Value Proposition
Products
Qare
JXWeb
JXUnit
Quick
Development Team
Site Map

Products

QARE	Framework for distributed web services
JXWeb	Test scripting tool for web services
JXUnit	Test scripting tool for Java
Quick	Tools for Java programmers for working with XML

Wiki Wiki Topics

Consolidated JavaDocs: Online/Download

QARE is a framework for distributed web services which supports Internet Application Integration (B2B), P2P and B2C. QARE uses a delegation trust model to combine access control with a Peer-to-Peer network. QARE applications are fast, distributed systems built using asynchronous messaging and push technology.

Status: Beta Release

Licenses: **LGPL**

Download: QareOnTomc at3.0b5.1(Binary) Qare3.0b5.1 (Source Code)

Installation Guide

Release Notes

Statistics: 30 Days Monthly

Qare Home

Pilot projects wanted! Write to blaforge@jxml.com

Top

JXWeb

JXWeb is a scripting tool for testing web services. It is an extension of JXUnit, with additional commands for navigating web pages.

When using JXWeb, generally no special test data is needed. This means that custom binding schema can be avoided when writing test scripts, making it much easer to use than JXUnit.

JXWeb uses HTTPUnit for web page access and navigation.

Status: Production Release
Licenses:Apache
Download: JXWeb1.1
Installation Guide
Release Notes
Statistics: 30 Days Monthly
JXWeb Home

Top

JXUnit

When code is tested, it is run against one or more sets of data. JXUnit is a test scripting system which uses Quick to convert XML into test data. (JXUnit is built on JUnit.)

Because JXUnit uses Quick to transform XML documents into test data, some knowledge of Quick's binding schema is needed to be able to write JXUnit test scripts.

Status: Production Release, Stable
Licenses:Modifed BSD
Download: JXUnit3.1.3
Installation Guide

Release Notes
Statistics: 30 Days Monthly
JXUnit Home

Top

Quick

Quick is a Data Binding system for transforming XML into Java objects and Java objects into XML. Quick builds on QJML, a binding schema which connects XML elements to Java classes.

Quick can be used to generate the Java code for data classes, but it keeps the marshalling code separate from data classes. This has two advantages:

1. Many classes not designed for Quick can be used.
2. The same class can be used with more than one binding schema.

Quick includes utlities for transforming DTDs into QJML, QJML into marshaling logic, QJML into documentation (HTML), and QJML into data classes. These tools are written using an MVC framework which is a part of Quick.

Status: Production Release, mature and stable
Licenses:Modifed BSD
Download: Quick4.3.1
Installation Guide
Release Notes
Statistics: 30 Days Monthly
Quick Home

Top

Value Proposition
Products
Qare
JXWeb
JXUnit
Quick
Development Team
Site Map

JXQuick Home Page

Download | Quick Summary Page | Quickutil Summary Page | Forum

JXQuick Home Page
The Quick Guide
Installation Guide
Quick Utilities

Quick

Pulling information out of code and putting it into data files is an effective way to keep code simple. First, you need a way to represent this information that is easy to process and rich enough to support a wide range of data structures. XML seems ideal for this. Quick gives you the facility to recombine this data with your code.

Quick is a tool for generating and processing XML. Quick converts arbitrary object structures into trees of XML elements. Converts Cross-linked XML documents into structures of objects.

Quick is a data modeling system for transforming XML into Java objects and Java objects into XML. Quick builds on QJML, a binding schema that connects XML elements to Java classes. Quick fully supports Java inheritance, including abstract and interface elements. The developer is given fine-grained control over code generation,so the generated code can extend and interoperate with pre-existing classes.

Quick works with Java Beans and Bean Property Editors. Developer-provided Bean Property Editors allow the use of custome data types (Java classes) when processing XML attributes and simple elements with text content. Quick provides a thread-safe framework (the ocm package) for simple and complex data transformations.

Quick provides utlities for transforming DTDs into QJML, QJML into marshaling logic, QJML into documentation (HTML), and QJML into data classes that are based on its MVC framework.

The Quick Guide

The object of this guide is to provide an overview of the concepts behind Quick and an introduction to some of the capabilities of Quick, as well as providing direction on how to use some of its features.

Contents

- Part I: An Overview of Quick
  - What is Quick?
  - Funding
  - History
  - Two kinds of XML
  - Document Validation, Error Messages and CLASSPATH
  - Inheritance
  - Quick's 3 Schema: QDML, QJML and QIML
  - QDML, Quick's XML Schema
  - A comparison of Quick and JAXB
- Part II: How to Use the Quick Utilities
  - A few things to get Started
  - Converting a DTD into a QDML Schema (cfgDtd2Qdml and cfgQdml2Dtd)
  - Setting the QDML Root (cfgSetQdmlRoot)
  - Converting QDML to QJML (cfgQdml2Qjml and cfgQjml2Qdml)
  - Generating Java from QJML (cfgQjml2Java2)
  - Generating QIML and schema factory classes (cfgQjml2Qiml and cfgQiml2Java)
- Part III: How to Use the Quick API
  - Using the Quick Runtime
  - Expressing Objects in XML
  - Building Objects from XML
- Part IV: Config
  - A Script for Validating Parameters
  - Scripts--A Potential Security Risk
  - Config Document Examples
  - Config Binding Schema
  - The Item Interface
  - A Generic Main Method
  - Processing Scripts
  - Invoking the Applicaiton
- Part V: Transforming XML
  - Model/View/Controller (MVC)
  - The Quick Utilities
  - MVC Design Issues
  - Example 1: cfgNoid
  - Example 2: cfgQjml2Html

- Part VI: Links

Part I: An Overview of Quick

What is Quick?

Quick is not a tool for processing XML. Rather, it is an extension to the Java language that uses XML. The Quick binding schema, QJML, models your Java code and makes explict the data model used by that code. Quick makes it easy to express graphs of JavaBeans using a custom grammer and to transform documents into validated assemblies of JavaBeans.

Quick is also a model/view/controller framework for advanced data transformations. The Quick utilities use this framework for schema transformations and code generation.

Quick has been developed for server-side processing, addressing issues like object reuse, thread safety and multiple class loaders.

top

Funding

Quick release 4, including this Guide, is sponsored by the Defense Advanced Research Projects Agency (DARPA). In particular we acknowledge the support of Dr. John Salasin and the Dynamic Assembly for System Adaptability, Dependability, And Assurance (DASADA) program, Contract Number F30602-00-C-0203.

Eariler releases of Quick were developed under DARPA SBIR 99.2-45, a joint submission by The Open Group and JXML Inc.

top

History

Work on Quick began in 1979, first under the name of **Coins** and then as **MDSAX**. The code base has been through 7 complete rewrites.

As Coins, the focus was on the dualistic potential of Java and XML, focusing on bi-directional conversions between XML documents and graphs of Java Beans. Coins was the leading binding technology of its day, while similar technologies focused on describing Java Beans (IBM's BML), using XML for serialization (Koala's KBML) and document processing (Michael Kay's SAXON). Like JavaSoft's JAXB, Coins included a custom XML parser, but eventually converted to SAX and AElfred.

MDSAX, short for Multiple-Document SAX, was a reimplementation of Coins based on SAX filters. It was a framework approach using heirarcial structures of filters to effect various transformations. These structures were themselves defined using an XML document.

Finally Quick took a meta-data approach, using a binding schema to describe the relationship between Java classes and XML elements. Quick4 introduced support for the entire Java inheritance model. And while earlier releases of Quick again focused on the Java and XML dualism, Quick4 introduced a model/view/controller approach for transformations. The Quick4 Utilities all build on this model/view/controller approach, putting an end to the high cost of maintenance experienced previously.

The Coins/MDSAX/Quick user base has grown steadily since 1997, reaching 2,000 downloads with Quick4.3. Meanwhile development has shifted to working on other products which use Quick: JXUnit, JXWeb and QARE.

top

Two kinds of XML

There are any number of ways of constraining and validating an XML document, but two of the most popular devices are DTDs and the W3C's XML Schema. (Relax and Schematron are alternatives which come quickly to mind.)

The W3C's XML Schema, because of its introduction of type, has strongly influenced the encoding of data into XML. I will borrow an example from Bradley Schmerl's paper on xAcme to illustrate this.

We want to represent a purchase order with a billing address and an optional mailing address. The addresses themselves can be either a US address or Australian. Here's an example using plain vanilla XML:

```
<purchaseOrder refNo="123">
    <orderDate>2001-02-27<orderDate>
    <shipDate>2001-03-01<shipDate>
    <USAddress>
        <name>Bradley Schmerl<name>
        <street>5000 Forbes Ave</street>
        <city>Pittsburg</city>
        <state>PA</state>
        <zip>15203</zip>
    </USAddress>
    <AusAddress>
        <name>Bradley Schmerl<name>
        <street>48 Main North Road</street>
        <city>Willaston</city>
        <postcode>5118</postcode>
        <state>SA</state>
    </AusAddress>
</purchaseOrder>
```

In the example above, the first address is always the billing address, while the optional shipping address always follows the billing address. And the element names always identify the type of data.

With the introduction of a type attribute in the W3C's XML Schema, elements are often used to identify the use of the data, rather than its type:

```
<purchaseOrder refNo="123">
    <orderDate>2001-02-27<orderDate>
    <shipDate>2001-03-01<shipDate>
    <billingAddress xsi:type="USAddress">
        <name>Bradley Schmerl<name>
        <street>5000 Forbes Ave</street>
        <city>Pittsburg</city>
        <state>PA</state>
        <zip>15203</zip>
    </billingAddress>
    <shippingAddress xsi:type="AusAddress">
        <name>Bradley Schmerl<name>
        <street>48 Main North Road</street>
        <city>Willaston</city>
        <postcode>5118</postcode>
        <state>SA</state>
    </shippingAddress>
```

```
</purchaseOrder>
```

Quick uses plain vanilla XML and supports the use of DTDs. SOAP, to name a well known example, uses the second form.

top

Document Validation, Error Messages and CLASSPATH

Quick uses a SAX parser to process XML documents. SAX parsers will validate an XML document against the DTD specified in the document.

On the other hand, Quick always validates XML documents against a binding schema. Documents do not always specify a DTD, or may us a DTD that is different from what is expected. Quick binding schema also provide additional type checking beyond what can be included in a DTD. By validating against the binding schema, there is better assurance that the document conforms to what is expected.

Quick does not always provide the best error messages. When developing a document like a QJML binding schema, it is best to specify the DTD for the document.

DTDs are specified as a URL in the DOCTYPE statement at the beginning of an XML document. To simplify things, Quick includes its own CLASSPATH protocol for accessing files from your classpath. (The DTDs for QDML, QJML and QIML are included in the Quick4util.jar file.) Here's an example of a QDML file which includes a reference to the QDML DTD:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qdml SYSTEM "classpath:///qdml.dtd">
<qdml>
      <text tag="hello"/>
</qdml>
```

top

Inheritance

Quick supports the full Java inheritance model as well as DTDs. Quick includes utilities to transform DTDs into one of its schema languages, QDML, and another utility to perform the reverse transform. Of course, when a QDML document makes heavy use of inheritance, the resulting DTD can get rather large.

There are two advantages to supporting Java inheritance. First, it simplifies maintenance. Second, it makes it easier to specify the data model used by the Java code. Generally it is better/easier to maintain a schema in QDML (or QJML) and then convert it to a DTD when needed.

Building on the previous example, here's a QDML schema that uses inheritance:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qdml SYSTEM "classpath:///qdml.dtd">
<qdml root="busDoc">

      <interface label="busDoc">
      <!--Anything derived from busDoc can be used as a root element-->
            <attributes>
                  <item coin="refNo"/>
            </attributes>
      </interface>
      <text tag="refNo"/>
```

```
        <bean tag="purchaseOrder">
              <implements>busDoc<implements>
              <!--Attributes are inherited by default-->
              <elements>
                    <item coin="orderDate"/>
                    <item coin="shipDate"/>
                    <item coin="address"/> <!--billing address-->
                    <item coin="address" optional="true"/> <!--shipping address-->
              </elements>
        </bean>
        <text tag="orderDate"/>
        <text tag="shipDate"/>

        <bean tag="address">
              <elements>
                    <sequence label="addrEle">
                          <!--sequence used to allow labeling for inheritance-->
                          <item coin="name"/>
                          <item coin="street"/>
                          <item coin="street" optional="true"/>
                          <item coin="city"/>
                    </sequence>
              </elements>
        </bean>
        <text tag="name"/>
        <text tag="street"/>
        <text tag="city"/>

        <bean tag="USAddress">
              <extends>address<extends>
              <elements>
                    <!--Elements require a cref for inheritance-->
                    <sequence cref="addrEle"/>
                    <item coin="state"/>
                    <item coin="zip"/>
              </elements>
        </bean>
        <text tag="state"/>
        <text tag="zip"/>

        <bean tag="AusAddress">
              <extends>address<extends>
              <elements>
                    <sequence cref="addrEle"/>
                    <item coin="postcode"/>
                    <item coin="state"/>
              </elements>
        </bean>
        <text tag="postcode"/>

  </qdml>
```

top

Quick's 3 Schemas: QDML, QJML and QIML

The Quick Data Markup Language (QDML) is a schema for describing XML, similar to the way DTDs describe XML, except for the addition of inheritance. Quick includes utilities for converting DTDs to QDML and back again.

