

# **Business Objects Gem Cutter Manual**

**Luke Evans  
Neil Corkum**

---

# Business Objects Gem Cutter Manual

by Luke Evans and Neil Corkum

Last modified: December 19, 2006

Copyright (c) 2007 BUSINESS OBJECTS SOFTWARE LIMITED  
All rights reserved.

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.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---

---

---

# Table of Contents

Welcome .....	viii
Chapter 1. The Anatomy of a Gem .....	1
1. Gems .....	1
2. Basic Representation of a Gem .....	1
3. Data Types .....	2
4. Gem Composition .....	4
Chapter 2. The Anatomy of the Gem Cutter .....	7
Chapter 3. Where Do Gems Live? .....	9
1. Modules and Workspaces .....	9
2. How Do I Find Gems? The Gem Browser .....	9
2.1. Browser View Quick Access Panel .....	11
Chapter 4. Defining a New Gem .....	13
1. Getting Help Defining a New Gem: IntelliCut .....	14
Chapter 5. Testing a New Gem .....	18
1. Viewing the Gem Definition .....	24
Chapter 6. Burning Inputs .....	25
Chapter 7. Special Gems .....	30
1. The Value Gem .....	30
2. The Code Gem .....	36
3. Collectors and Emitters .....	48
3.1. Creating Local Functions .....	51
4. The Record Creation Gem .....	54
5. The Record Field Selection Gem .....	55
Chapter 8. The Properties Browser .....	56
1. Viewing Gem and Module Metadata .....	56
2. Editing Metadata .....	58
Chapter 9. Miscellaneous Features .....	69
1. Renaming Entities .....	69
2. Generating Enumerations .....	69
3. Creating a New Minimal Workspace Declaration .....	71
4. Searching Module Code .....	73
5. Generating a JDBC Data Source .....	75
Chapter 10. Using the Documentation Generator in the Gem Cutter .....	78
1. CALDoc Generation Options .....	80
Chapter 11. Generating Gems From Java Code .....	83
1. Creating a CAL Type From a Java Class .....	83
2. Importing Java Methods and Fields .....	86
3. Generating a Foreign Import Module .....	88
3.1. Using the Java Foreign Input Factory (JFit) in the Gem Cutter .....	88
Chapter 12. Understanding Type Expressions .....	92
Glossary .....	95

---

## List of Figures

1.1. equals Gem .....	1
1.2. equals Gem with tooltip displayed .....	2
1.3. sin Gem .....	2
1.4. equals and sin Gems .....	2
1.5. Output tooltips for equals and sin Gems .....	3
1.6. head Gem - a polymorphic Gem .....	4
1.7. take Gem .....	4
1.8. sum Gem .....	5
1.9. take Gem being connected to sum Gem .....	5
1.10. take Gem connected to sum Gem .....	6
2.1. Interface of the Gem Cutter .....	8
3.1. The Gem Browser .....	10
3.2. The Quick Access Panel .....	11
4.1. A newly defined Gem in the Gem Cutter .....	13
4.2. IntelliCut in action .....	15
4.3. IntelliCut suggesting valid connections between Gems .....	16
4.4. IntelliCut Preferences panel .....	17
5.1. A sample Gem .....	18
5.2. Cannot Execute Gem dialog .....	18
5.3. Testing a Gem - Value Entry panels .....	19
5.4. Value Entry panel for sum Gem .....	20
5.5. List Value Editor .....	21
5.6. Type selection panel .....	21
5.7. List Value Editor with Double values input .....	22
5.8. Calculator panel .....	22
5.9. Gem Results panel .....	23
5.10. Toolbar buttons when testing a Gem .....	23
5.11. List Value Editor for Ints .....	24
6.1. any Gem with tooltips displayed .....	25
6.2. greaterThan Gem with tooltips displayed .....	25
6.3. pi Gem connected to greaterThan Gem .....	26
6.4. Predicate function created by input burning .....	26
6.5. Predicate function connected to the any Gem .....	27
6.6. Testing a Gem with input burning .....	27
6.7. Automatic input burning .....	28
6.8. The Gem Cutter cannot determine which input to burn .....	28
6.9. IntelliCut showing Gems that can be connected with input burning .....	29
7.1. Special Gems Toolbar buttons .....	30
7.2. Value Gem with sections labelled .....	31
7.3. Value Gem type selection panel .....	31
7.4. Value Gem representing the Time type .....	31
7.5. Unspecialized Value Gem .....	32
7.6. Selecting the record type for a Value Gem .....	33
7.7. Value Gem representing a record type .....	33
7.8. Record editor .....	33
7.9. Selecting a type for a record component .....	34

