provides the mechanisms for initial client connection and authentication. For this purpose, the server "Server" is augmented by some system attributes, that require that the server class "Server" is not transient:

C9: The server "Server" shall not be transient if objects of this class are meant to provide

initial authentication.

Having a first connection, a client can connect to arbitrary many server objects.

Internal Types, External Types, List Types, and Map Types

Base Types

Each Goja model contains the following *base types*:

Goja	Oracle	Server	Client
String	VARCHAR(2000)	java.lang.String	java.lang.String
Integer	NUMBER	long	long
Date	DATE	java.sql.Date	java.util.Date
Timestamp	TIMESTAMP	java.sql.Timestamp	java.util.Date
Text	CLOB	java.lang.String	java.lang.String

String subtypes

Using *regular expressions*, Goja allows to define arbitrary *subtypes of the base type "String"*. The syntax is as follows[7]:

```
S8: string-subtype := string subtype <name> ::= ' <reg-exp> ';
```

String subtypes can be used as a type of a field and of an operation parameter. They are not allowed as return types.

The syntax for regular expressions is as follows:

```
S9: reg-exp := <var-def>*[||;|;] <var-reg-exp>
S10: var-def := @<name>@ = <var-reg-exp>
S11: var-reg-exp := <base-var-reg-exp> {*|+|?}
S12: Base-var-reg-exp ::= <var-app> | any-character | Sequence | Choice
S13: var-app := @<name>@
S14: sequence := <var-reg-exp>[(||)]
S15: choice := <var-reg-exp>[[||]]
```

C10: Variables must be defined before they are applied.

C11: White space within an regular expression is skipped.

C12: Escapes are introduced by "\".

Internal and external types

The interfaces, classes, servers, singletons, subtypes of "String", and exceptions defined in the model are the *internal types* of the model.

7 If the first string subtyping is introduced after some generations without subtyping, the class viewClient.wizard has to be dropped and newly generated!

In all clauses where interface names are expected (extends-clauses in interfaces and implements-clauses in classes, servers, and singletons), any interface can be referenced that is defined in the same model. Additionally, external references to interfaces outside the model are possible.

These external type references consist of a name preceded by the keyword **`extern`**. The operations of the external interface, however, shall be explicitly declared in the class or interface in order to obtain standard implementations in the proxy classes. The modifier "system" prevents the generation of Goja Exceptions.

List and map types

For any internal type `T`, Goja generates an internal *list type*, syntacticly denoted by `T**`. In the application server, there is only a small interface to list types:

(1) `add(..)`, which adds at the end of the list, and

(2) `iterator()`, which provides a database consistent Iterator.

(3) Removal of elements from the list is only possible via the iterator's operation `remove()`.

Goja list types are represented as cursors in Oracle, instances of specializations of the class `SearchListRoot` in the server, and as instances of `java.util.Vector` in the client.

For any pair of internal types `T,T'`, Goja generates an internal *map type* from `T'` to `T`, syntacticly denoted by `T**T'**`. In the application server, there is only a small interface to list types:

(1) `put(T' key, T value)`, which changes the value in the map at *key* to *value*,

(2) `T get(T' key)`, which provides the *value* at the *key* in the map,

(3) `void remove(T' key)`, which removes the entry at key in the map, and

(4) `TSearchList getValues()`, which provides a list of all values in the map.

Goja map types are represented as 3-valued relations (owner, key, value) in Oracle, as `Hashtable<T',T>` in the server, and as instances of `java.util.Vector`, comprising the list of values, in the client.

Operations

The operation declaration syntax is similar to Java:

```
S16: operation := {[{database{changing}}{abstract}{active}{system}
       {synchronized}{no-view|server-only|checked}{event}] <return type> <name>
       <parameter>*[(|,|)] {throws <exception name>+[|,|]};

S17: parameter := <parameter type> <name> {<path>}[{checked}{PASSWORD}{final}]

S18: path := <server path> | <parameter path>

S19: server path := <attribute name>+[{|;|}]

S20: parameter path := <attribute name>+[{;|;|}]
```

A parameter type can be any base type, any internal type, any internal list type, and any external reference, i. e. the keyword **`extern`** followed by a possibly qualified name.[8]

C13: Operations in a server or active operations cannot use external types as parameter types.