The Quick Java Markup Language (QJML) is a binding schema. QJML is a super set of QDML, binding XML elements and attributes to Java classes, variables, JavaBean properties, and property editors. Quick includes utilities for converting QDML files into QJML and back again. Quick also includes utilities for converting a QJML into a set of simple Java classes. (Code generated from a QJML file does not include any marshalling and unmarshalling logic.)

Developers are expected to read and edit QDML and QJML files when using Quick. This is not the case with the Quick Internal Markup Language (QIML). Think of a QIML file as the compiled form of a QJML file. It is designed to be easy to process, not easy to read or edit. Quick includes utilities for converting QJML files into QIML and for generating marshalling/unmarshalling logic from a QIML file.

top

QDML, Quick's XML Schema

QDML is a subset of QJML that allows you to specify a data model (e.g. an XML markup language) without dealing with how that model is implimented in Java. Being less verbose than QJML, it is easier to read and to edit.

QDML is similar in capabilities to DTDs, except for the addition of the Java inheritance model and the introduction of some of the Java types.

When starting with a DTD, it is a good idea to convert it to QDML and then perform as much cleanup as possible before converting the result to QJML. And when learning Quick, QDML is the best place to start. Get familiar with the details of QDML first, before facing the additional details of binding XML to Java.

The QDML root Attribute

When designing QDML, we wanted to be able to convert a DTD into a valid QDML document, and DTDs do not specify which elements can be used as the top-level or root element in a document. So QDML does not require that you specify the root--the root attribute is optional.

QJML does require that you specify the root element. Before converting a QDML document to QJML, be sure to add the root attribute to the QDML element.

```
<qdml root="top">

      <interface label="top"/>

      <bean tag="hello">
            <implements>top<implements>
      </bean>

      <bean tag="goodbye">
            <implements>top<implements>
      </bean>

</qdml>
```

The Coin elements: abstract, interface, bean and text

The four elements abstract, interface, text and bean occur within a QDML document directly under the qdml element. The abstract and interface elements corrispond to abstract classes. The text element corrisponds to scalars and classes which can be specified with a simple text string, while the bean element corrisponds to JavaBeans. Collectively, these four QDML elements are called coins.

On the XML side, a QDML bean element always maps to an XML element. A QDML text element can be mapped to either an XML attribute or element, but not both. The QDML elements abstract and interface have no

corrisponding part in XML, just as abstract classes and interfaces have no instance objects.

The tag and label Attributes

In a QDML document, tag attribute always specifies an element tag name or an attribute tag name. The QDML elements bean and text, which are used to define elements and attributes, must always have a tag attribute.

The label attribute is used to identify the various parts of a QDML document, so that they can be referenced elsewhere in the document. Within the scope of a QDML document, the value assigned to a label must be unique. The QDML elements interface and abstract, which have no corrisponding parts in XML, must always have a label attribute. Other QDML elements which can be labled are sequence, selection and item.

When a tag attribute is specified, its value is also used as the default label value. But when both the tag and lable attributes are specified, that part of the QDML document must always be referenced using the value of the label attribute.

Text

A QDML text element corrisponds to Java scalars and classes which can be specified with a simple text string. And they map to XML attributes or elements. Quick predefines several types:

idref	A reference to an object defined elsewhere. (Aplies only to attributes.)
CDATA	A String or other object which is expressed as an XML CDATA section. (Not valid when defining attributes.)
PCDATA	A String or other object. (This is the default type.)
int	An int or an Integer.
short	A short or a Short.
long	A long or a Long.
byte	A byte or a Byte.
boolean	A boolean or a Boolean.
float	A float or a Float.
double	A double or a Double.
base64	A byte array which is base64 encoded when expressed as XML.
url	A URL.
char	A char or a Char.
BigDecimal	A BigDecimal.

Here's an example of a declaration of an element named x which can only be assigned the values 1, 2 or 3:

```
<qdml root="x">
      <text tag="x" type="int">
            <enum value="1"/>
            <enum value="2"/>
            <enum value="3"/>
      </text>
</qdml>
```

And here is a valid document:

```
<x>1</x>
```

Attributes

Attributes can be optional or required, but are required by default. Attributes can have a default value or a fixed value.

```
<qdml root="myBean">

      <bean tag="myBean">
            <attributes>
                  <item coin="french"/>
                  <item coin="fried" optional="true" value="twice"/>
                  <item coin="potatoes" optional="true" fixed="true" value="Idaho"/>
            </attributes>
      </bean>
      <text tag="french"/>
      <text tag="fried"/>
      <text tag="potatoes"/>

</qdml>
```

In the example above, the three attributes are defined as follows:

french	required
fried	optional, default=twice
potatoes	optional, fixed value=Idaho

And here is a valid document:

```
<myBean french="large"/>
```

Inherited Attributes

Attributes are, by default, inherited. But this can be turned off.

```
<qdml root="myBean">

      <bean tag="myBean">
            <attributes>
                  <item coin="french"/>
                  <item coin="fried" optional="true" value="twice"
label="myBean.fried"/>
                  <item coin="potatoes" optional="true" fixed="true" value="Idaho"/>
            </attributes>
      </bean>
      <text tag="french"/>
      <text tag="fried"/>
      <text tag="potatoes"/>

      <bean tag="easyBean">
            <extends>myBean</extends>
            <attributes inherited="false">
                  <item cref="myBean.fried"/>
            </attributes>
      </bean>

</qdml>
```

In the example above, the only inherited attribute is fried. Here is a valid document:

```
<easyBean fried="battered"/>
```

Elements

The elements within an element can be optional or required, but are required by default. Elements can also be repeating, occuring more than once.

As the order of elements is significant, a sequence of elements can be defined. Again, this sequence can be optional or required and can also be repeating.

A selection of alternative elements or sequences can also be defined, which can again be optional or required and also repeating. And of course, a selection can occur within a sequence.

```
<qdml root="a">

      <bean tag="a">
            <elements>
                  <selection optional="true" repeating="true">
                        <item coin="a"/>
                        <item coin="b"/>
                  </selection>
            </elements>
      </bean>

      <bean tag="b">
            <elements>
                  <item coin="a"/>
                  <item coin="a"/>
            </elements>
      </bean>

</qdml>
```

The following is a valid document:

```
<a>
      <a/>
      <a/>
      <b>
            <a>
                  <a/>
            </a>
            <a/>
      </b>
      <a/>
</a>
```

Inherited Elements, Sequences and Selections

Unlike attributes, element content is never inherited by default. The cref attribute is used to reference a similar QDML element that has been labeled. So an item can reference another item, a sequence can reference another sequence and a selection can reference another selection.

```
<qdml root="a">

      <bean tag="a">
            <elements>
                  <item coin="a" optional="true" label="a.a"/>
```

```
                </elements>
        </bean>

        <bean tag="b">
                <extends>a</extends>
                <elements>
                        <item cref="a.a"/>
                </elements>
        </bean>

</qdml>
```

One added advantage to inheritance is that a derived element can occur anywhere its parent can occur. In this case, element b can occur anywhere element a is valid. The following is a valid document:

```
<b>
        <b/>
</b>
```

MIXED and BIMODAL Elements

A mixed element contains both text and other elements. Mixed elements are not supported by Quick.

Quick does support bimodal elements, which can contain either text or other elements. However, bimodal elements are usually employed to support elements with both attributes and text content, as the QDML text element does not support attributes.

```
<qdml root="fun">

        <bean tag="fun" type="BIMODAL">
                <attributes>
                        <item coin="game"/>
                </attributes>
        </bean>
        <text tag="game"/>

</qdml>
```

Here is a valid document:

```
<fun game="tag">A good form of exercise!</fun>
```

ID and IDREF

XML documents have the structure of a tree, but there is a provision for adding additional links through labeling (adding IDs) and references (IDREFs).

The value of an ID must be unique within the document.

```
<qdml root="x">

        <bean tag="x">
                <attributes>
                        <item coin="ID">
                                <id/>
                        </item>
                        <item coin="REF" optional="true"/>
                </attributes>
```

```
                <elements>
                      <item coin="x"/>
                </elements>
          </bean>
          <text tag="ID"/>
          <text tag="REF" type="idref"/>

    </qdml>
```

Here is a valid document:

```
    <x ID="1">
          <x ID="2"/>
          <x ID="3"/>
          <x ID="4" REF="1"/>
    </x>
```

Wild

There is one case where tag is not require on text or bean: when the wild attribute is true. Wild allows an element to hold any attribute or any element.

```
    <qdml root="element">

          <bean label="element" wild="true" type="BIMODAL">
                <attributes>
                      <item coin="attribute" optional="true" repeating="true"/>
                </attributes>
                <elements>
                      <item coin="element" optional="true" repeating="true"/>
                </elements>
          </bean>
          <text label="attribute" wild="true"/>

    </qdml>
```

The above can represent any document. (Note that this is the only time an attribute can be defined as repeating.)

Links

Link elements are used to define document hyperlinks.

```
    <qdml root="x">

          <bean tag="x">
                <elements>
                      <item coin="docLnk"/>
                </elements>
          </bean>
          <link label="docLnk" coin="element"
    schema="http://www.jxml.com/QDMLs/dom.qdml"/>

          <bean label="element" wild="true" type="BIMODAL">
                <attributes>
                      <item coin="attribute" optional="true" repeating="true"/>
                </attributes>
                <elements>
                      <item coin="element" optional="true" repeating="true"/>
```

```
                </elements>
            </bean>
            <text label="attribute" wild="true"/>

        </qdml>
```

The coin attribute on a link defines the kind of element being referenced (in this case, any). The schema attribute names a QDML file describing the kind of document being referenced. Here's a valid document:

```
<x href="http://www.jxml.com/sample.doc"/>
```

top

A comparison of Quick and JAXB

I. Similarities between Quick and JAXB

1. Both will handle XML that can be defined using DTDs.
2. Both transform Java objects into XML and XML into Java Objects.
3. Both can generate marshalling/unmarshalling logic.
4. Both can generate Java classes to hold the data defined by a DTD.

II. Advantages of JAXB

1. JAXB is an integrated architecture, giving it an advantage of speed. It is faster than a SAX parser, since the parsing logic is specific to a particular DTD.
2. JAXB is light-weight and easy to learn, providing a simple correspondence between a DTD and a set of generated Java classes.

III. Disadvantages of JAXB

1. Generated classes contain the marshalling and unmarshalling logic, adding to their apparent complexity.
2. Classes must be generated, with application logic inserted into the generated class or extending the generated class. There is no provision for pre-existing classes.
3. Applications which work with multiple DTDs must move data field-by-field, as each DTD produces its own set of Java classes--the classes can not be reused.

IV. Advantages of Quick

1. All marshalling logic for a given DTD is placed in a single class, with all other generated classes containing only the data items corresponding to the various elements and attributes of the XML. This reduces the apparent complexity, as programmers have no need to become familiar with the marshalling/unmarshalling logic.
2. There is extensive provision for the use of pre-existing classes. Quick can convert a DTD into its own schema language, QJML, which can then be modified to reference existing classes, fields and methods (bean properties).

   As pre-existing classes can be used, Quick can be used to create test data. JXUnit is a product which exploits this capability.

3. Multiple DTDs are easily supported, as classes can often be shared. A class can have several fields with some used with one DTD and some used with the another. (Careful use of default values is assumed here.)

   Free movement of data between XML messages using different schema is exploited by the QARE product, an Open Source XML portal.

