Java Meets Quark

Contributors: Luke Evans, Edward Lam, Rick Cameron, Magnus Byne, Joseph Wong
Last modified: October 24, 2007

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of Business Objects nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Contents

Overview

This document explains how to use Quark in a program written in Java.

Before reading this document you should be familiar with the basics of Quark programming. You should also be familiar with the Gem Cutter and the concepts it uses, such as *gems*, *collectors* and *emitters*.

Scope

This document does not attempt to provide a complete guide to combining Quark and Java. In focuses primarily on how Quark can be incorporated into a Java program.

Quark offers a functional approach whereas Java provides an object oriented view; both are complete systems with distinct advantages. Depending on the nature of the problem at hand it may be appropriate to use Java or Quark for differing amounts of the solution.

Document layout

This document is divided into the following sections:

In **Why Use Quark in a Java Application?** we discuss how Quark differs from an object-oriented language like Java, and examine the benefits of using Quark code in a Java application

In **Workspaces** we explore the various ways you can organize your Quark modules.

In **Initializing Quark and Running Gems** we show how to run simple gems from Java.

In **Input and Output Policies** we describe in more detail how values can be marshaled between Java and Quark.

In **Exploring the Workspace** we examine the entities in a workspace and how they are related.

In **The High-Level API: Gem Graphs** we see how to plug together existing gems and run the resulting composition, much as you would do in the Gem Cutter.

In **The Low-Level API: Source Model** we explore the API that allows you to build Quark functions in an incremental manner.

In **Using Java from CAL** we show how CAL can use Java classes.

In **Advanced Topics** we delve into some of the features you'll need to use in special circumstances.

In **Deploying Quark-based Applications** we discuss commonly used approaches for deploying applications developed with Quark.

Call for feedback

This document aims to be as complete and easy-to-use as possible. Any feedback you might have to help improve it would be very welcome. Please send any comments, suggestions, or questions to the CAL Language Discussion forum on Google Groups (http://groups.google.com/group/cal_language).

Why Use Quark in a Java Application?

Quark is Designed for Data Transformation

The Quark system is based on functional reduction semantics common to many functional languages. As a functional language, CAL is very different from an object-oriented language like Java.

- The fundamental components of Java are objects, which usually carry state. Executing a method call in Java can produce side effects that alter the state of the system.

- The fundamental components of CAL are functions, which are stateless[1]. A call to a function in CAL will typically not modify any state; it will return a result that is determined completely by the arguments of the call.

- Java is an imperative language: the statements in a function will be executed in sequence, whether the result of a statement is needed or not.

- CAL is a declarative language: each expression in a function's definition will only be evaluated if the result it produces is needed. In addition, evaluation is done lazily, so that only the minimal work needed to produce a result is performed.

- CAL has a rich set of primitive data types (including lists, tuples and records) and has powerful, simple syntax to perform basic operations on these types. As a result, a data transformation can usually be expressed much more succinctly in CAL without sacrificing clarity.

Quark is better at capturing actions on data in arbitrary size "chunks" and at any level of abstraction. Every piece of behavior is precisely typed, no matter what it represents, and unlike Java, there is no need to build heavier-weight programming artifacts like interfaces to describe the contracts that one piece of implementation makes with the rest of the world.

Because types can be inferred automatically for any function definition, Quark-based logic has a compositional style, and this is much closer to formal algebras for data. Not only does this mean that data logic can be expressed concisely, but it tends to have a syntactic representation where it is more apparent what transformations have been combined – the code itself is more like a specification.

---

[1] Although this is typically the case, CAL can import and use Java functions which may not be stateless.

For instance, relational algebra, dimensional calculations, structural transformations and arithmetic will all appear more "direct" in Quark. Additionally, there is likely to be much more validation of composed logic than would be achieved in an object-oriented implementation.

Workspaces

Every language object in Quark – functions, type declarations, type class declarations, etc. – is defined in a module. For example, the function `add`, the type `Boolean` and the type class `Ord` are all defined in the module `Prelude`.

A workspace is a collection of modules. In order to use Quark functions from Java, you must define a workspace that includes all the modules you need and any dependent modules that are required.

Quark modules can be stored in several different kinds of *vaults*. The only kind of vault supported by this release of Quark is the *standard vault*. This vault represents the Java class path. The loader looks in subdirectories named "CAL" for CAL source files.

For further information on vaults, please see Appendix A.

A workspace definition can refer to Quark modules in the standard vault. It can also import other workspace definitions.

For example, here is the default workspace definition for the Gem Cutter (with comments removed):

```
StandardVault GemCutterSaveModule

import StandardVault cal.libraries.cws
```

Such a workspace definition can be stored in a text file with the extension `.cws` and placed in a directory called "Workspace Definitions" on the Java class path.

Initializing Quark and Running Gems

In a Java program that uses Quark, the starting point is an instance of the class `org.openquark.cal.services.BasicCALServices`. The instance is usually created by a call to the static factory method `makeCompiled`. Example code which creates a compiled `BasicCALServices` instance is shown below:

```
CompilerMessageLogger messageLogger = new MessageLogger();

BasicCALServices calServices =
       BasicCALServices.makeCompiled(workspaceFileName, messageLogger);

if (calServices == null)
       System.err.println(messageLogger.toString());
```

workspaceFileName: The name of the workspace e.g. "cal.platform.test.cws"

To run an existing gem you must first you must create an `EntryPointSpec`, which contains the Gem's name. The Gem can then be run using the `BasicCALServices` instance. The example below shows how to do this for the `getNthPrime` gem:

```
EntryPointSpec getNthPrimeEntrySpec =
    EntryPointSpec.make(QualifiedName.make(CALPlatformTestModuleNames.M2,
                                           "getNthPrime"));

Object result = calServices.runFunction(getNthPrimeEntrySpec,
                                        new Object[] {new Integer(5000)});

System.out.println("getNthPrime(5000) is: " + result);
```

The `getNthPrime` gem computes the n^{th} prime number. It has a very simple type signature; it takes a single integer as input, n, and outputs a single integer representing the n^{th} prime number.