A return type is any parameter type or the void-type denoted by **`void`**;

C14: Operations in a server or active operations cannot use external types as return types.

8 An example for an external reference using a qualified name is: **`extern java.util.Iterator`**.

C15: An operation can only throw exceptions that are declared in the same model.

C16: An operation cannot use exceptions as parameter or return type.

Any operation that is declared as `database` or `database changing`, results in a function declaration in the corresponding Oracle package. The application server implements the operation by delegating to this database function via JDBC. The final method for operations of this type must be implemented in the Oracle package body using PL/SQL. Operations that are declared `database changing` shall, additionally, provide a collection of all changed database rows (classId, objectId) in the generated `out`-parameter. This collection is used by the application server to consistently update the corresponding objects in the cache.

An `active` operation can be delegated into some transactional thread, see `active` on the type level above.

A `system` operation does not throw any Goja-specific exceptions that *must* be caught. It throws an error of type `PersistentError` instead, that *can* be caught. This is especially useful for the import of external interfaces like `Runnable` or `Comparable`.

C17: The modifiers `no-view`, `server-only`, and `checked` for operations can only be used in a server.

The first suppresses the generation of a context menu entry for this operation.

The second hides the operation completely from the client, i. e. it is not published as a web-services via XML-RPC. The modifier `server-only` implies `no-view`.

The third generates a call to `check4op` if it is a modifier for an operation named `op`. This check operation needs to be implemented in the client class outside the generated part! Returning `null` or the `empty string` indicates a positive check result. Returning another string *s* indicates a negative check result together with an appropriate message, namely *s*.

Operations specified as `events` notify all registered observers when called.

All specifications for parameters except name, type, and the `final`-modifier (with standard Java semantics) are useful in server operations only, since they influence the generation of the swing client only. The `check`-modifier generates a client call to a filter operation which has to be programmed by the user. Only parameters that pass the filter are allowed in RPC-calls. If the checked parameter is object-valued and non set-valued and the first parameter, the filter operation has to be implemented in the client outside the generated part. In all other cases, the filter operation has to be implemented (outside the generated part) in a new class that extends the appropriate `UserXXXSelectionPanel`.

C18: The modifier `PASSWORD` is allowed for `String`-typed parameters only.

It replaces the standard string input panel by a special password-panel in all generated GUI-parts.

A path can be used to restrict the choices for the selection of object-valued parameters.

C19: Path specifications are possible for object-valued parameters only.

The path specifies the root of the object-tree (object and all its associated objects as children) that is offered for selection. A server path starts syntactically with the symbol "`{`" and semantically at the current server-object. A parameter path starts syntactically with the symbol "`{;`" and semantically at another parameter, the name of which is the first step in the path, of the same operation.

C20: Steps in pathes must reference object-valued attributes and only the last can reference a set-valued attribute.

C21: For the time being, there is no cycle check among operation pathes.

Attributes

All attributes defined in a Goja model are accessible by public getter and setter operations. The syntax for attributes is given by:

```
S21: [{no-view|server-only|skip-view|sorted-view}{client-as-string}{transient |
       indexed | observe | symmetric {cached} | one-to-one {cached}} {prior}
       {final | specializable}{derived}{filtered}] <attribute type> <name>
       {hierarchy <name>*[|,|]};
```

The type of an attribute can be any internal type, string subtype, internal list type, internal map type, or Goja base type.

The modifier `no-view` excludes the attribute from being visible in the `TreeModel`- and `TableModel`-implementations in the client. The modifier `server-only` implies `no-view` and hides the attribute completely for the client.

The modifier skip-`view` of an attribute `a` of type `T` in a class `C` prevents links of type `a` from being shown in the client's standard treeview. Instead, all associated objects of a `C`-object are shown directly as children of the class-`C`-object.

C22: The type of an skip-view attribute must be internal and must not be a list or map type.

The modifier `client-as-string` specifies that the target of each link which is typed in the association is communicated to the client as a simple string (even if set-valued). It produces a corresponding `String XXXAsString()` operation (with associated TODO) in the containing class.

C23: The type of an client-as-string attribute must not be a Goja base type.

C24: The client-as-string modifier cannot be combined with server-only, skip-view, and sorted-view.