4. Quick is a mature product. The code base has been through 7 complete rewrites, under the names Coins, MDSAX and Quick.

5. Quick includes OCM, a model-view-controller framework for transforming XML into (a) a different DTD, (b) Java classes or (c) HTML. The Quick Utilites all serve as examples of this framework.

6. Quick supports hyperlinks between XML documents, including circular links. When documents are transformed into objects, Quick keeps track of which document each object is associated with. When one document is subsequently updated, objects beloning to other documents are not included. (Quick solves many of the same problems addressed by an ODBM.)

7. Through extensions to its schema language, Quick fully supports the Java inheritance model. Programmers starting with a set of Java classes can write a schema which can then be converted into a DTD. (Inheritance simplifies the maintenance needed for complex DTDs, and using the Java inheritance model makes it relatively easy to learn.)

8. Quick uses a layered architecture and runs on any SAX1 parser.

V. Disadvantages of Quick

1. Quick is a mature product, making it appear rather complex.

2. Quick is slowed by its efforts in tracking which objects belong to each document.

top

Part II: How to Use the Quick Utilities

A few things to get Started

I am working on a Windows 2000 laptop, with JDK1.3 installed--the PATH environment variable already contains a reference to C:\jdk1.3\bin.

First, I create a directory, C:\QuickGuide, to work in and unzip the Quick4.3.0.zip file. This puts Quick in the C:\Quick4 directory.

Now I need to add Quick's BATs directory to PATH, define the CLASSPATH variable and set the QuickJARs variable to point to Quick's JARs directory:

```
set PATH=%PATH%;C:\Quick4\BATs
set CLASSPATH=.
set QuickJARs=C:\Quick4\JARs
```

That's really all I need to do to install Quick and create a development environment.

top

Converting a DTD into a QDML Schema (cfgDtd2Qdml and cfgQdml2Dtd)

DTDs are a very concise way to describe an XML document. And since a lot of folk are familiar with them, they are a good place to start.

Example 1

Lets say I have a file, test1.dtd, which holds this very simple DTD:

```
<!ELEMENT a EMPTY>
```

Having installed Quick as described in the previous section, you can convert this DTD into a QDML schema using a simple command:

```
cfgDtd2Qdml -in test1.dtd -out test1.qdml
```

We have just created the test1.qdml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<qdml>
    <bean tag="a"/>
</qdml>
```

To see the QDML file converted back to a DTD, we use this command:

```
cfgQdml2Dtd -in test1.qdml
```

And, as expected, the following DTD is displayed:

```
<!ELEMENT a EMPTY>
```

(Many of the Quick utilities follow the convention of printing the output when no **out** parameter is given.)

Example 2

Before moving on, lets look at a few more examples. Here's the contents of file test2.dtd, which exercises some of the options available with attributes:

```
<!ELEMENT b EMPTY>
<!ATTLIST b
    p CDATA #IMPLIED
    q CDATA #REQUIRED
    r (1 | 2 | 3) #IMPLIED
    s FIXED "23"
    t CDATA "12"
>
```

Again we ran cfgDtd2Qdml, this time creating file test2.qdml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<qdml>
    <bean tag="b">
        <attributes>
            <item coin="b.p" optional="true"/>
            <item coin="b.q"/>
            <item coin="b.r" optional="true"/>
```

```
                <item coin="b.s" fixed="true" optional="true" value="23"/>
                <item coin="b.t" optional="true" value="12"/>
            </attributes>
        </bean>
        <text label="b.p" tag="p"/>
        <text label="b.q" tag="q"/>
        <text label="b.r" tag="r">
            <enum value="1"/>
            <enum value="2"/>
            <enum value="3"/>
        </text>
        <text label="b.s" tag="s"/>
        <text label="b.t" tag="t"/>
    </qdml>
```

One thing you might notice here is that the conversion utility tries to generate unique labels by prefixing attribute names with their element name. (Attribute names are not required to be unique within a DTD, while element names must be unique.)

At this point we should take care to clean up the QDML file--generated code can be a good starting point, but it can often be improved significantly! We can remove the element qualifiers from the attribute names, which means we can also eliminate the use of labels. We can also add some type information. And since we are editing the file, we should add a DOCTYPE statement. Here's what we end up with:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
<qdml>
    <bean tag="b">
        <attributes>
            <item coin="p" optional="true"/>
            <item coin="q"/>
            <item coin="r" optional="true"/>
            <item coin="s" fixed="true" optional="true" value="23"/>
            <item coin="t" optional="true" value="12"/>
        </attributes>
    </bean>
    <text tag="p"/>
    <text tag="q"/>
    <text tag="r" type="int">
        <enum value="1"/>
        <enum value="2"/>
        <enum value="3"/>
    </text>
    <text tag="s" type="int"/>
    <text tag="t" type="int"/>
</qdml>
```

Example 3

Finally, lets look at an example dealing with nested elements. Here's the contents of file test3.dtd, which includes Id/IDREF and an element with both text content and attributes:

```
<!ELEMENT c (d|e|f)*>
<!ATTLIST c
    u IDREF #IMPLIED>

<!ELEMENT d EMPTY>
<!ATTLIST d
    v CDATA #IMPLIED>
```

```
<!ELEMENT e (#PCDATA)>

<!ELEMENT f (#PCDATA)>
<!ATTLIST f
    w ID #IMPLIED>
```

Once again we ran cfgDtd2Qdml, this time creating file test3.qdml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<qdml>
    <bean tag="c">
        <attributes>
            <item coin="c.u"/>
        </attributes>
        <elements>
            <selection optional="true" repeating="true">
                <item coin="d"/>
                <item coin="e"/>
                <item coin="f"/>
            </selection>
        </elements>
    </bean>
    <text label="c.u" tag="u" type="idref"/>
    <bean tag="d">
        <attributes>
            <item coin="d.v" optional="true"/>
        </attributes>
    </bean>
    <text label="d.v" tag="v"/>
    <text tag="e"/>
    <bean tag="f" type="BIMODAL">
        <attributes>
            <item coin="f.w" optional="true">
                <id/>
            </item>
        </attributes>
    </bean>
    <text label="f.w" tag="w"/>
</qdml>
```

There are only a few changes we can make to clean up this file:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
<qdml>
    <bean tag="c">
        <attributes>
            <item coin="u"/>
        </attributes>
        <elements>
            <selection optional="true" repeating="true">
                <item coin="d"/>
                <item coin="e"/>
                <item coin="f"/>
            </selection>
        </elements>
    </bean>
    <text tag="u" type="idref"/>
    <bean tag="d">
```

```
                <attributes>
                    <item coin="v" optional="true"/>
                </attributes>
            </bean>
            <text tag="v"/>
            <text tag="e"/>
            <bean tag="f" type="BIMODAL">
                <attributes>
                    <item coin="w" optional="true">
                        <id/>
                    </item>
                </attributes>
            </bean>
            <text tag="w"/>
        </qdml>
```

top

Setting the QDML Root (cfgSetQdmlRoot)

The root attribute in a QDML schema specifies the outer-most (or top-most) element in the type of document being defined. This is an optional attribute and does not corrispond to anything in a DTD. However, it must be specified before a QDML schema can be converted in a QJML schema. This can be done either by editing the QDML file or by using the cfgSetQdmlRoot utility.

The following command adds a root attribute to the qdml element in file test1.dtd, and creates file test1a.dtd:

```
cfgSetQdmlRoot -in test1.qdml -out test1a.qdml -root a
```

Here is the contents of file test1a.qdml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<qdml root="a">
    <bean tag="a"/>
</qdml>
```

top

Converting QDML to QJML (cfgQdml2Qjml and cfgQjml2Qdml)

While a QDML file is a schema which describes a type of XML document, a QJML file is a binding schema. A QJML file binds the elements and attributes of XML to Java classes, fields and properties. QJML binding schema are at the heart of Quick.

Example 1

Converting a QDML file to QJML is easily done:

```
cfgQdml2Qjml -in test1a.qdml -package quickGuide -out test1.qjml
```

Here is the contents of file test1.qjml:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
```

```
<qjml root="a">
    <bean tag="a">
        <targetClass>quickGuide.A</targetClass>
    </bean>
</qjml>
```

The test1.qjml file binds the XML element a to the class quickGuide.A. Converting this QJML file back to QDML is just as easy:

```
cfgQjml2Qdml -in test1.qjml
```

Here is the output from the cfgQjml2Qdml utility:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qdml SYSTEM "classpath:///qdml.dtd">
<qdml root="a">
    <bean tag="a"/>
</qdml>
```

Example 2

Here is the QJML file generated for the second example:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
<qjml root="b">
    <bean tag="b">
        <targetClass>quickGuide.B</targetClass>
        <attributes>
            <item coin="p" optional="True">
                <property name="p_1"/>
            </item>
            <item coin="q">
                <property name="q_2"/>
            </item>
            <item coin="r" optional="True">
                <property name="r_3"/>
            </item>
            <item coin="s" fixed="True" optional="True" value="23">
                <property name="s_4" initializer="23"/>
            </item>
            <item coin="t" optional="True" value="12">
                <property name="t_5" initializer="12"/>
            </item>
        </attributes>
    </bean>
    <text tag="p"/>
    <text tag="q"/>
    <text tag="r" type="int">
        <enum value="1"/>
        <enum value="2"/>
        <enum value="3"/>
    </text>
    <text tag="s" type="int"/>
    <text tag="t" type="int"/>
</qjml>
```

The property names are already unique and do not need the added postfix. Here then is the revised QJML file:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
<qjml root="b">
    <bean tag="b">
        <targetClass>quickGuide.B</targetClass>
        <attributes>
            <item coin="p" optional="True">
                <property name="p"/>
            </item>
            <item coin="q">
                <property name="q"/>
            </item>
            <item coin="r" optional="True">
                <property name="r"/>
            </item>
            <item coin="s" fixed="True" optional="True" value="23">
                <property name="s" initializer="23"/>
            </item>
            <item coin="t" optional="True" value="12">
                <property name="t" initializer="12"/>
            </item>
        </attributes>
    </bean>
    <text tag="p"/>
    <text tag="q"/>
    <text tag="r" type="int">
        <enum value="1"/>
        <enum value="2"/>
        <enum value="3"/>
    </text>
    <text tag="s" type="int"/>
    <text tag="t" type="int"/>
</qjml>
```

Example 3

And finally, here is the QJML file generated for the third example:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
<qjml root="c">
    <bean tag="c">
        <targetClass>quickGuide.C</targetClass>
        <attributes>
            <item coin="u">
                <property name="u_1"/>
            </item>
        </attributes>
        <elements>
            <selection optional="True" repeating="True">
                <property coin="c_2" kind="list" name="c_2"/>
                <item coin="d"/>
                <item coin="e"/>
                <item coin="f"/>
            </selection>
        </elements>
    </bean>
    <abstract generate="False" label="c_2" validInherited="True">
        <targetClass>java.lang.Object</targetClass>
```

```
        </abstract>
        <text tag="u" type="idref"/>
        <bean tag="d">
            <targetClass>quickGuide.D</targetClass>
            <attributes>
                <item coin="v" optional="True">
                    <property name="v_3"/>
                </item>
            </attributes>
        </bean>
        <text tag="v"/>
        <text tag="e"/>
        <bean tag="f" type="BIMODAL">
            <targetClass>quickGuide.F</targetClass>
            <attributes>
                <item coin="w" optional="True">
                    <id/>
                </item>
            </attributes>
        </bean>
        <text tag="w"/>
    </qjml>
```

There are a number of changes we can make:

- Add coin="f" to the definition of the field for attribute u. This indicates that the IDREF will always be a reference to an f element. (Now when we generate the Java code, it will use a variable of type F.)
- Change property name c_2 to list, which is more descriptive.
- Drop the property name qualifiers.
- The abstract coin c_2 can be dropped, as it adds no value in this case.
- Replace all properties with fields, so we can see how Quick works with variables.

This then is the final QJML file:

```
    <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
    <!DOCTYPE qjml SYSTEM "classpath:///qjml.dtd">
    <qjml root="c">
        <bean tag="c">
            <targetClass>quickGuide.C</targetClass>
            <attributes>
                <item coin="u">
                    <field name="u" coin="f"/>
                </item>
            </attributes>
            <elements>
                <selection optional="True" repeating="True">
                    <field kind="list" name="list"/>
                    <item coin="d"/>
                    <item coin="e"/>
                    <item coin="f"/>
                </selection>
            </elements>
        </bean>
        <text tag="u" type="idref"/>
        <bean tag="d">
            <targetClass>quickGuide.D</targetClass>
            <attributes>
                <item coin="v" optional="True">
                    <field name="v"/>
```