Input and Output Policies

The way that values are passed from the Java world to the Quark world and back is controlled by input and output policies. Each input and output policy specifies a function that is used to marshal values between Java and Quark types. Input and output policies can be specified using some of the factory methods for `EntryPointSpec`, e.g.:

```
public static EntryPointSpec make(
      QualifiedName functionalAgentName,
      InputPolicy[] inputPolicies,
      OutputPolicy outputPolicy)
```

If the input or output policies are not specified the default input policy, `InputPolicy.DEFAULT_INPUT_POLICY`, is used to marshal each input, and the default output policy, `OutputPolicy.DEFAULT_OUTPUT_POLICY`, is used to marshal the output.

The default input policy uses the `Inputable` class method `input` to marshal values. Therefore in order to use the default input policy the Quark type must be an instance of `Inputable`. The default output policy uses the `Outputable` class method `output` to marshal values. Therefore, in order to use the default output policy the Quark type must be an instance of `Outputable`.

All the basic Quark types are instances of `Inputable` and `Outputable`, allowing the default policies to be used. They provide marshalling as described below:

CAL Type	Input From	Output To
Boolean	Java.lang.Boolean	Java.lang.Boolean
Byte	Java.lang.Byte	Java.lang.Byte
Char	Java.lang.Character	Java.lang.Character
Byte	Java.lang.Byte	Java.lang.Byte
Short	Java.lang.Short	Java.lang.Short
Int	Java.lang.Integer	Java.lang.Integer
Integer	java.math.BigInteger	java.math.BigInteger
Long	Java.lang.Long	Java.lang.Long
Float	Java.lang.Float	Java.lang.Float
Double	Java.lang.Double	Java.lang.Double
String	Java.lang.String	Java.lang.String
[a]	Java.util.List, java.util.Iterator	Java.util.List
Maybe a[2]	org.openquark.cal.foreignsupport.module.Prelude.MaybeValue	org.openquark.cal.foreignsupport.module.Prelude.MaybeValue
Either a b[3]	Org.openquark.cal.forignsupport.module.Prelude.Either	Org.openquark.cal.forignsupport.module.Prelude.Either

[2] The type a must be an instance of Inputable and Outputable

CAL Type	Input From	Output To
`Record`[4]	`Java.util.List`[5], `java.util.Map`[6]	`Java.util.List`

It is possible to make any user defined type `Inputable` and `Outputable`. The details of this are covered in the section Implementing Inputable and Outputable.

Other Input Output Policies

This section briefly describes some other input and output policies. Some complete examples can be found in `samples/org/openquark/cal/samples/InputOutputDemo.java` in the Quark distribution.

Typed Input and Output Policies

Many Quark gems are polymorphic. For example, the `CAL.Core.Prelude.add` gem works with many types of numbers. If the default input and output policies are used, the types of the inputs will be ambiguous and an exception will occur. To overcome this, a typed policy must be used. Typed input and output policies are defined for most of the simple CAL types, for example `INT_INPUT_POLICY`, `FLOAT_INPUT_POLICY`, `BOOLEAN_OUTPUT_POLICY`. Typed policies for other types can be created using the Input and Output policies' factory methods.

Iterator Output Policy

A key feature of Quark is lazy evaluation. For example, the `Cal.Test.General.M1.allPrimes` gem exploits this to represent an infinite list containing all the prime numbers. Values in the `allPrimes` list are only computed as they are needed. To use the `allPrimes` gem from Java the result cannot be output as a Java list, the default representation. Instead the `OutputPolicy.ITERATOR_OUTPUT_POLICY` is used, which marshals the Quark list to an `Iterator`. The following example shows the steps needed to use the `allPrimes` gem:

---

[3] The types a and b must be instances of Inputable and Outputable

[4] All the fields in the record must be instances of Inputable and Outputable

[5] When a CAL record is input or output as a Java list, the list must have exactly the same number of elements as the record has fields. The elements in the list are interpreted in the following order: ordinal fields first, ordered numerically, followed by textual fields, the ordering of which is defined by the `Sting.CompareTo` order of the field names.

[6] When a CAL record is input as a Java map, the set of keys in the map must exactly match the set of record field names.


```
EntryPointSpec allPrimesEntrySpec =
    EntryPointSpec.make(QualifiedName.make(CALPlatformTestModuleNames.M1,
                                           "allPrimes"),
                        OutputPolicy.ITERATOR_OUTPUT_POLICY);

Iterator allPrimesIterator =
    (Iterator)calServices.runFunction(allPrimesEntrySpec, new Object[0]);
```

CAL Value Input and Output Policies

Often the result of one gem is used as the input to another. In such a case it is inefficient to marshal the result to Java then back to CAL again. In some cases, it is not possible – for example a value may not be not `inputable` and `outputable`. In this case the `CAL_VALUE_OUTPUT_POLICY` can be used, which simply returns an opaque reference to a CAL value. Such values can be used in subsequent calls to gems using the `CAL_VALUE_INPUT_POLICY`.