7.10. Record containing an Int, Date and Color value .....	34
7.11. Record after renaming the Date type field .....	34
7.12. Changing the Color field .....	35
7.13. Record with Color changed to red .....	35
7.14. filter Gem connected to result target .....	36
7.15. filter Gem with tooltip .....	36
7.16. New Code Gem added to Table Top .....	37
7.17. Code Gem Editor .....	37
7.18. Code Gem Editor containing CAL code .....	38
7.19. Gem to filter elements greater than 1000 from a list .....	39
7.20. Code Gem Editor containing incorrect code .....	39
7.21. Code Gem Editor with tooltip describing code error .....	40
7.22. Code Gem with tooltips .....	40
7.23. Two add Gems connected to a Code Gem .....	41
7.24. Code Gem Editor with three parameters .....	41
7.25. Code Gem with added third argument .....	42
7.26. Code Gem Editor after removing <i>secondArg</i> .....	42
7.27. Argument type clash in the Code Gem Editor .....	43
7.28. Argument type clash error tooltip .....	43
7.29. Output type clash in the Code Gem Editor .....	44
7.30. Code Gem Editor before argument reordering .....	44
7.31. Arguments being reordered in the Code Gem Editor .....	45
7.32. Code Gem Editor after argument reordering .....	45
7.33. average Code Gem and editor .....	46
7.34. average Code Gem after changing function name to argument .....	47
7.35. Code completion in the Code Gem Editor .....	47
7.36. Renaming a variable in the Code Gem Editor .....	48
7.37. Code Gem Editor after renaming a variable .....	48
7.38. Gems on Table Top .....	49
7.39. Table Top with Collector Gem added .....	50
7.40. Emitter Gems connected on Table Top .....	50
7.41. Completed filtering Gem layout .....	51
7.42. Modified filtering Gem layout .....	52
7.43. Modified filtering Gem layout with filterFunc collector .....	53
7.44. filterFunc Emitter Gem .....	53
7.45. Completed modified filtering Gem using local function .....	54
7.46. Record Creation Gem .....	54
7.47. Creation of a record with "name" and "age" fields .....	55
7.48. Record Field Selection Gem .....	55
8.1. The Properties Browser .....	57
8.2. Properties Browser toolbar .....	58
8.3. Section of Properties Browser navigation bar .....	58
8.4. nChooseR Gem with default tooltips .....	59
8.5. Properties Browser in edit mode .....	60
8.6. <b>Basic Properties</b> panel .....	61
8.7. <b>Basic Properties</b> panel with metadata entered .....	62
8.8. <b>Gem Properties</b> panel .....	63
8.9. <b>Gem Arguments and Return Value</b> panel .....	63
8.10. <b>Gem Arguments and Return Value</b> panel with metadata entered .....	64

8.11. <b>Usage Examples</b> panel .....	64
8.12. <b>Usage Examples</b> panel with new example .....	65
8.13. <b>Usage Examples</b> panel with completed example .....	65
8.14. Metadata for the <code>nChooseR</code> Gem .....	67
8.15. <code>nChooseR</code> Gem with new tooltips .....	68
9.1. <b>Generate Enumeration</b> dialog box .....	69
9.2. Generating a <code>Season</code> enumeration .....	70
9.3. <code>Season</code> enumeration in the Gem Browser .....	70
9.4. <code>Spring</code> data constructor .....	71
9.5. Value Gem representing the <code>Season</code> data type .....	71
9.6. <b>Create Minimal Workspace Declaration</b> dialog box .....	72
9.7. Creating a minimal workspace including the <code>List</code> module .....	73
9.8. New minimal workspace in the Gem Browser .....	73
9.9. <b>Search</b> dialog box .....	74
9.10. Using the <b>Search</b> dialog box .....	75
9.11. Example Data Connection Gem Design .....	76
9.12. <b>Generate JDBC Resultset Gem</b> dialog box .....	76
10.1. <b>Generate CALDoc Documentation</b> dialog box .....	79
10.2. <b>Output Directory</b> section .....	80
10.3. <b>Scope</b> section .....	80
10.4. Filtering options section .....	80
11.1. Generate Java Data Type dialog box .....	83
11.2. Selecting a class to import .....	84
11.3. Selecting the <code>java.util.Stack</code> class .....	84
11.4. Selecting a constructor and adding a comment to an imported class .....	85
11.5. Gem Browser with newly created <code>makeJStack</code> Gem .....	85
11.6. <code>makeJStack</code> Gem .....	85
11.7. Generate Java Method or Field dialog box .....	86
11.8. Importing the <code>empty()</code> method from <code>java.util.Stack</code> .....	87
11.9. Gem Browser with newly created <code>jUtilStack_empty</code> Gem .....	87
11.10. <code>jUtilStack_empty</code> Gem .....	87
11.11. Testing the <code>jUtilStack_empty</code> Gem .....	88
11.12. Generate Java Foreign Import Module dialog box .....	89