```
            </item>
        </attributes>
    </bean>
    <text tag="v"/>
    <text tag="e"/>
    <bean tag="f" type="BIMODAL">
        <targetClass>quickGuide.F</targetClass>
        <attributes>
            <item coin="w" optional="True">
                <id/>
            </item>
        </attributes>
    </bean>
    <text tag="w"/>
</qjml>
```

top

Generating Java from QJML (cfgQjml2Java2)

When first starting, it is a good idea to generate code from a QJML file, to get an idea of the kind of code that the QJML can bind to. For while Quick is designed to bind XML to pre-existing code, there are a number of constraints placed on that code.

Example 1

The following command generates the Java code for the data model (not the marshalling/unmarshalling code) and places the source directory tree in the current working directory:

```
cfgQjml2Java2 -in test1.qjml
```

The above command generates a singel file, quickGuide\A.java:

```
package quickGuide;
import com.jxml.quick.*;

public class A
{
}
```

The instances of the generated class, quickGuide.A, can be used to represent XML elements of type a. One important thing to note here is that Quick generally requires a null constructor. (A null constructor is a constructor with no parameters.) But when no other constructors are present, the Java compiler always produces one by default.

Example 2

The second example deals with a variety of attributes, which are bound by the QJML file to various JavaBean properties. Generating the Java data model code from the updated test2.qjml file adds B.java to the quickGuide source directory:

```
package quickGuide;
import com.jxml.quick.*;

public  class B
{
        public String _p;
```

```
            public String _q;
            public int _r;
            public int _s=23;
            public int _t=12;

            public String getP()
            {
                    return _p;
            }

            public void setP(String __v)
            {
                    _p=__v;
            }

            public String getQ()
            {
                    return _q;
            }

            public void setQ(String __v)
            {
                    _q=__v;
            }

            public int getR()
            {
                    return _r;
            }

            public void setR(int __v)
            {
                    _r=__v;
            }

            public int getS()
            {
                    return _s;
            }

            public void setS(int __v)
            {
                    _s=__v;
            }

            public int getT()
            {
                    return _t;
            }

            public void setT(int __v)
            {
                    _t=__v;
            }
    }
```

The instances of the generated class, quickGuide.B, can be used to represent XML elements of type b. If you like working with JavaBean properties, you may quite enjoy this code generator, as it generates much of the code that you need for you. (In this case, it generated code for properties q, r, s, and t.)

Example 3

The third example deals with a list of elements contained by the top-level element, and uses fields rather than properties. Generating the Java data model code from the updated test3.qjml file adds three more files to the quickGuide source directory: C.java, D.java and F.java.

C.java

```
package quickGuide;
import com.jxml.quick.*;
import java.util.*;

public  class C
{
        public F  u;
        public ArrayList  list=new ArrayList();
}
```

The instances of the generated class, quickGuide.C, can be used to represent XML elements of type c.

D.java

```
package quickGuide;
import com.jxml.quick.*;

public  class D
{
        public String  v;
}
```

The instances of the generated class, quickGuide.D, can be used to represent XML elements of type d. The attribute v is bound to the public variable v.

F.java

```
package quickGuide;
import com.jxml.quick.*;

public class F  implements  QBiModal
{
        public String __text="";

        public String getQText()
        {
                return __text;
        }

        public void setQText(String __v)
        {
                __text=__v;
        }
}
```

The instances of the generated class, quickGuide.F, can be used to represent XML elements of type f. The attribute w is an XML ID and is not bound to anything in the Java object.

Class quickGuide.F is an example of a special case, BIMODAL, which is used when an element has attributes and also text content. The class must always implement interface QBiModal.

top

Generating QIML and schema factory classes (cfgQjml2Qiml and cfgQiml2Java)

QIML files are the internal binding schemas used by quick. A QIML file is a direct representation of the data structures used to define a transformation engine capable of converting (bi-directionally) XML and Java Objects. QIML files tend to be more than twice as long as a QJML file, and somewhat difficult to read.

The following command converts the QJML file for example 1 into a QIML file:

```
cfgQjml2Qiml -in test1.qjml -out test1.qiml
```

Here is the contents of file test1.qiml:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qiml SYSTEM "classpath:///qiml.dtd">
<qiml>
    <elementDefinition ID="a" tagName="a">
        <targetClass>quickGuide.A</targetClass>
        <childElement>
            <accessClass>com.jxml.quick.access.QMALAccess</accessClass>
            <elementDefinition tagName="_attributes">
                <targetFactory>com.jxml.quick.tf.QMALTF</targetFactory>
            </elementDefinition>
        </childElement>
    </elementDefinition>
</qiml>
```

QIML files can be processed by the Quick runtime API, but since the files tend to be rather large, processing them at runtime adds significantly to the runtime overhead. Instead, it is much better to convert a QIML file into the Java code which assemblies the schema data structures directly. Once this code has been compiled, the resulting bytecode file is significantly smaller and runs very quickly.

The following command converts the QIML file for example 1 into a SchemaFactory class:

```
cfgQiml2Java -in test1.qiml -out quickGuide\Schema1.java -class quickGuide.Schema1 -
key classpath:///test1.qiml
```

Here is the contents of file Schema1.java:

```
// generated from test1.qiml
package quickGuide;
import com.jxml.quick.*;
import com.jxml.quick.engine.*;
import com.jxml.quick.recycle.*;
import com.jxml.quick.access.*;
import com.jxml.quick.qmap.*;
import com.jxml.quick.tf.*;
import org.xml.sax.*;

public class Schema1 extends QSchemaFactory
{
        protected static QDoc schema=null;

        public static QDoc createSchema()
                throws SAXException, QE, QPE, ClassNotFoundException,
                        InstantiationException, IllegalAccessException
        {
```

```
		if (schema!=null)
			return schema;
		return (new Schema1()).create();
	}

	public QDoc create()
		throws SAXException, QE, QPE, ClassNotFoundException,
			InstantiationException, IllegalAccessException
	{
		if (schema!=null)
			return schema;
		QSoftDocHash sdh=Quick.schemaHash;
		QDoc rv=sdh.get("classpath:///test1.qiml");
		if (rv!=null)
		{
			schema=rv;
			return schema;
		}
		synchronized(this.getClass())
		{
			if (schema!=null)
				return schema;
			rv=build();
			schema=sdh.add(rv);
			if (schema!=rv)
				return schema;
			QRefHash.add(schema);
			return schema;
		}
	}

	protected QDoc build()
		throws SAXException, QE, QPE, ClassNotFoundException,
			InstantiationException, IllegalAccessException
	{
		QDocImpl doc=new QDocImpl();
		QIML rootCE=new QIML();


		doc.setRoot(rootCE);
		QElementFactory var0=new QElementFactory();
		doc.setId("a",var0);
		var0.tagName="a";
		QClassWrapper var1=new QClassWrapper();
		var0.targetFactory=var1;
		var1.wClassName="quickGuide.A";
		QCElementFactory var2=new QCElementFactory();
		doc.setId("genid_1021492478306",var2);
		var2.access=new com.jxml.quick.access.QMALAccess();
		QElementFactory var3=new QElementFactory();
		doc.setId("genid_1021492478307",var3);
		var3.tagName="_attributes";
		var3.targetFactory=new com.jxml.quick.tf.QMALTF();
		var3.addToFactory(var2);
		var2.addToFactory(var0);

		var0.addToFactory(rootCE);
		doc.setSchema(Quick.QIMLSchema());
		doc.setKey("classpath:///test1.qiml");
		doc.pool=new QContextPool(doc);
		return doc;
```

```
                }
        }
```

top

Part III: How to Use the Quick API

Using the Quick Runtime

Again, I am working on a Windows 2000 laptop, with JDK1.3 installed--the PATH environment variable already contains a reference to C:\jdk1.3\bin. Quick is in the C:\Quick4 directory. And I am working in the C:\QuickGuide directory.

We will need to add two JAR files from Quick's JARs directory:

```
set CLASSPATH=C:\Quick4\JARs\crimson.jar;C:\Quick4\JARs\Quick4rt.jar
```

To complete our development environment, lets also add the root directory of the class directory tree to our classpath:

```
set CLASSPATH=%CLASSPATH%;C:\QuickGuide
```

Now we can compile the Java code we generated in Part II (above):

```
cd C:\QuickGuide\quickGuide
javac *.java
```

top

Expressing Objects in XML

The binding schema that we developed in Part II can be used to express the Java objects bound to those schema as XML files.

Example 1

Expressing an object in XML, once we have a QJML file and have used the Quick utilities to convert it into a schema factory class, is really very simple:

1. Create the schema object.
2. Wrap the object to be expressed in a QDoc object together with the schema object.
3. Express the XML.

Here's the code for a program which creates an object, a, and then expresses it in XML:

```
import quickGuide.*;
import com.jxml.quick.*;

public class Express1
{
        public static void main(String args[])
                throws Exception
        {
                A a=new A();
```

```
                QDoc schema1=Schema1.createSchema();
                QDoc aDoc=Quick.createDoc(a,schema1);
                String xml=Quick.express(aDoc);
                System.out.println(xml);
        }
}
```

And when you run it, this is your output:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<a/>
```

Example 2

This time we are going to create an object that has some data and express it to a file:

```
import quickGuide.*;
import com.jxml.quick.*;

public class Express2
{
        public static void main(String args[])
                throws Exception
        {
                B b=new B();
                b.setP("Easy Street");
                b.setQ("How's this?");
                b.setR(2);
                QDoc schema2=Schema2.createSchema();
                QDoc bDoc=Quick.createDoc(b,schema2);
                Quick.express(bDoc,"b.xml");
        }
}
```

Here's the output put in file b.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<b p="Easy Street" q="How&apos;s this? r="2"/>
```

Note that Quick always alphabitizes parameters when it expresses them. You can also see from this example that special characters are encoded by Quick when necessary to produce valid XML.

Sometimes you want to express attributes even when they are set to their default value. Here's some sample code that does that:

```
import quickGuide.*;
import com.jxml.quick.*;

public class Express2All
{
        public static void main(String args[])
                throws Exception
        {
                B b=new B();
                b.setQ("kitchen sink");
                b.setR(1);
                QDoc schema2=Schema2.createSchema();
                QDoc bDoc=Quick.createDoc(b,schema2);
                Quick.express(bDoc,"b-all.xml",true);
```

```
        }
}
```

Here's the output put in file b-all.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<b q="kitchen sink" r="1" t="12"/>
```

In this case, t is expressed with its default value. Attribute s, which is both fixed and optional, is still not expressed.

Example 3

In this example we construct a graph of several objects and then express them as XML. Here's the code:

```
import quickGuide.*;
import com.jxml.quick.*;

public class Express3
{
        public static void main(String args[])
                throws Exception
        {
                C c=new C();
                F f=new F();
                f.setQText("an F");
                c.list.add(f);
                c.u=f;
                c.list.add("first e");
                c.list.add("second e");
                QDoc schema3=Schema3.createSchema();
                QDoc cDoc=Quick.createDoc(c,schema3);
                Quick.express(cDoc,"c.xml");
        }
}
```

Here's the output put in file c.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<c>
    <f w="genid_1022595499291">an F</f>
    <e>first e</e>
    <e>second e</e>
</c>
```

top

Building Objects from XML

The binding schema that we developed in Part II can also be used to build the Java objects bound to those schema from XML files.