Exploring the Workspace

A Quark workspace contains several kinds of entities:

Entity	What is it?	Java Type[7]	Accessed From
Module	A top-level container in a workspace	`ModuleTypeInfo`	`WorkspaceManager (getModuleNamesInProgram, getModuleTypeInfo)`
Type Constructor	The declaration of a data type	`TypeConstructor`	`ModuleTypeInfo (getNTypeConstructors, getNthTypeConstructor)`
Data Constructor	One of the constructors in a data type	`DataConstructor`	`TypeConstructor (getNDataConstructors, getNthDataConstructor)`
Type Class	The declaration of a type class	`TypeClass`	`ModuleTypeInfo (getNTypeClasses, getNthTypeClass)`
Class Method	One of the methods in a type class	`ClassMethod`	`TypeClass (getNClassMethods, getNthClassMethod)`
Class Instance	The declaration that a data type belongs to a type class	`ClassInstance`	`ModuleTypeInfo (getNClassInstances, getNthClassInstance)`
Instance Method	Within a class instance, the implementation of a method	`InstanceMethod`	`ClassInstance (getNInstanceMethods, getNthInstanceMethod)`
Function	A top-level function in a module	`Function`	`ModuleTypeInfo (getNFunctions, getNthFunction)`

[7] All the types are in the package `com.businessobjects.lang.cal.compiler`

For example:

```
module Example;                              // a Module

public data Point =                          // a Type Constructor
    Point                                    // a Data Constructor
        x :: Int
        y :: Int
    ;

public class Sizeable a where                // a Type Class
    public magnitude :: a -> Double;         // a Class Method
    ;

instance Sizeable Point where                // a Class Instance
    magnitude = pointMagnitude;              // an Instance Method
    ;

public pointMagnitude pt =                   // a Function
    case pt of
        Point x y -> sqrt (toDouble (x * x) + toDouble (y * y));
    ;
```

You can start exploring the workspace from the instance of `BasicCALServices` you created (see the section **Initializing Quark and Running Gems**). A call to the method `getWorkspaceManager` returns the `WorkspaceManager` for the current workspace. `WorkspaceManager` has two methods that provide access to modules:

```
String[] getModuleNamesInProgram()
ModuleTypeInfo getModuleTypeInfo (ModuleName moduleName)
```

From an instance of `ModuleTypeInfo` you can retrieve type constructors, type classes, class instances and functions. For each of these entities there is a pair of methods that return the number of that kind of entity in the module and the n^{th} entity. For example:

```
int getNFunctions()
Function getNthFunction(int n)
```

Similarly, you can get data constructors, class methods and instance methods from their parent entities as shown in the table above. Once again, there are methods that return the count of entities and the n^{th} entity.

All of these entities are immutable, since they represent the state of the workspace. You can add or remove modules by calling methods on the `WorkspaceManager`, and you can add functions and other entities to a module by compiling CAL source code.

Data constructors, functions and class methods are all functional agents; that is, they can be used in function applications. You can retrieve a `TypeExpr` that

represents the type signature of a functional agent by calling its `getTypeExpr` method.

The `TypeExpr` for a functional agent is made up of 0 or more argument types and the result type. Each of these types is also represented by a `TypeExpr`.

`TypeExpr` has methods that allow you to explore its structure:

```
int getArity()
TypeExpr[] getTypePieces()
TypeExpr getArgumentType()
TypeExpr getResultType()
```


Metadata

Each of these entities, as well as a few others, has a set of properties associated with it. In the Quark API this is called the metadata on the entity.

Metadata is available for these entities:

Entity	Metadata Class[8]
Module	`ModuleMetadata`
Function	`FunctionMetadata`
Class method	`ClassMethodMetadata`
Type constructor	`TypeConstructorMetadata`
Data constructor	`DataConstructorMetadata`
Type class	`TypeClassMetadata`
Class instance	`ClassInstanceMetadata`
Argument	`ArgumentMetadata`
Instance method	`InstanceMethodMetadata`

You can retrieve the metadata for an entity from the `CALWorkspace` object. This object can be obtained by calling `getCALWorkspace` on the `BasicCALServices` object, or by calling `getWorkspace` on the `WorkspaceManager` object.

The `getMetadata` method of `CALWorkspace` has two overloads:

```
ScopedEntityMetadata getMetadata(ScopedEntity entity, Locale locale)
CALFeatureMetadata getMetadata(CALFeatureName featureName, Locale locale)
```

The object returned can be cast to the appropriate class for the kind of entity.

Some entities (functions, class methods, type constructors, data constructors and type classes) can be passed to the first overload. Otherwise you can create a `CALFeatureName` to represent the entity. This class combines the kind of entity with its name. Its constructor is:

```
CALFeatureName(FeatureType type, String name)
```

…and it has static final fields representing the various `FeatureTypes`, such as `CALFeatureName.MODULE`, `CALFeatureName.FUNCTION`, etc.

Metadata for arguments can be retrieved by calling `getArguments` on an instance of `FunctionMetadata`, `ClassMethodMetadata`, `DataConstructorMetadata` or `InstanceMethodMetadata`.

---

[8] All of these classes are in the package `com.businessobjects.lang.cal.metadata`

All entities support the following properties:

Property	Data Type	Accessors
Display name	`String`	`getDisplayName` `setDisplayName`
Short description	`String`	`getShortDescription` `setShortDescription`
Long description	`String`	`getLongDescription` `setLongDescription`
Expert[9]	`Boolean`	`isExpert` `setExpert`
Hidden	`Boolean`	`isHidden` `setHidden`
Preferred[10]	`Boolean`	`isPreferred` `setPreferred`
Creation date[11]	`Date`	`getCreationDate` `setCreationDate`
Modification date[12]	`Date`	`getModificationDate` `setModificationDate`
Version	`String`	`getVersion` `setVersion`
Author	`String`	`getAuthor` `setAuthor`

Some of the entities support additional standard properties:

Property	Applies To	Data Type	Accessors
Categories	Function, Class method, Data constructor, Instance method	`String[]`	`getCategories` `setCategories`
Examples	Function, Class method, Data constructor, Instance method	`CALExample[]`	`getExamples` `setExamples`

[9] This property is described as follows: "true if this feature is intended for expert users rather than normal users"

[10] This property is described as follows: "true if this feature is particularly important for presenting to humans"

[11] When the metadata is saved, if the creation date hasn't already been set, it is set to the current date/time.