---

# Welcome

Welcome to Business Objects *Gems*. Business Objects Gems are a new way of describing your data and business logic. They are designed to make creating the data access and analysis components of your application remarkably easy, and their component nature allows for publishing libraries of Gems for use across multiple applications.

Business Objects Gems are powerful components which hide most of the details about the originating data and analytic algorithms which get value from it. They can be used singularly or combined together powerfully, to create abstractions which can describe data filtering, consolidation or a full-blown business model.

Out of the box, you will already have a very powerful suite of Gems which can be applied straight away to data. These libraries can be used to explore the data with our Gem enabled tools. However, you will normally have reason to refine your business logic by deriving new Gems from the default set, and you can do this easily by using the *Business Objects Gem Cutter*.

The Business Objects Gem Cutter is the design tool for Gems, providing all the tools you need to experiment, build new Gems and test them with real data.

The design emphasis of the Gem Cutter was to create an environment in which the developer can develop business logic much more organically than traditional approaches. You are encouraged to try things out and to learn by experimentation and discovery. The Gem Cutter will only allow the creation of 'legal' Gems which at least have a meaningful relationship to the data and to each other. The environment promotes a Rapid Application Development approach to business logic and we think you'll find it an extremely productive way to develop your application's data manipulation logic.



---

# Chapter 1. The Anatomy of a Gem

## 1. Gems

We'll start by talking about what a *Gem* actually is and what benefits they provide.

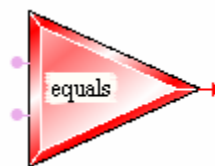
A Gem is simply a function. A function is a mathematical concept that takes one set and 'maps' it onto another set. Simply put, this means that a Gem takes an input and produces an output. Functions are nothing new in computing of course, and almost every language has some sort of function concept. However, in the Business Objects Gems framework, everything is a function (even the data!). This keeps everything extremely simple and well-ordered.

At the point of consumption in compatible tools, a Gem is simply a named function which can be applied to data, can define the content of a field, or can be executed to perform some transformation to the data. Doing this simply involves picking the Gem from a browser or using the Intellicut feature (see *Section 1, "Getting Help Defining a New Gem: IntelliCut"*) to find and apply a Gem for the correct function.

At design-time, the true nature of a Gem is exposed in order to permit editing and abstraction. We will begin to introduce the visual symbology used in the Gem Cutter as we explain the nature of Gems.

## 2. Basic Representation of a Gem

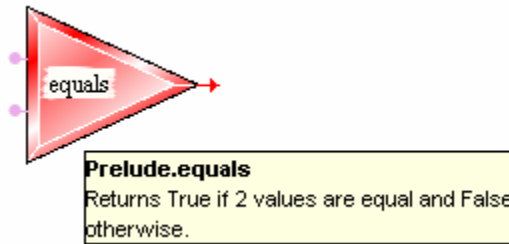
We can represent a Gem (a function) graphically, as an object which consumes some data and produces a result. A simple example is displayed in *Figure 1.1*:



**Figure 1.1. equals Gem**

We use a left-to-right reading of the function application, and we can see two light pink inputs (with circular connectors) and a red output (with an arrow connector). This Gem (called `equals`) takes two inputs and produces a result.

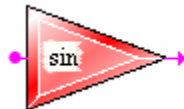
An easy way to get more information on the purpose of a Gem is to place your mouse pointer over the Gem and read the tooltip which appears. *Figure 1.2* gives an example.