Example 1

Lets start by looking at an XML file, a.xml, that we want to convert to an object:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE qdml SYSTEM "classpath:///test1.dtd">
<a/>
```

Building an object from this file, once we have a QJML file and have used the Quick utilities to convert it into a schema factory class, is really very simple:

1. Enable the classpath protocol. (This protocol is handy, as it allows you to access DTDs and other files via your CLASSPATH.)
2. Create the schema object.
3. Use the schema object to parse the XML file, creating a document object.
4. Extract the application object from the document wrapper object.

Here's the code for a program which creates an object, a, from file a.xml:

```
import quickGuide.*;
import com.jxml.quick.*;
import com.jxml.protocol.*;

public class Build1
{
        public static void main(String args[])
                throws Exception
        {
                Protocol.addJXMLProtocolPackage();
                QDoc schema1=Schema1.createSchema();
                QDoc aDoc=Quick.parse(schema1,"file:a.xml");
                A a=(A)aDoc.getRoot();
                System.out.println(a);
        }
}
```

Example 2

This time we are going to convert a String holding an XML document into an object:

```
import quickGuide.*;
import com.jxml.quick.*;

public class Build2
{
        public static void main(String args[])
                throws Exception
        {
                QDoc schema2=Schema2.createSchema();
                String xml="<b q='What fun!'/>";
                QDoc bDoc=Quick.parseString(schema2,xml);
                B b=(B)bDoc.getRoot();
                System.out.println(b.getQ());
        }
}
```

Example 3

Finally, we are going to read an XML document (file c.xml, produced in the previsous section), update the object graph and then rewrite the file. Here's the code:

```
import quickGuide.*;
```

```
import com.jxml.quick.*;
import com.jxml.protocol.*;

public class Build3
{
        public static void main(String args[])
                throws Exception
        {
                Protocol.addJXMLProtocolPackage();

                String fileName="c.xml";
                String url="file:"+fileName;

                QDoc schema3=Schema3.createSchema();
                QDoc cDoc=Quick.parse(schema3,url);

                C c=(C)cDoc.getRoot();
                D d=new D();
                d.v="A real giggle!";
                c.addD(d);

                Quick.express(cDoc,fileName);
        }
}
```

And here's the updated file, c.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<c>
    <f w="genid_1022595499291">an F</f>
    <e>first e</e>
    <e>second e</e>
    <d v="A real giggle!"/>
</c>
```

top

Part IV: Config

A Script for Validating Parameters

Programs run from a command line always need to begin by validating their parameters. Servlets too must validate the parameters passed to them. Usually you just write the code you need to do this, but perhaps a better approach is to use an XML document to describe the process.

There are a numpber of advantages to using a document to describe parameter validation in place of just writing the code:

- Unlike code, a document can be validated against a schema, giving a better assurance of correctness.
- The document may be easier to maintain than the code.
- A document can be used to generate the documentation.
- A document can be used to generate an html form.
- A document can be used to generate a graph of objects to validate the parameters at run time, or to generate the code to do the validation.

The config markup language is part of Quick. It is used by all of the Quick Utilities to validate their parameters. It is also used by QARE, an XML Portal that builds on Quick, to validate all the various user requests passed by via an

HTML GET.

top

Scripts--A Potential Security Risk

Encoding scripts in XML may simplify the valiation process, but validation is not always sufficient to prevent malicious behavior. This is true of config files, as they include references to other classes which may not be appropriate.

A script document, while it can be processed as data, should not generally be passed as content, but should be treated as code. QARE, for example, which relies heavily on config scripts, always retrieves them from the application JAR files--they are never passed as part of a user request.

top

Config Document Examples

Here are two sample config files, taken from the Quick Utilities:

setRoot.config

This config file is used by the **cfgSetQdmlRoot** utility:

```
<cfg>
        <RequiredItem name="in"/>
        <RequiredItem name="root"/>
        <NVItem name="out"/>
        <eval class="com.jxml.quick.util.qdmlRoot.SetRoot"/>
</cfg>
```

The config file first checks for the two required parameters, in and root, and processes the optional parameter, out. Then it invokes the eval method on class SetRoot. No other parameters are allowed. Here's a sample invocation of the cfgSetQdmlRoot utility:

```
cfgSetQdmlRoot -in test.qdml -root fudge
```

qjml2Java.config

This config file is used by the old **cfgQjml2Java** utility:

```
<cfg>
        <NVItem name="sinkSchemaClass"
                value="com.jxml.quick.util.CreateUtil"
                fixed="true"/>
        <NVItem name="sinkIn"
                value="classpath:///com/jxml/quick/util/qjml2java/preferences.util"
                fixed="true"/>
        <NVItem name="out"/>
        <NVItem name="printClone" value="false"/>
        <NVItem name="printQAware" value="false"/>
        <NVItem name="setParents" value="false"/>
        <NVItem name="printPrint" value="false"/>
        <NVItem name="printFullConst" value="true"/>
        <NVItem name="printMoreMethods" value="true"/>
```

```
            <NVItem name="oneNewClass" value="true"/>
            <NVItem name="genPackage"/>
            <RequiredItem name="in"/>
            <eval class="com.jxml.quick.ocm.OCMsinkSupport"/>
    </cfg>
```

This config file will process:

- two fixed parameters, sinkSchemaClass and sinkIn,
- the optional out parameter,
- several optional parameters with default values,
- the optional genPackage parameter and
- the required in parameter,
- and then invokes the OCMsinkSupport.eval method.

And here's a sample invocation of the old cfgQjml2Java utility:

```
cfgQjml2Java -in test.qjml -printPrint true
```

top

Config Binding Schema

There is a lot we can learn from looking closely at a QJML file:

- The grammer of the XML document,
- The Java classes bound to the XML elements and attributes,
- The variables and properties of the Java classes used to hold the data content and
- The Java class hierarchy.

So here is the complete QJML file for config:

```
<qjml root="cfg" SYSTEM="classpath:///config.dtd">

        <interface label="Item">
                <rem>All content of cfg must implement this interface.</rem>
                <targetClass>com.jxml.quick.config.Item</targetClass>
        </interface>

        <bean tag="cfg">
                <rem>Root element of all config documents.</rem>
                <implements>Item</implements>
                <targetClass>com.jxml.quick.config.ConfigImpl</targetClass>
                <elements>
                        <item coin="htmlHead" optional="true">
                                <field name="htmlHead"/>
                        </item>
                        <item coin="htmlTitle" optional="true">
                                <field name="htmlTitle"/>
                        </item>
                        <item coin="instruction" optional="true" repeating="true">
                                <field name="instruction" kind="list"/>
                        </item>
                        <item coin="Item" optional="true" repeating="true">
                                <field name="items" kind="list"/>
                        </item>
                </elements>
```

```
        </bean>

        <text tag="htmlHead">
                <rem>Form heading on the web page</rem>
        </text>

        <text tag="htmlTitle">
                <rem>Title of the web page</rem>
        </text>

        <text tag="instruction">
                <rem>Instructions listed on the web page</rem>
                <targetClass>com.jxml.quick.config.Instruction</targetClass>
        </text>

        <bean tag="NVItem">
                <rem>Defines a default value for a given name</rem>
                <rem>and extracts an overriding value from the command-line
arguments</rem>
                <implements>Item</implements>
                <targetClass>com.jxml.quick.config.NVItem</targetClass>
                <attributes>
                        <item coin="name">
                                <field name="name"/>
                        </item>
                        <item coin="value" optional="true" >
                                <field name="value"/>
                        </item>
                        <item coin="fixed" optional="true" value="false">
                                <field name="fixed"/>
                        </item>
                        <item coin="fieldLength" optional="true" value="24">
                                <field name="fieldLength"/>
                        </item>
                </attributes>
        </bean>

        <text tag="name">
                <rem>The name of a parameter.</rem>
        </text>

        <text tag="value">
                <rem>The default value of a parameter.</rem>
        </text>

        <text tag="fixed" type="boolean">
                <rem>The value is a constant, it can not be changed.</rem>
        </text>

        <text tag="fieldLength">
                <rem>Specifies the field length on the form input</rem>
        </text>

        <bean tag="RequiredItem">
                <rem>Identifies a mandatory command-line argument.</rem>
                <implements>Item</implements>
                <targetClass>com.jxml.quick.config.RequiredItem</targetClass>
                <attributes>
                        <item coin="name">
                                <field name="name"/>
                        </item>
                        <item coin="fieldLength" optional="true" value="24">
```

```
                                    <field name="fieldLength"/>
                            </item>
                    </attributes>
            </bean>

            <bean tag="dump">
                    <rem>Displays the names and values of all parameters.</rem>
                    <implements>Item</implements>
                    <targetClass>com.jxml.quick.config.Dump</targetClass>
            </bean>

            <bean tag="eval">
                    <rem>Instantiates an Item and calls the eval method.</rem>
                    <implements>Item</implements>
                    <targetClass>com.jxml.quick.config.Eval</targetClass>
                    <attributes>
                            <item coin="class">
                                    <field name="evalClass"/>
                            </item>
                    </attributes>
            </bean>

            <text tag="class">
                    <rem>The full name of a class which implements Item.</rem>
            </text>
    </qjml>
```

The elements htmlHead, htmlTitle and instructions and the attribute fieldLength are not part of the normal process, but were included for use by a cfg2html utility for generating HTML forms.

top

The Item Interface

The Item interface is implemented by nearly all the config classes. It has a single method, eval:

```
package com.jxml.quick.config;
import java.util.*;

public interface Item
{
        public void eval(Map properties, List args, ClassLoader cl)
                throws Exception;
}
```

At the start of a config script, the properties map is empty, or contains only pre-defined name/value pairs, and the args list contains all the unprocessed parameters.

As the script is executed, parameters are removed from the args list and add to the properties map.

The CloassLoader parameter, cl, will normally be null except when, as is the case with QARE, multiple JAR class loaders are used.

top

A Generic Main Method

When we run a utility like cfgSetQdmlRoot, we enter a command like this:

```
cfgSetQdmlRoot -in test.qdml -root fudge
```

But when we dig through the command scripts, we find that we are really running this:

```
java com.jxml.quick.config.Main
classpath:///com/jxml/quick/util/qdmlRoot/setRoot.config -in test.qdml -root fudge
```

By using config, all of the Quick Utilities can use a common main method:

```
package com.jxml.quick.config;
import com.jxml.quick.*;
import com.jxml.protocol.*;
import org.xml.sax.*;
import java.io.*;
import java.util.*;

public class Main
{
        public static void main(String[] arg)
        {
                try
                {
                        Protocol.addJXMLProtocolPackage();
                        ArrayList args=new ArrayList(Arrays.asList(arg));

                        int sargs=args.size();
                        if (sargs<1)
                        {
                                System.out.println("At least one arguments are always
required:");
                                System.out.println("    - The name of the config
file!");
                                return;
                        }
                        QDoc schema=CreateConfig.createSchema();
                        QDoc doc=Quick.parse(schema,arg[0]);
                        args.remove(0);
                        Item item=(Item)doc.getRoot();

                        TreeMap properties=new TreeMap();
                        StringBuffer sb=new StringBuffer();
                        properties.put("out",sb);
                        item.eval(properties,args,null);
                        System.out.print(sb.toString());
                }
                catch (Exception pe)
                {
                        QPE.display(pe);
                }
        }
}
```

Main builds a graph of objects from the config file named by the first command-line argument using the CreateConfig schema factory. (This schema factory was, of course, created from the config QJML file using the cfgQjml2Qiml and cfgQiml2Java utilities.)

After creating the object graph, main calls the eval method on the root of the graph, passing the remaining arguments

for processing.

The properties map is pre-loaded with a StringBuffer object named out. If there is no out parameter in the command-line argument list, this StringBuffer is used for output. Its contents are printed after the config script has been processed.

top

Processing Scripts

The config QJML file binds the root element of a config file, cfg, to the class com.jxml.quick.config.ConfigImpl. So when Main calls the eval method on the root object of the object graph built from a config file, it is always calling ConfigImpl.eval:

```
package com.jxml.quick.config;
import java.util.*;

public class ConfigImpl implements Item
{
        public ArrayList items=new ArrayList();
        public String htmlTitle="";
        public String htmlHead="";
        public ArrayList instruction=new ArrayList();

        public void eval(Map properties, List args, ClassLoader cl)
                throws Exception
        {
                int i,s;
                s=items.size();
                for (i=0;i<s;++i)
                {
                        Item item=(Item)items.get(i);
                        item.eval(properties,args,cl);
                }
        }
}
```

Processing the config script couldn't be simpler. The script has already been converted into a graph of objects and assembled as a list under root object, ConfigImpl. It is just a matter of stepping through the list and invoking eval on each item.

top

Invoking the Application

The config QJML file binds the eval element to the class com.jxml.quick.config.Eval. Eval is always the last step in a config script and is responsible for instantiating an application object and then calling the eval method on that object. Here's the code:

```
package com.jxml.quick.config;
import java.util.*;

public class Eval implements Item
{
        public String evalClass;
```

```
        public void eval(Map properties, List args, ClassLoader cl)
                throws Exception
        {
                if(args.size()>1) throw new Exception("The list items should be
empty");
                Class c=null;
                if (cl==null)
                        c=Class.forName(evalClass);
                else
                        c=cl.loadClass(evalClass);
                Object o=c.newInstance();
                Item i=(Item)o;
                i.eval(properties,args,cl);
        }
}
```

Before doing anything else, Eval checks the args list to make sure that there are no remaining, unprocessed parameters.