[12] When the metadata is saved, the modification date is always set to the current date/time.

Property	Applies To	Data Type	Accessors
Return value description	Function, Class method, Instance method	`String`	`getReturnValueDescription` `setReturnValueDescription`

You can add your own properties to a metadata object. A user-defined property is identified by a `String` name and its value must be a `String`.

There are several methods on `CALFeatureMetadata` that manage user-defined properties:

```
String getAttribute(String attributeName)
void setAttribute(String attributeName, String value)
void clearAttribute(String attributeName)
Iterator getAttributeNames()
void clearAttributes()
```

Localization

The methods that retrieve a metadata object take a `Locale` parameter. This is used to search for a localized copy of the metadata. If there is no metadata for the requested locale, an attempt is made to load metadata for more general locales – that is, first ignoring the variant, then the country, finally trying the invariant locale (`LocaleUtilities.INVARIANT_LOCALE`).

If this search doesn't find a metadata object, a new object is created using the locale as requested.

If you want to create metadata that applies to all locales, you can pass `LocaleUtilities.INVARIANT_LOCALE` as the locale parameter to `getMetadata`.

Persistence

Metadata is loaded automatically. If you modify metadata, you are responsible for saving it.

To save a metadata object, call the method `saveMetadata(CALFeatureMetadata)` on the `CALWorkspace` object.

The High-Level API: Gem Graphs

The Gem Graph API is like a programmatic version of the Gem Cutter application. It allows you to construct a gem design just as you would in the Gem Cutter. You can add function gems, value gems, collectors, emitters and code gems to the graph, make connections between them and burn inputs.

The Gem Graph API is appropriate for applications that use Quark in a compositional mode, plugging together existing gems to perform a computation. If you need to create custom Quark functions, you may find the Source Model API, described below, suits you better.

Once you have created a `GemGraph`, you can compile it, producing an executable object that you can invoke many times, supplying arguments for unbound inputs and retrieving the result.

If you're not already acquainted with the Gem Cutter application, we recommend that you familiarize yourself with its concepts before continuing with this section.

There is a sample program that uses the Gem Graph API in the file `samples/com/businessobjects/lang/cal/samples/GemModelDemo.java` in the Quark distribution.

Initialization

To use the Gem Graph API you should start by creating an instance of `BasicCALServices` as described above in the **Initializing Quark and Running Gems** section. In what follows, we will assume that there is an instance of `BasicCALServices` available, called `calServices`.

It is also useful to create a `GemFactory` for creating gems, e.g.

```
GemFactory gemFactory = new GemFactory (calServices);
```

Creating a `GemGraph`

To create a `GemGraph` you call the default constructor of the class:

```
GemGraph gemGraph = new GemGraph ();
```


Adding a Gem to the `GemGraph`

Each kind of gem is added to the graph in a similar way: you create the gem, using a constructor for the appropriate class or the `GemFactory`, and you add the gem to the graph with a call to `GemGraph.addGem`.

Function gem	`gem = gemFactory.makeFunctionalAgentGem(QualifiedName);`
Value gem	`gem = gemFactory.makeValueGem (valueObject);`
Collector	`gem = new CollectorGem ();` `gem.setName ("name"); // optional`
Emitter	`gem = new EmitterGem (collector);`
Code gem[13]	`gem =` `  new CodeGem (codeAnalyzer, visibleCode, argVarNames);`

Currently, a value gem can be created for any of the following Java classes: `Character`, `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`, `Color` and `com.businessobjects.util.time.Time`.

Connecting Gems

To connect the output of one gem to one of the inputs of another gem, you can use `GemGraph.connectGems`. This function takes a `PartOutput` and a `PartInput` as parameters. The `PartOutput` for the first gem can be retrieved by calling `getOutputPart`, and the n^{th} `PartInput` for the second gem can be obtained by calling `getInputPart (partN)`.

Burning an Input

Burning an input on a gem changes the result type of the gem from a simple data type to a function type. This allows the gem to be connected to a gem that takes a functional input, such as `List.map`.

For example, if we start with the `add` gem, connect its first input to the constant `10` and burn the second input, we create a gem whose result is the function "add 10 to x".

To burn an input for a gem, you retrieve the `PartInput` using `getInputPart (partN)` and call `setBurnt (true)` on the `PartInput`.

[13] We won't discuss code gems further in this document. If you need to create a function by building up CAL code, the source model is a much better approach.

Compiling the `GemGraph`

Every `GemGraph` has a special collector called the target collector. It can be retrieved by a call to `GemGraph.getTargetCollector`.

When the graph is compiled and run, the result of execution will be the value fed into the target collector. Thus, it is important to connect the target collector to the gem that produces the final result of your computation.

To compile the `GemGraph`, perform the following steps:

1. Convert the `GemGraph` to a source model function by calling `GemGraph.getCALSource`.

2. Add the function to the current program.

Here is an example:

```
// step 1: GemGraph to source model
SourceModel.FunctionDefn.Algebraic functionDef = gemGraph.getCALSource();

// step 2: Add the function to a new module in the current program
EntryPointSpec gemEntryPointSpec =
   calServices.addNewModuleWithFunction(ModuleName.make("NewGemModule"),
                                        functionDef);
```

In the example the new Gem is added to a module called "NewGemModule". If this module already existed it would be replaced.