The modifier `sorted-view` is suitable for set-valued attributes and generates an infrastructure on the client (in the `Proxi`) that allows to sort the entries in the client-vector associated with the attribute (by implementing a `comparesTo`-operation at the generated TODO).

The modifier `transient` excludes the attribute from the persistence services, i. e. the attribute exists in the application server but not in Oracle.

The modifier `indexed` for attribute A with type T in class C generates a static application server operation `public static CSearchList getCByA(T index)` in C. This operation provides all C-objects whose A-attribute matches `index`.

C25: The type of an indexed attribute must be a Goja base type.

If the type of an indexed attribute is String, standard database wild cards (%, _) are supported.[9]

The modifier `observe` realizes an observe pattern between the class, in which the attribute is defined and the type of the attribute.

C26: The type of an observed attribute must not be a base type.

If the type of the attribute is a non-list type, the `setter` automatically `deregisters` the receiver at the old value and `registers` the receiver at the new value. If the type is a list type, the `add`-operation `registers` and the `remove`-operation via the `iterator` `deregisters`. If the type is a

9 Not available, if the generator is advised to generate without database!

map type, the put-operation registers at the new value and deregisters at the old value if this was not null.*/+

The necessary interfaces ObsInterface and SubjInterface as well as the protocol for registering and deregistering objects are generated automatically. Objects of class C notify all current observers after each call of one of C's event-operations. Objects of class A observing an attribute of type C, i. e. observe C c is a field of class A or of an superclass of A, can react by implementing c_update(model.meta.CMssgs event). The event-Parameter stores all information about the call of the event-operation.

By the modifier symmetric, an association a in class C becomes symmetric and Goja provides an inverse getter operation public CSearchList inverseGetA() in the type T of the attribute. If this type is an interface, the inverse getter is added to all implementations.

The modifier one-to-one does almost the same as symmetric. But it requires that the result of the inverse getter is always a search list with a single entry (at most one and at least one). The inverse getter operation selects this element in the list and returns it.

Adding the modifier cached to a symmetric or one-to-one attribute is useful if the inverse getter "final", i. e. provides always the same result. The result of the first database-based calculation of the inverse getter is cached in the application server. This cached value is provide in all consecutive calls to the inverse getter.

C27: The type of observed, symmetric, or one-to-one attributes cannot be a base type.

The modifier prior requests a value for that attribute as a parameter in the factory-method (create) for that class.

C28: Prior attributes cannot be set-valued.

The modifier final inhibits the generation of a setter and checks that the attribute value is not null after creation.

C29: Final attributes cannot be set-valued or have base types.

The modifier specializable implies final. It allows the attribute to be "*overwritten*" in a subclass by an specializable attribute with the same name and a more special type.

The modifier derived inhibits the generation of a setter and the getter must be implemented by the user.

C30: Derived attributes cannot be final, observed, sysmmetric, prior, or indexed.

The modifier filtered provides a mechanism to filter a set-valued attribute before it is transferred to the client. (This can express for example sorting on the server.) For this purpose the generated operation filterFor... has to be implemented (see TODO).

C31: Filtered attributes must be set-valued and cannot be observed.

Names

Goja uses the names of the types, attributes, and operations in the model file to create (partly combined) names for objects (tables, columns, packages, and operations) in Oracle and objects (classes, interfaces, attributes, and operations) in Java. Since the maximal length of names in Oracle is 32 characters, the length of the names in a Goja model file must be restricted to 11 characters to fit all purposes.

To overcome this limitation, at least for the Java part of the generated code, Goja names are structured: <longName>##<shortName>. The long part can have arbitrary length, the short part

is restricted to maximal 11 characters. Java objects are named using the long part. For Oracle objects, Goja uses the short part. If a name is not structure (i. e. does not consist of two parts separated by ##), Goja supposes that the long part for Java and the short part for Oracle are identical.

C32: Beware of using short names that are keywords in Oracle![10]

The long name is also the default name that is used as a name for the item in the standard graphical user interface (GUI). If this name does not fit, it is allowed to specify a separate name for the GUI. The format is: `<longName>##<shortName>@@<view-name>`. View-names can contain "_"-characters. They are substituted by blanks in the view. The `<shortName>` and `<view-name>` are optional. If both are specified, the short name comes first.

10 A good pragmatic is to build oracle names from the long names by dropping all vowels.