There is one thing worth noting here, and that is the use of the optional ClassLoader parameter, cl, for creating the application's class object. This is important when using a framework like QARE, which supports multiple class loaders.

top

Part V: Transforming XML

Model/View/Controller (MVC)

The Model/View/Controller programming pattern is an important one when performing complex transformations. It is not an easy pattern to understand, but the resulting code is relatively easy to maintain. The basic idea is that the code producing the output (the view) is distinct from the code which models the data being transformed (and which often contains application logic), with a third set of code used to control the process.

The JTree class in Java Swing is a good example using the MVC pattern. There is a clear distinction between the controller logic and the application data model, with peer objects used to support the view. It is one of the more difficult Swing classes to master, but the resulting code is easy to maintain.

Java Server Pages (JSP) is a good example of a system which does not support the MVC pattern. Maintenance of JXP application code is difficult as an application grows in complexity. Indeed, a common complaint about JSP is that it is difficult to use with MVC.

XSL is the tool of choice for transforming XML, though it does not support MVC. Complex transformations, typical when using XML to encode data, are often difficult to code in XSL.

top

The Quick Utilities

The Quick Utilities perform a number of complex transformations:

- DTD -> XML (cfgDdt2Qdml)
- XML -> DTD (cfgQdml2Dtd)
- XML -> XML (cfgQdml2Qjml, cfgQjml2Qdml, cfgQjml2Qiml, cfgQdmlExpand, cfgQjmlExpand, cfgSetid and

cfgNoid)
  - XML -> Java (cfgQjml2Java, cfgQjml2Java2 and cfgQiml2Java)
  - XML -> HTML (cfgQjml2Html)

Maintaing the code for version 3 of the Quick Utilities was very difficult and time consuming. This changed in version 4 with the development of OCM, a framework which supports MVC. Maintenance of the Quick Utilities is now very much easier, freeing development to focus on other projects, despite the increased complexity of the version 4 schema languages and an increase in the number of supported utilites.

In Quick version 4 the Utilities use the OCM and Util markup languages to define these transformations, sometimes as a series of simpler transformations. And when the transformation uses MVC or is a simpler MC pattern, OCM is used to map the relationship between the data model and controller objects. (Util is an extention of OCM, containing many constructs specific to these utilities.)

top

MVC Design Issues

The Model

The model is used to hold the data to be transformed. The classes comprising the model could be a generic Document Object Model (DOM), like JDOM. Or given a QJML file, it could be the simple classes generated by the cfgQjml2Java utility. But except when the model is quite simple, this is not really good enough.

The model should be treated as an API for any application which deals with that type of data. It is a great place to include logic for data extraction and navigation. The QJML model classes are a good case in point, the data is fairly rich and the model includes a number methods that make it easier to work with.

The Controller

OCM works by creating a peer object for many of the objects in a model, where the class of the peer often depends on the kind of model object it is connected to. This is a type of aggrigation--each controller object contains a model object, but the model objects have no knowledge of the controller objects.

Aggrigation is a bit more complicated than using inheritance but has a major advantage here: the code which builds the model is not coupled to the controller logic. The cfgDtd2Qdml utility is a good case in point. This utility uses the DTDParser package to create the data model. Aggrigation facilitates code reuse.

The View

Using a view is helpful for complex transformations, but sometimes it just adds needless complexity. So it is optional.

When working with MVC, it is less confusing when each part has a different form. OCM uses simple text files as output templates for its views, making them quite distinct from the controller logic.

top

Example 1: cfgNoid

The cfgNoid utility performs a simple transformation on any XML file--it removes ID attributes from the elements. Here's a sample set of input and output documents:

Input

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<qjml ID="ID1" root="A">
    <bean ID="ID2" tag="A">
        <targetClass ID="ID3">A</targetClass>
        <attributes ID="ID4">
            <item ID="ID5" coin="A.a" optional="True">
                <property ID="ID6" name="A_a_1"/>
            </item>
        </attributes>
    </bean>
    <text ID="ID7" label="A.a" tag="a" validInherited="True">
        <targetClass ID="ID8">A_a</targetClass>
    </text>
</qjml>
```

Output

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<qjml root="A">
    <bean tag="A">
        <targetClass>A</targetClass>
        <attributes>
            <item coin="A.a" optional="True">
                <property name="A_a_1"/>
            </item>
        </attributes>
    </bean>
    <text label="A.a" tag="a" validInherited="True">
        <targetClass>A_a</targetClass>
    </text>
</qjml>
```

Here's the config file used to process the parameters and invoke the utility:

```
<cfg>
        <NVItem name="sinkSchemaClass"
                value="com.jxml.quick.util.CreateUtil"/>
        <NVItem name="sinkIn"
                value="classpath:///com/jxml/quick/util/noid/counter.util"/>
        <RequiredItem name="in"/>
        <NVItem name="out"/>
        <eval class="com.jxml.quick.ocm.OCMsinkSupport"/>
</cfg>
```

The OCMsinkSupport.eval method invoked by this config file, and by the config files of most of the Quick Utilities, it the entry point for the ocm package. This method uses the for parameters, sinkSchemaClass, sinkIn, in and out. The sinkIn parameter names the configuration file used to construct the transformation; the sinkSchemaClass parameter names the schema class used to parse the configuration file. (The schema must be for OCM or a derived schema like Util.)

Here then is the Util file used to construct the cfgNoid utility:

```
<contextSupport outputSchema="classpath:///OCMdom.qiml">
        <input inputSchema="classpath:///OCMdom.qiml"/>
        <factoryMap>
                <factoryEntry
                        dataModelClass="com.jxml.quick.ocm.OCMelement"
                        peerClass="com.jxml.quick.util.noid.Element"/>
        </factoryMap>
</contextSupport>
```

The cfgNoid utility, like other Quick Utilities which transform XML -> XML, does not employ a view. This utility only has a model and controller. The model is **OCMdom**, which includes the noid **OCMelement** class. The controller is the **Element** class.

The OCMdom Model

The cfgNoid utility uses the OCMdom model to represent almost any kind of XML document. (Elements holding both text content and other elements can not be processed by Quick.) Here's the QJML file for OCMdom:

```
<qjml root="element">
    <bean label="element" type="BIMODAL" wild="true">
        <targetClass>com.jxml.quick.ocm.OCMelement</targetClass>
        <attributes>
            <item coin="attribute" optional="true" repeating="true">
                <accessClass accessor="com.jxml.quick.access.QDOMAttAccess"/>
            </item>
        </attributes>
        <elements>
            <item coin="element" optional="true" repeating="true">
                <property kind="list" name="QElements"/>
            </item>
        </elements>
    </bean>
    <text label="attribute" wild="true">
        <targetClass>com.jxml.quick.ocm.OCMattribute</targetClass>
    </text>
</qjml>
```

This is indeed an unusual QJML file, with its use of wild attributes, an accessClass element and a text element which contains a targetClass. But we need to focus on the classes used to represent elements and attributes:

OCMelement

```
package com.jxml.quick.ocm; // Open Conversion Model
import com.jxml.quick.*;
import org.xml.sax.*;
import java.io.*;
import java.util.*;

/**
 * Generic element model.
 */
public class OCMelement implements QDOMEle
{
        public String name;
        public String text;
        public ArrayList elements=new ArrayList();
        public ArrayList attributes=new ArrayList();
```

```
transient boolean allSet=false;

public void setIDs(QDoc doc)
        throws QPE
{
        if (allSet)
                return;
        allSet=true;

        int i,s;

        s=attributes.size();
        for (i=0;i<s;++i)
        {
                OCMattribute a=(OCMattribute)attributes.get(i);
                if ("ID".equals(a.name))
                {
                        i=s;
                        doc.setId(a.text,this);
                }
        }

        s=elements.size();
        for (i=0;i<s;++i)
        {
                OCMelement e=(OCMelement)elements.get(i);
                e.setIDs(doc);
        }
}

public String getQName()
{
        return name;
}

public void setQName(String name)
{
        this.name=name;
}

public String getQText()
{
        return text;
}

public void setQText(String text)
{
        this.text=text;
}

public List getQElements()
{
        return elements;
}

public List getQAttributes()
{
        return attributes;
}

public void addAttribute(OCMattribute att)
{
```

```
                int i,s;
                s=attributes.size();
                for(i=0;i<s;i++)
                {
                        OCMattribute a=(OCMattribute)attributes.get(i);
                        if(a.equals(att))
                        {
                                attributes.set(i,att);
                                return;
                        }
                }
                attributes.add(att);
        }
}
```

OCMattribute

```
package com.jxml.quick.ocm; // Open Conversion Model
import com.jxml.quick.*;
import org.xml.sax.*;
import java.io.*;

/**
 * Generic attribute model.
 */
public class OCMattribute implements QName
{
        public String name;
        public String text;

        public OCMattribute(String name, String text)
        {
                this.name=name;
                this.text=text;
        }

        public OCMattribute(String text)
        {
                this.text=text;
        }

        public String toString()
        {
                return text;
        }

        public String getQName()
        {
                return name;
        }

        public void setQName(String name)
        {
                this.name=name;
        }

        public boolean equals(Object x)
        {
                if (!(x instanceof OCMattribute))
                        return false;
                OCMattribute a=(OCMattribute)x;
```

```
                return name.equals(a.name);
        }
}
```

Both OCMelement and OCMattribute have a number of unusual features. But we need to pay particular attention here to the public variables in OCMelement and OCMattribute, as these fields are used by the cfgNoid controller.

The cfgNoid Controller

The controller will transverse the tree of the model, generating SAX events. These events are then expressed as an XML document, which is the final output for the cfgNoid utility. Here's the controller logic, Element:

```
package com.jxml.quick.util.noid;
import com.jxml.quick.*;
import org.xml.sax.*;
import java.util.*;
import com.jxml.quick.ocm.*;
import java.io.*;

/**
 * Removes IDs.
 */
public class Element extends OCMrootPeerSupport
{
        /**
         * User-provided code to perform the conversion.
         */
        public void cvt(Object param)
                throws SAXException, IOException
        {
                int i,s;

                OCMelement element=(OCMelement)dataModel;

                s=element.attributes.size();
                for (i=0;i<s;++i)
                {
                        OCMattribute a=(OCMattribute)element.attributes.get(i);
                        if (!"ID".equals(a.name))
                                addStringAttribute(a.name,a.text);
                }

                startElement(element.name);

                s=element.elements.size();
                if (s==0)
                        addStringContent(element.text);
                else
                        doCvt(element.elements,this);

                endElement(element.name);
        }
}
```

OCM begins by converting the input document into a model and then extracting the root object, which in this case is OCMelement. OCM then creates a controller (peer) object and calls the **cvt** method on that controller object.

The cfgNoid controller, Element, subclasses **OCMrootPeerSupport**, which provides the API for generating SAX

events, including the **startElement**, **endElement**, **addStringContent** and **addStringAttribute** methods, as well as a number of other convenience methods.

The member variable **dataModel** is important, as it references the object in the model that the controller object is paired with.

Finally, the **doCvt** method provides the means for invoking **cvt** on other controller objects. The first argument to doCvt is an object from the model. If that object already has a peer controller object, then cvt is called on that controller object. Otherwise a controller object is created for that model object and then cvt is called on the new object. (When no controller object is defined for a model object, doCvt simply returns.)

The second argument of doCvt is simply passed as the **param** argument when calling cvt.

top

Example 2: cfgQjml2Html

The cfgQjml2Html utility converts a QJML file into a collection of web pages using the full MVC pattern. Here's the config file:

```
<cfg>
        <NVItem name="sinkSchema"
                value="classpath:///com/jxml/quick/util/util.qiml"/>
        <NVItem name="sinkIn"
                value="classpath:///com/jxml/quick/util/qjml2html/qjml2html.util"
                fixed="true"/>
        <NVItem name="out"/>
        <NVItem name="generateOutputTo"/>
        <NVItem name="schemaName"/>
        <RequiredItem name="in"/>
        <eval class="com.jxml.quick.ocm.OCMsinkSupport"/>
</cfg>
```

The config files for cfgNoid and cfgQjml2Html do have a lot in common. The sinkSchema parameter is the same. The sinkIn parameter names the util file which defines the utility. These commonalities are because both utilities build on the OCMsinkSupport class.