Running the Program

Once an `EntryPointSpec` has been obtained, it is used in exactly the same way as described in the section **Initializing Quark and Running Gems**. If the `GemGraph` has unbound inputs, you must supply arguments for the inputs in the form of an array of `Object`. The result of the computation will be returned as an `Object`.

The Low-Level API: Source Model

If programming against the Gem Graph API is like working in the Gem Cutter, then programming against the Source Model API is like writing CAL source code by hand.

This means that the Source Model API is extremely powerful and flexible. You might expect that it would also be more verbose than the Gem Graph API; however, this isn't always the case.

The classes that make up the Source Model are all inner classes of `com.businessobjects.lang.cal.compiler.SourceModel`. There is a class that corresponds with each syntactic element of the CAL language. For example, a complete module is represented by `SourceModel.ModuleDefn`; a function definition by `SourceModel.FunctionDefn.Algebraic`.

Each class has a static method called `make` that can be used to create an instance. Often there are several overloads of `make`.

A sample program that uses the source model API can be found in the file `samples/org/openquark/cal/samples/SourceModelDemo.java` in the Quark distribution.

Source Model Classes

Here is a list of some commonly-used syntactic elements and the corresponding classes:

Syntactic Element	Source Model Class
Module	`SourceModel.ModuleDefn`
Algebraic[14] function definition	`SourceModel.FunctionDefn.Algebraic`
Foreign function definition	`SourceModel.FunctionDefn.Foreign`
Local function definition (used within a `let` expression)	`SourceModel.LocalDefn.Function`
Variable, function or class method reference	`SourceModel.Expr.Var`
Numeric literal	`SourceModel.Expr.Literal.Num`

Arithmetic operators	`SourceModel.Expr.BinaryOp.Add`
	`SourceModel.Expr.BinaryOp.Subtract`
	`SourceModel.Expr.BinaryOp.Multiply`
	`SourceModel.Expr.BinaryOp.Divide`

[14] An algebraic function definition is one that is implemented as CAL source code.

Syntactic Element	Source Model Class

Comparison operators	`SourceModel.Expr.BinaryOp.Equals`
	`SourceModel.Expr.BinaryOp.NotEquals`
	`SourceModel.Expr.BinaryOp.LessThan`
	`SourceModel.Expr.BinaryOp.LessThanEquals`
	`SourceModel.Expr.BinaryOp.GreaterThan`
	`SourceModel.Expr.BinaryOp.GreaterThanEquals`

Let expression	`SourceModel.Expr.Let`
Case expression	`SourceModel.Expr.Case`

Case alternatives	`SourceModel.Expr.Case.Default`
	`SourceModel.Expr.Case.Pattern.Var`
	`SourceModel.Expr.Case.UnpackListCons`
	`etc.`

Function application	`SourceModel.Application`
Parameter declaration	`SourceModel.Parameter`
Reference to a type constructor (used in a type declaration)	`SourceModel.TypeExprDefn.TypeCons`
Function type expression	`SourceModel.TypeExprDefn.Function`
List type expression	`SourceModel.TypeExprDefn.List`
Type variable	`SourceModel.TypeExprDefn.TypeVar`
Function type declaration	`SourceModel.FunctionTypeDeclaration`
Type signature	`SourceModel.TypeSignature`

Using the Source Model

Since the source model is as flexible as writing CAL source code, it's difficult to capture its use in a series of steps. In general, though, you will do the following:

1. Write methods that create the top-level elements of the module, such as type declarations and function definitions.

2. Create a `SourceModel.ModuleDefn` from these top-level elements.

3. Add the module to the current program by calling `BasicCALServices.makeNewModule`.

4. Run functions in the new module.

For detailed examples of how to use these classes, please see the sample file `samples/com/businessobjects/lang/cal/samples/SourceModelDemo.java`.

There are some useful helper methods in the class `SourceModelUtilities`. For example, `SourceModelUtilities.ImportAugmenter` can be used to determine the minimal list of modules that need to be imported into a `SourceModel.ModuleDefn`.

Functions in the new module can be run in exactly the same way as described in the section **Initializing Quark and Running Gems**.

Java-CAL Binding Classes

The ICE tool can produce a Java binding class from a Quark module, using the command `:javaBinding`. The generated class includes fields and methods that can be used as shortcuts when programming against the source model.

For a module named `Foo`, the Java binding class is called `CAL_Foo`.

Each binding class has one or more inner classes, such as `TypeClasses`, `TypeConstructors`, `DataConstructors` and `Functions`. Each of these inner classes holds the helper fields and methods for that kind of Quark object. For example, the `Functions` class has a static field that holds the `QualifiedName` of each function, and a static method that can be called to create a `SourceModel.Expr` that represents a call to the function.

The parameters of these static methods are normally of the type `SourceModel.Expr`, but if the Quark function has one or more parameters that are declared as simple types, such as `String`, `Int` or `Double`, you will find an overload of the static method that has parameters declared with the corresponding Java classes.