The parameters **generateOutputTo** and **schemaName** are particularly interesting, as these parameters are used by the cfgQjml2Html controller logic. The generateOutputTo parameter gives the directory name where the web pages are to be written; the schemaName parameter names the schema being processed and is used included in the generated output.

Here's the qjml2html.util file named in the above config file:

```
<contextSupport>
        <input inputSchema="classpath:///com/jxml/quick/model/qjmlModel.qiml"/>
        <factoryMap>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMAbstract"
                        peerClass="com.jxml.quick.util.qjml2html.QCAbstract">
                        <param name="template"
url="classpath:///com/jxml/quick/util/qjml2html/textDetails.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMAttributes"
                        peerClass="com.jxml.quick.util.qjml2html.QCAttributes">
```

```
                        <param name="attributes"
        url="classpath:///com/jxml/quick/util/qjml2html/attributes.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMBean"
                        peerClass="com.jxml.quick.util.qjml2html.QCBean">
                        <param name="template"
        url="classpath:///com/jxml/quick/util/qjml2html/qjml2html.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMElements"
                        peerClass="com.jxml.quick.util.qjml2html.QCElements">
                        <param name="contents"
        url="classpath:///com/jxml/quick/util/qjml2html/contents.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMField"
                        peerClass="com.jxml.quick.util.qjml2html.QCField">
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMItem"
                        peerClass="com.jxml.quick.util.qjml2html.QCItem">
                        <param name="usage"
        url="classpath:///com/jxml/quick/util/qjml2html/usage.txt"/>
                        <param name="attributesDetails"
        url="classpath:///com/jxml/quick/util/qjml2html/attributesDetails.txt"/>
                        <param name="contentDetails"
        url="classpath:///com/jxml/quick/util/qjml2html/contentDetails.txt"/>
                        <param name="urls"
        url="classpath:///com/jxml/quick/util/qjml2html/urls.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMQJML"
                        peerClass="com.jxml.quick.util.qjml2html.QCQJML">
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMSequence"
                        peerClass="com.jxml.quick.util.qjml2html.QCSequence">
                        <param name="template"
        url="classpath:///com/jxml/quick/util/qjml2html/contentDetails.txt"/>
                        <param name="urls"
        url="classpath:///com/jxml/quick/util/qjml2html/urls.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMSelection"
                        peerClass="com.jxml.quick.util.qjml2html.QCSelection">
                        <param name="template"
        url="classpath:///com/jxml/quick/util/qjml2html/contentDetails.txt"/>
                        <param name="urls"
        url="classpath:///com/jxml/quick/util/qjml2html/urls.txt"/>
                </factoryEntry>
                <factoryEntry
                        dataModelClass="com.jxml.quick.model.qjml.QMText"
                        peerClass="com.jxml.quick.util.qjml2html.QCText">
                        <param name="template"
        url="classpath:///com/jxml/quick/util/qjml2html/textDetails.txt"/>
                </factoryEntry>
        </factoryMap>
</contextSupport>
```

The model used by cfgQjml2Html is **qjmlModel**. The classes used in this model provide a rich set of methods for queries and navigation.

Looking at the QJML file for this model, you would see that the root element, qjml, is bound to the class **QMQJML**. In the above util file, this class is in turn bound to **QCQJML**, cfgQjml2Html's root controller class.

Note in particular the param elements found in this util class. These elements bind the template files to the various controller classes, which constitute the view of the MVC pattern. For cfgQjml2Html, these views contain fragments of HTML.

Now lets look at the code for the root controller class, **QCQJML**:

```
package com.jxml.quick.util.qjml2html;
import com.jxml.quick.model.qjml.*;
import com.jxml.quick.*;
import org.xml.sax.*;
import java.util.*;
import com.jxml.quick.ocm.*;
import java.io.*;


public class QCQJML extends OCMrootPeerSupportLite
{
        public void cvt(Object param) throws SAXException, IOException
        {
                QMQJML dm=(QMQJML)dataModel;
                dm.init();
                doCvt(dm.coinMap,null);
        }

        public String express(boolean expressDefaults)
                throws SAXException, IOException
        {
                context.setStringBuffer(new StringBuffer());
                cvt(null);
                return context.getStringBuffer().toString();
        }

        public void express(PrintWriter pw, boolean expressDefaults)
                throws SAXException, IOException
        {
                pw.print(express(expressDefaults));
        }
}
```

Unlike the controler class for cfgNoid, the QCQJML controler class extends **OCMrootPeerSupportLite**, not OCMrootPeerSupport. The OCMrootPeerSupportLite class does not support the API for generating SAX events.

The OCMrootPeerSupportLite.express routines which drive the utility have been overridden by QCQJML. This is because OCMrootPeerSupportLite expects to be producing an XML document. But cfgQjml2Html generates a set of HTML files.

The **context** variable referenced in the first express method has been initialized to reference an instance of a nested class, OCMcontextSupport.SubContextSupport. The methods and member variables on this nested class provide the overall operating context for the conversion. In particular, the StringBuffer in this nested class is the place where all output is collected.

The call to **cvt** in the first express method is the invocation for the convertion itself. On completion of this invocation,

anything remaining in the StringBuffer is displayed. In the case of cfgQjml2Html the StringBuffer would only hold error messages at this point, as the HTML should have already been written to the appropriate files.

The **cvt** method is required, as this method is abstract in OCMrootPeerSupportLite. In this method we first initialize **dm** using the **dataModel** member variable to reference the data model for the qjml element. The call to **dm.init()** then initializes the model, a process which requires a transversal of the entire tree of the model. Finally, the invocation **doCvt(dm.coinMap,null)** is made. This is a convenience method which does a doCvt call for every value in dm.coinMap. (The coinMap is built when the model is initialized. It holds the model objects of all the elements under the root element.)

The doCvt method takes a model object as the first parameter. (Convenience forms of doCvt take a list or map of model objects.) OCM then matches the model object to a controller object. Failing that, OCM matches the model object to a controller class and instantiates the controller object. (If there is no matching controller class, the invocation to doCvt is ignored.) The cvt method on the controller object is then called, passing it the second parameter passed to doCvt.

There are several different elements which can occur under the root element in a QJML file. Here's a table of those elements, the classes they are bound to and the template files used for their view:

QJML Element	Model Class	Controler Class	View
abstract	QMAbstract	QCAbstract	textDetails.txt
bean	QMBean	QCBean	qjml2html.txt
interface	QMInterface	-	-
text	QMText	QCText	textDetails.txt

If you look at these classes you will see that all of these model classes extend the class QMCoin, while all of the controler classes extend QCCoin. (Not having a controller class, the conversion process ignores the interface element, except as data provided via queries to the model.) Here then is the code for the controler class QCCoin:

```
package com.jxml.quick.util.qjml2html;
import com.jxml.quick.model.qjml.*;
import com.jxml.quick.*;
import org.xml.sax.*;
import java.util.*;
import com.jxml.quick.ocm.*;
import java.io.*;

public class QCCoin extends OCMtemplate
{
        public StringBuffer sb;
        public Map properties;
        public QMQJML qjml;
        public TreeMap coinMap;
        public QMCoin ed;
        public String shortClassName;
        public String packageName=null;
        public String folder="";
        public File qjmlInHtml;
        public String targetClassName;
        public String className;
        public String htmlFileName;
        public ArrayList remList=new ArrayList();

        public void cvt(Object param) throws SAXException, IOException
        {
                OCMcontextSupport.SubContextSupport
```

```
scs=(OCMcontextSupport.SubContextSupport)context;
                sb=scs.sb;
                properties=scs.properties;
                qjml=(QMQJML)scs.dataModelRoot;

                String generateOutputTo=(String)properties.get("generateOutputTo");

                coinMap=qjml.coinMap;

                ed = (QMCoin)dataModel;
                targetClassName=ed.getClassName();
                if (targetClassName.startsWith("java."))
                        return;
                if(targetClassName.startsWith("javax."))
                        return;

                if(ed instanceof  QMBean)
                         htmlFileName=((QMBean)param).tag +".html";
                else if(ed instanceof QMText)
                         htmlFileName=((QMText)param).tag +".html";
                else if(ed instanceof QMAbstract)
                         htmlFileName=((QMAbstract)param).getLabel() +".html";

                if(!"".equals(generateOutputTo))
                        htmlFileName=generateOutputTo +"/" + htmlFileName;


                if(!htmlFileName.endsWith(".html"))
                        htmlFileName=htmlFileName + ".html";
                properties.put("className",targetClassName.replace('.','/'));

                expand("template");

                qjmlInHtml=new File(htmlFileName);
                qjmlInHtml.createNewFile();
                FileOutputStream fos=new FileOutputStream(qjmlInHtml);
                sb=context.getStringBuffer();
                byte[] buff=sb.toString().getBytes();
                fos.write(buff);
                sb.delete(0,sb.length());
                fos.close();
        }

        public String process(String templateName, String arg)
                throws SAXException, IOException
        {
                if("className".equals(arg))
                {
                        return targetClassName;
                }

                if("remarks".equals(arg))
                {
                        remList=ed.remList;
                        for(int i=0;i<remList.size();i++)
                        {
                                sb.append(((QMRem)remList.get(i)).getQText());
                                sb.append("\n ");
                        }
                }
                return "";
        }
```

```
}
```

The class QCCoin extends **OCMtemplate**, which extends OCMrootPeerSupportLite in turn. Class OCMtemplate implements a simple template mechanism. Templates are just strings in which the pattern *{{tagName}}* is used. When a template it expanded, it is added to the output after replacing these patterns with the values returned by the process method. (Where the tagNames found in the template are passed as a parameter when calling process.)

Time to take a look at the QCCoin.cvt method:

From the **content** member variable we are able to access a number of items:

- **sb** - the StringBuffer to which all ouptut is appended;
- **dataModelRoot** - the root object of the model; and
- **properties** - a map holding the parameters from the config file, including in this case **generateOutputTo**.

The **dataModel** member variable is used to access the model object that the control object is bound to.

The **expand** method is used to fetch a template bound to the controller object by the util file. Again, the template is searched for a *{{arg}}* pattern and expanded bay a call to the process method.

The last step in the cvt method writes the contents of the StringBuffer, sb, to an .html file and then clears the StringBuffer.

The QCCoin.process method is used to expand the patterns *{{**className**}}* and *{{**remarks**}}*, but it is overridden by classes which extend QCCoin to support the expansion of additional patterns.

Lets look next at QCText, a controller class which extends QCCoin:

```
package com.jxml.quick.util.qjml2html;
import com.jxml.quick.model.qjml.*;
import com.jxml.quick.*;
import org.xml.sax.*;
import java.util.*;
import com.jxml.quick.ocm.*;
import java.io.*;

public class QCText  extends QCCoin
{
        public QMText ed;

        public void cvt(Object param)
                throws SAXException, IOException
        {
                OCMcontextSupport.SubContextSupport
scs=(OCMcontextSupport.SubContextSupport)context;
                StringBuffer sb=scs.sb;
                ed=(QMText)dataModel;
                if (param==null)
                        super.cvt(ed);
                else
                {
                        ArrayList remList=ed.remList;
                        int s=remList.size();
                        for(int i=0;i<s;i++)
                        {
                                sb.append(((QMRem)remList.get(i)).getQText());
                        }
                }
```

```
            }

            public String process(String templateName, String arg)
                    throws SAXException, IOException
            {
                    if("elementName".equals(arg))
                            return ed.tag;

                    return  super.process(templateName,arg);
            }
    }
```

The importing thing to note here is the **param** parameter on the cvt method. In cfgQjml2Html, by convention, the root controller object, QCQJML, invokes its subordinate controller objects with a param of NULL.

Now there is no reason to restrict the data in the output generated by the MVC transform to be in the same overall order as the input to the transform. This can partly be handled by adding query logic to the data model.

Another approach to handle output data whose order differs from the input data is to invoke the controller objects in the order needed to properly arrange the output, perhaps by using navigational methods on the model and then a doCvt call to access the controller object. This gives rise to the possibility of invoking the controller object multiple times, to produce different data at different points in the output. This is when **param** can be used to indicate the kind of data to be output for a particular invocation.

In the case of QCText, when param is NULL, the invocation is from QCQJML and we need to invoke QCCoin.cvt by calling **super.cvt**. Otherwise, all we need to do is to display any associated rem elements.