For example:


```
// Make a TypeCons representing the Quark type Prelude.Integer
SourceModel.TypeExprDefn.TypeCons integerTypeCons =
  SourceModel.TypeExprDefn.TypeCons.make(
    CAL_Prelude.TypeConstructors.Integer
  );

// Make the equivalent of List.product (Prelude.upFromTo 1 n)
SourceMode.Expr productCall =
  CAL_List.Functions.product(
    CAL_Prelude.Functions.upFromTo(
      SourceModel.Expr.Literal.Num.make(1),
      SourceModel.Expr.Var.make(null, "n"))
  );
```

The implementation of these helpers is very simple:

```
public static final QualifiedName Integer =
  QualifiedName.make(CAL_Prelude.MODULE_NAME, "Integer");

public static final SourceModel.Expr product(SourceModel.Expr list) {
  return SourceModel.Expr.Application.make(
    new SourceModel.Expr[] {
      SourceModel.Expr.Var.make(Functions.product),
      list
    }
  );
}
```


Using Java from CAL

We have seen in the previous sections how Quark gems can be used by a Java program. In this section we provide a brief introduction to how Java classes can be used by Quark programs. A much more complete treatment is given in the "CAL User Guide", included with the Quark distribution in the "docs" folder.

A simple import example is shown below:

```
data foreign unsafe import jvm public "java.lang.String"
    public String deriving Inputable, Outputable;

foreign unsafe import jvm "method length"
    public length :: String -> Int;

foreign unsafe import jvm "method substring"
    public substring :: String -> Int -> Int -> String;

halfString :: String -> String;
halfString string = substring string 0 ( (length string) / 2);
```

This imports the Java `String` class and two of its methods: `length` and `substring`. Finally, the `halfString` function, which returns the first half of a string, shows how the imported `String` methods can be used within CAL. The Java String class is immutable and its methods do not produce any side effects. This makes the `String` methods very natural CAL functions. In fact, `String` is imported by the `Prelude` module.

The next example shows how the Java standard output can be imported into CAL:

```
data foreign unsafe import jvm "java.io.PrintStream" private JPrintStream;

foreign unsafe import jvm "static field java.lang.System.out"
       private javaOut :: JPrintStream;

foreign unsafe import jvm "method java.io.PrintStream.println"
       private javaPrintLn :: JPrintStream -> String -> ();

printHelloWorld :: ();
public printHelloWorld = javaPrintLn javaOut "Hello World!";
```

This first imports the `PrintStream` class into the CAL program as the `JPrintStream` type. The next import statement imports the static `out` object from Java, which is the standard output stream. Next the `PrintStream.println` method is imported. The `printHelloWorld` function shows how the `PrintStream.println` method can be invoked on the `out` object. This will write the string "Hello World!" to the JVM's standard output.

The Java `PrintStream` methods produce side affects. They are not typical of CAL functions. It is the convention within CAL to prefix such types with a "J" to indicate this.

Advanced Topics

Implementing Inputable and Outputable

Although Quark is very well integrated with Java, the two environments do have distinct type systems. When a value crosses from one environment to the other, it must be mapped into the type system of the destination. This mapping is called an *input policy* when the value is moving from Java to Quark and an *output policy* when it is moving from Quark to Java.

In order to use the default input policy for a Quark type, that type is made an instance of the `Inputable` type class. This type class has one method:

```
public class Inputable a where
    public input :: JObject -> a;
    ;
```

Similarly, the type class `Outputable` is used by the default output policy.

```
public class Outputable a where
    public output :: a -> JObject;
    ;
```

(As you may know, `JObject` is the foreign type that corresponds to `java.lang.Object`.)

To make a user defined type inputable and outputable, you would first decide what the Java representation of the Quark type will be. The `input` and `output` methods will convert between the Quark type and an appropriate instance of this Java class.

A common approach is to define a foreign type for the target Java class, write CAL functions that convert to and from that type, using foreign functions, then convert to or from `JObject` using a foreign function that merely casts between the target Java class and `Object`.

For example, assume we have a type that represents a point with integer coordinates:

```
data public Point =
    public Point
        x :: Int
        y :: Int
;
```

We would like to make this inputable and outputable, using the class `java.awt.Point`.

We can make `Point` an instance of the `Inputable` and `Outputable` type classes like this:

```
instance Outputable Point where
    output = pointToJObject;
    ;

instance Inputable Point where
    input = jObjectToPoint;
    ;
```

We will make `java.awt.Point` accessible with some foreign declarations:

```
data foreign unsafe import jvm "java.awt.Point" JPoint
    deriving Inputable, Outputable;

foreign unsafe import jvm "constructor java.awt.Point"
    new_Point :: Int -> Int -> JPoint;
foreign unsafe import jvm "field java.awt.Point.x"
    point_getX :: JPoint -> Int;
foreign unsafe import jvm "field java.awt.Point.y"
    point_getY :: JPoint -> Int;
```

Note that `JPoint` derives `Inputable` and `Outputable`. For a foreign type this means that there are default implementations of input and output that merely cast between the class and `Object`.

The actual work of converting between `Point` and `java.awt.Point` is done by these functions:

```
outputPoint :: Point -> JPoint;
outputPoint pt =
    new_Point pt.Point.x pt.Point.y;
    ;

inputPoint :: JPoint -> Point;
inputPoint jpt =
    Point (point_getX jpt) (point_getY jpt)
    ;
```

Finally, we can define the class methods:

```
pointToJObject :: Point -> JObject;
pointToJObject !pt = output (outputPoint pt);

jObjectToPoint :: JObject -> Point;
jObjectToPoint !jpt = inputPoint (input jpt);
```

Here the invocations of `input` and `output` merely convert between `JPoint` and `JObject`.

If we define the function `magnitude` like this:

```
public magnitude pt =
    case pt of
        Point x y -> sqrt (toDouble (x * x) + toDouble (y * y));
    ;
```

…we can invoke the function from Java, passing in a `java.awt.Point`:

```
EntryPointSpec entryPointSpec = EntryPointSpec.make(
    QualifiedName.make("IOExample", "magnitude");

Object result = calServices.runFunction(entryPointSpec,
                                        new Object[] {new Point (3, 4)});
```

The `result` is a `Double` with value 5.0.