When QCText invokes QCCoin.cvt, that method in turn calls QCText.process, which checks for the *{{**elementName**}}* pattern in the template before invoking QCCoin.process by calling **super.process**.

Finally, lets look at the file textDetail.txt, which is the template bound to the QCText controller:

```
    <HTML>
    <HEAD><TITLE>Title</TITLE></HEAD>
    <BODY>
    <CENTER>
            <TABLE BORDER="5" width="100%">
            <TBODY>
            <TR>
            <TD BGCOLOR="skyblue">   <b>*{{elementName}}*</b></TD>
            <TD>
                    <TABLE BORDER="1" width="100%">
                    <TBODY>
                    <TR>
                    <TD BGCOLOR="yellowgreen">
                    *{{className}}*
                    </TD>
                    </TR>
                    <TR>
                    <TD>
                            <TABLE BORDER="1" width="100%">
                            <CAPTION><B>Contents:</B></CAPTION>
                            <TBODY>
                             *{{contentDetails}}*
                            </TBODY>
                            </TABLE>
                    </TD>
                    </TR>
                    <TR>
```

```
                    <TD>
                            <TABLE BORDER="1"width="100%">
                            <TBODY>
                            <TD BGCOLOR="white">(text content)</TD>
                            <TD BGCOLOR="pink">Required</TD>
                            <TD BGCOLOR="gold">*{{remarks}}*</TD>
                            </TBODY>
                            </TABLE>
                    </TD>
                    </TR>
                    </TBODY>
                    </TABLE>
            </TD>
            </TR>
            </TBODY>
            </TABLE>
    </CENTER>
    </BODY>
    </HTML>
```

Except for the ***{{arg}}*** patterns, the template file is the html file generated when a text element is processed by cfgQjml2Html. This considerably reduces the maintenance effort for cfgQjml2Html.

top

Part VI: Links

- **Products Page** for Quick, JXUnit, JXWeb and Qare (with links for license, download, installation guide, release notes, statistics, and home page)
- **Utilities Overview**
- Schema Markup Languages: **QDML**, **QJML** and **QIML**
- Markup Languages used by the Quick Utilities: **Config**, **OCM** and **Util**
- API (javadoc): **Protocol**, **Quick**, **QDoc**, **QBiModal**, **QName**, **ocm package**
- Quick Utility Models (javadoc): **qdml package**, **qjml package**, **qiml package**

top

jxquick: Data-Centric Programming with Java and XML

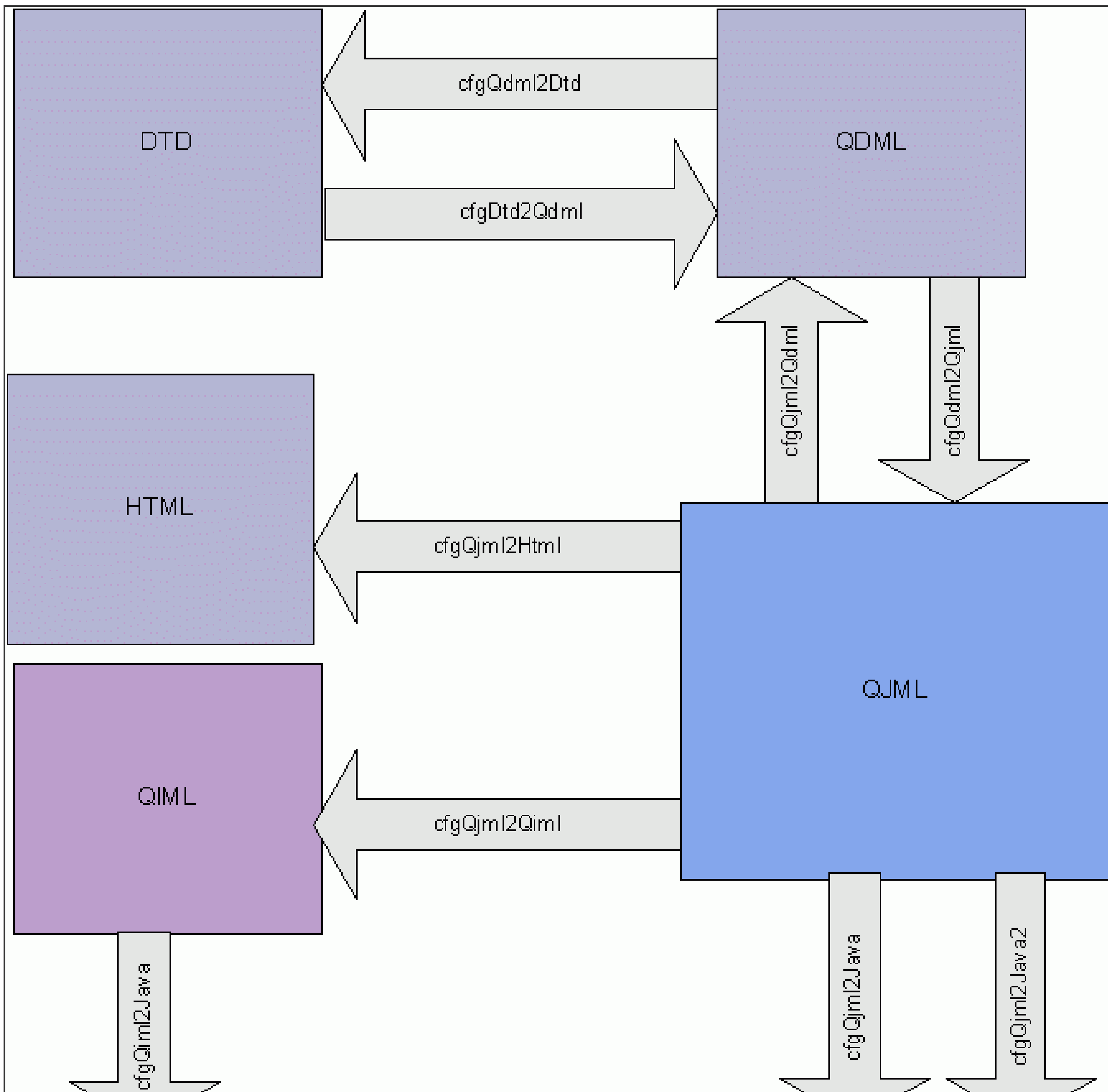
Topic:

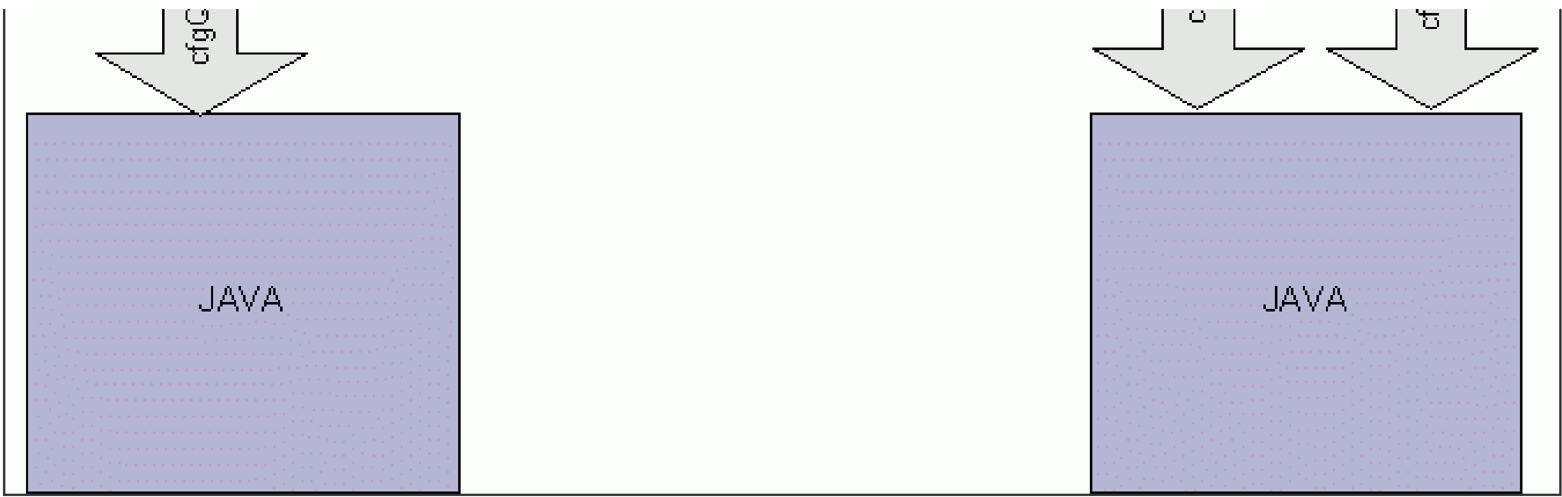
jxquick: Data-Centric Programming with Java and XML

Topic:

Site Map

Quick Utilities





note Be sure to run CfgSetQdmlRoot before running CfgQdml2Qjml!!!

(For CfgQjml2Java see also CfgQjml2Java2)

Utilities are a piece of code ---tools which convert one form of file into another.

CfgDtd2Qdml	CfgDtd2Xml	CfgDtdPrettyPrint	CfgNoid
CfgQdml2Dtd	CfgQdml2Qjml	CfgQdmlDefaults	CfgQdmlExpand
CfgQdmlExpandID	CfgQdmlNoDefaults	CfgQdmlValidate	CfgQiml2Java
CfgQjml2Java	CfgQjml2Qdml	CfgQjml2Qiml	CfgQjmlDefaults
CfgQjmlExpand	CfgQjmlExpandID	CfgQjmlNoDefaults	CfgQjmlValidate
CfgSetid	CfgSetQdmlRoot	CfgQjml2Java2	CfgQjml2Html

OSDN: Our Network - Newsletters - Advertise - Shop SEARCH:

Project: Java/XML Quick: Summary

Summary | Admin | Home Page | Forums | Tracker | Bugs | Lists | Tasks | News | CVS | Files |

Quick is a Java package for workin with XML. Quick uses a binding schema to map XML tags to Java classes. Quick is bi-directional: XML <=> Objects. (JDK1.2 is required)

:Java Foundry

- Development Status: 5 - Production/Stable
- Intended Audience: Developers
- License: GNU Library or Lesser General Public License (LGPL)
- Operating System: OS Independent
- Programming Language: Java
- Topic: Software Development

Project UNIX name: jxquick
Registered: 2000-07-11 06:22
Activity Percentile (last week): 70.4715%
View project activity statistics

Developer Info

Project Admins:
as_tiwari
sudeshsoni
simonstl
dtauzel
blaforge

Developers:
13 [View Members]

Latest File Releases

Package	Version	Date	Notes / Monitor	Download
jxquick	4.3.1	May 30, 2002	Re - M	Download
javadocs	javadocs	July 2, 2002	Re - M	Download

[View ALL Project Files]

Most Active

1 phpMyAdmin
2 TUTOS
3 Compiere ERP + CRM Business Solution
4 Firewall Builder
5 JBoss.org
6 CDex
7 phpChrystal - An Open Intranet System
8 SquirrelMail
9 Dev-C++
10 Privoxy

More Activity>>

Top Downloads

1 CDex
2 VirtualDub
3 Java/C++ Registrar Toolkit for EPP V04
4 ZSNES
5 JBoss.org
6 phpMyAdmin
7 Gnucleus
8 DC++
9 Dev-C++
10 GnuWin32

More Statistics>>

Sponsored Content

Public Areas

Project Home Page

Tracker

- Bugs (**7 open / 15 total**)
Bug Tracking System

- Feature Requests (**1 open / 1 total**)
Feature Request Tracking System

Public Forums (**542** messages in **2** forums)

Mailing Lists (**1 total**)

Task Manager
- Quick Internals
- QJML Wizard
- Misc. Utilities
- Schema

CVS Repository (**1,013** commits, **252** adds)
- Browse CVS

Latest News

JXQUICK Release 4.0.0
blaforge - 2001-02-09 04:30
(0 Comments) [Read More/Comment]

Release 3.3.1
blaforge - 2000-10-03 10:06
(0 Comments) [Read More/Comment]

[News archive]
[Submit News]

Powered by SourceForge(tm) collaborative software development (CSD) platform from VA Software
© Copyright 2002 - OSDN Open Source Development Network, All Rights Reserved
About SourceForge.net • About OSDN • Privacy Statement • Terms of Use • Advertise • Self Serve Ad System • Contact Us