Internationalization

Quark has several features that support internationalization.

The `Locale` Type

The `Locale` type, found in the module `Locale`, represents a locale, using the same model as Java. (In fact, the `Locale` type is defined as the foreign class `java.util.Locale`.)

There are several functions in the `Locale` module to construct locales, to retrieve properties of a locale, and to retrieve standard locales.

String Localization

Quark supports localized strings. The module `StringProperties` provides the types `StringProperties` and `StringResourceBundle`.

A `StringProperties` represents a dictionary of localized strings, keyed by strings. It is implemented using the foreign class `java.util.Properties`.

A `StringResourceBundle` represents a `StringProperties` loaded from a Quark resource file.

Quark resource files are identified by a module name and a resource name. The resource loader searches for the resource file in a folder whose name is based on the module name, inside a folder called `CAL_Resources` on the classpath.

A `StringResourceBundle` can be created using the functions `makeStringResourceBundle`, `makeStringResourceBundleInLocale` or `makeStringResourceBundleWithExtensionInLocale`. All three functions take the module name and resource name. Optionally, the locale and the file extension can be specified.

The function `bundleStrings` retrieves the `StringProperties` from a `StringResourceBundle`.

The functions `lookup` and `lookupWithDefault` can be used to retrieve a localized string from a `StringProperties`. Alternatively, you can use `format0` with the `StringResourceBundle`, and bypass the call the `bundleStrings`.

Metadata Localization

Metadata on Quark entities is localized. This is discussed in the section on **Metadata**.

Data Formatting

The module `MessageFormat` contains functions that turn data values into formatted strings. There are functions such as `formatNumber` that format a single data value using the conventions for a locale. Other functions, such as `formatWithPattern`, format several data values using a pattern string. Finally, there are functions such as `format` that format several data values using a pattern string loaded from a `StringResourceBundle`.

Collation

As you probably know, the sort order of strings varies significantly from one locale to another. Thus, when sorting strings that will be displayed to the user, it's important to use locale-sensitive collation, rather than a locale-independent sorting function.

The module `Locale` provides types and functions that support collation. The function `makeCollator` and its variants create a `Collator` based on a `Locale` and, optionally, other parameters. The function `compareByCollator` compares two strings using a `Collator` and returns an `Ordering`. This function can be used with `List.sortBy` to sort a list of strings in locale-sensitive order.

`Locale` also defines the type `CollationKey`, which can be used to make collation more efficient in certain cases. This is a foreign type, wrapping `java.text.CollationKey`. Comparing collation keys is faster than comparing strings, so you should consider creating collation keys and comparing them when you need to do many comparisons between the strings in a collection.

Examples

All of these features are explored in the module `Tutorial_LocalizedStrings` in the `CAL_Samples` project.

Deploying Quark-based Applications and Libraries

There are a number of ways in which applications and libraries developed with Quark can be deployed. This section describes two commonly used approaches: deploying via *Car-jars* and deploying via *standalone JARs*.

Deploying via Car-jars

A CAL Archive – also called a *Car* for short – is an archive file format used for aggregating files which constitute CAL resources. In particular, a Car may contain:

- Workspace declaration files
- CAL source files
- cmi files (compiled module files)
- metadata files
- gem design files
- user resource files

Cars are intended to be used as a deployment mechanism: instead of deploying CAL modules as a set of source files and resource files, these modules can be packaged into a small number of Cars, so that they can be distributed and deployed more easily.

One can add Cars to the StandardVault by putting them directly on the Java classpath. These files are known as *Car-jars*, and have names ending with a .car.jar suffix. As the name implies, Car-jars employ the JAR file format.

Car-jar files can be generated using the `:car` command in ICE, using the wizard in the Gem Cutter, or using the command-line CarTool.

This approach is described in more detail in the *Using CAL Archives (Car files)* document.

Required JARs

To deploy with Car-jars, you will need the following Quark Platform JARs on your classpath:

- `Quark/bin/java/release/calUtilities.jar`
- `Quark/bin/java/release/calRuntime.jar`

- `Quark/bin/java/release/calPlatform.jar`
- `Quark/bin/java/release/calLibraries.jar`
- `Quark/bin/java/release/quarkGems.jar`
    - this is needed if you use the high-level Gem API
- all JARs `in Quark/lib/Resources/External/java/`
    - `antlr.jar, asm-all-3.0.jar, commons-collections-3.1.jar, icu4j.jar, junit.jar, log4j.jar, xercesImpl.jar, xmlParserAPIs.jar`

In addition, you will need the Car-jar files and their external Java dependencies.

If you are deploying the Gem Cutter tool as well, you will also need:

- `Quark/bin/java/release/quarkGems.jar`
    - if not already included from the list above
- `Quark/lib/Resources/GemCutterHelpFiles.jar`
- `Quark/lib/Resources/External/Sun/JavaHelp/2.0_02/jh.jar`

Deploying via Standalone JARs

If you are deploying an application or a library written in CAL, and do not require metaprogramming facilities (e.g. running pieces of CAL expressions from within Java, adding new modules at runtime, querying for functions matching a particular type), you may want to consider using *standalone JARs* as the deployment mechanism.

A standalone JAR may package up a CAL application by gathering together all the classes necessary for running a specific CAL function, and a class whose `public static void main(String[] args)` method runs the CAL function with the supplied command-line arguments. You can package up a CAL function of the type `[String] -> ()` into a standalone application JAR.

The generated main class is able to run the CAL function directly without having to first initialize a CAL workspace. Thus, the start-up time to run the function will be much smaller than in cases where workspace initialization is required (e.g. running the function via `BasicCALServices` from within a Java program.)

A standalone JAR may also package up one or more *library classes* – a library class is a non-instantiatable class containing static methods corresponding to the non-private functions and data constructors defined in a particular CAL module.

This makes it possible to expose CAL libraries in Java, by defining API modules in CAL (whose functions may include code for marshalling to/from foreign types), from which library classes are generated.

To build a standalone JAR, you can use the command-line Standalone JAR Tool, or the `:jar` command in ICE. A standalone JAR produced by this facility includes only the classes necessary for running the specified CAL applications and supporting the specified CAL library modules, and thus does not take up more space than necessary.

User resources are also bundled in the generated standalone JAR, so that the CAL functions may load and use localized resources (such as string properties files) via modules such as `Cal.Core.Resource` and `Cal.Utilities.StringProperties`.

The tool also supports the generation of a companion source zip file containing source files for all generated classes. One can use the zip file in an IDE to link the class files in the JAR to sources, or build a version of the JAR with debug information (e.g. line numbers) from the sources for debugging purposes. Moreover, the library classes will have proper Javadoc comments generated from CALDoc comments. Thus it is possible to use these source files with the *javadoc* tool to produce API documentation for the standalone libraries.

This approach is described in more detail in the *Using Quark with Standalone JARs* document.

Required JARs

When deploying with standalone JARs, the required set of JARs is:

- `Quark/bin/java/release/calUtilities.jar`
- `Quark/bin/java/release/calRuntime.jar`
- `Quark/bin/java/release/calLibraries.jar`
- `Quark/lib/Resources/External/java/icu4j.jar`
- `Quark/lib/Resources/External/java/log4j.jar`
- the generated standalone JAR, and its external Java dependencies

To assist in building up this classpath, the Open Quark distribution includes a `quarklaunch` script that makes it easy to run an application packaged as a standalone JAR. Documentation for the script is included in the aforementioned document.

Appendix A: Quark Module Storage

This appendix describes the mechanics and currently supported use patterns of Quark workspaces. A description is provided of the principles involved and their interactions.

Workspace

A workspace is a logical "location" in which the working copies of Quark resources are gathered. When CAL source files are compiled, or metadata are loaded, the corresponding files are loaded from the workspace.

There are two types of workspace:

Nullary

- This is the default workspace configuration.
- Only one nullary workspace exists.
- The location of the workspace coincides with the location of the StandardVault.
- Only modules and CAL archives (Cars) from the StandardVault may be specified in the workspace declaration.
- The workspace reads directly from and writes directly to the resources in the StandardVault.

Discrete

- This is an optional workspace configuration.
- Any number of discrete workspaces may exist. Each discrete workspace is uniquely identified by an identifier string.
- Modules from any vault may be specified in the workspace declaration.
- Cars from the StandardVault, the EnterpriseVault, and the SimpleCarFile vault may be specified in the workspace declaration.
- The workspace is a working copy of the resources specified in the workspace file, i.e. this is the standard sandbox model. It is intended to support versioning and configuration management, though this work is incomplete.
- The initial workspace is initialized with the resources specified in the workspace declaration. Subsequent updates to files specified in the vaults are propagated via sync operations; they do not happen automatically. Changes

made to the workspace can be propagated back to vaults via the appropriate export operation.

- A client specifying an identifier for an existing discrete workspace on startup will load that workspace. The workspace declaration will not be consulted.

Workspace Declarations

The contents of a workspace are specified using a workspace declaration. On the local file system these will have the file extension ".cws".

A workspace declaration consists of a series of statements, each of which can specify either a module, a CAL archive (Car), or another workspace declaration. The contents of the resulting workspace will be the specified modules, plus any modules from the specified workspace declarations and Cars.

Syntax

Module:

```
VaultType ModuleName [LocationString [RevisionNumber]]
```

Car:

```
import car VaultType CarName [LocationString [RevisionNumber]] CarspecName
```

where CarspecName is the name of the Car Workspace Spec file located within the Car

Workspace:

```
import VaultType WorkspaceName [LocationString [RevisionNumber]]
```

where LocationString is a vault-specific identifier

Module precedence is textual order. For example, suppose a declaration contains the line "VaultA ModuleName", and an imported declaration contains the line "VaultB ModuleName". If "VaultA ModuleName" appears before the import, the VaultA module is used. If it appears after the import, the VaultB module is used.

Vaults

A vault represents a repository for CAL resources.

Three types of elements may be stored: a workspace declaration, a CAL archive (Car), or a persisted module.

There are currently five available vault types:

StandardVault

- In this vault, resources are located with respect to folders on the current classpath.

- CAL source files are found by looking on the classpath for subfolders named "CAL", workspace declarations are found in subfolders named "Workspace Declarations", etc.

SimpleCALFile

- This vault represents a single CAL source file in the local file system. The location string is the file URL. e.g. "file:///C:/dev/cal/Prelude.cal"

SimpleCarFile

- This vault represents a single Car file in the local file system. The location string is the file URL. e.g. "file:///C:/dev/Car/everything.car"

EnterpriseVault

- The EnterpriseVault is not supported in this distribution of the Quark Platform.

- This is a versioning vault where persisted modules are stored as objects in the BusinessObjects Enterprise repository. The location string is the name of the Enterprise CMS.

- Through the EnterpriseVault, the Quark Platform provides the ability to store consistent modules centrally and to distribute and deploy these pieces of logic to applications.

JarVault

- This vault represents a single module whose resources have been combined into a single read-only Jar file. The location string is the Jar file URL. e.g. file:///C:/dev/cal/Prelude.jar