**Figure 1.2. equals Gem with tooltip displayed**

The tooltip provides us with a brief description of the function of the Gem. In the case of the `equals` Gem, the result produced will be `True` if the input values are equal to each other, and `False` otherwise.

Figure 1.3 shows another Gem, the `sin` Gem.



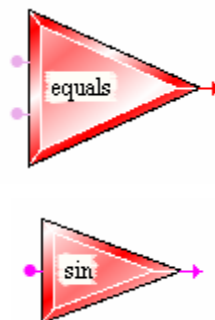
**Figure 1.3. sin Gem**

Notice that the colours of the input and output are different on the `sin` Gem from those on the `equals` Gem. This is because colour is used as one of the indicators of type. We'll now talk a bit about the concept of type.

### 3. Data Types

*Type* is a hugely important concept in Gems. Type is the way in which we can check if two Gems are compatible and can be meaningfully connected.

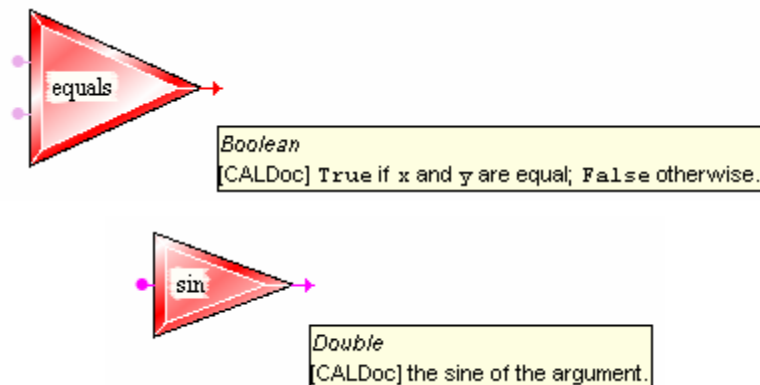
Looking at the two Gems in Figure 1.4:



**Figure 1.4. equals and sin Gems**

Notice that the colours of the Gem inputs and outputs are different. Each input and output with the same type will appear as the same colour. So, just from looking at these two Gems, it is clear that the two inputs to the `equals` Gem are of the same type, while the output is a different type. The input and output of the `sin` Gem are the same type, but this type differs from either of the types present on the `equals` Gem.

Colour is really only a secondary hint about type however. To get specific information on a type, you can look up the Gem's reference description (which includes notes about its intended behaviour, plus a description of types) by right-clicking on the Gem and selecting **View Properties...** from the popup menu. You can also roll-over the connectors with the mouse, which will show you the types, as well as some additional information, in a tooltip, such as in *Figure 1.5*:



**Figure 1.5. Output tooltips for `equals` and `sin` Gems**

Notice that in this case, bright pink is representing a `Double` (double precision, floating point number) type and red is the colour used for a `Boolean` (boolean value) type.

We can therefore see that the `sin` Gem will take a `Double` and return a `Double` (the sine of the input value) and the `equals` Gem will return `True` or `False` (depending on whether the two input values are equal).

This is fairly straight-forward, and nothing that you wouldn't see in a traditional language. At this level, it is easy to see how we can arrange for the Gem Cutter to only permit an output of a Gem to connect to an input of another Gem if (and only if) the types are the same.

The problem with this 'monomorphic typing' approach is that there has to be a Gem with a particular type configuration that performs the function required. This would lead to an explosion in the number of Gems that would have to be provided in order to cope with the large variety of types that are present in real systems. Instead, the type system supports what is known as *polymorphism*. This means we can describe Gems whose types will be compatible with a set of other types.

*Figure 1.6* is an example of a polymorphic Gem:

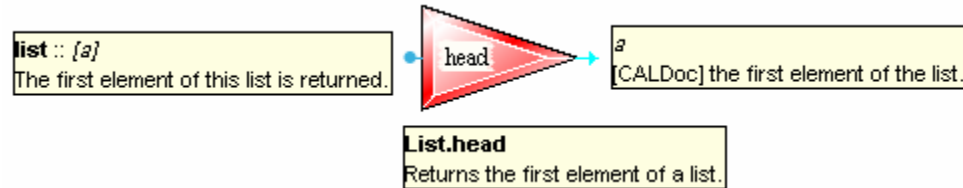


Figure 1.6. head Gem - a polymorphic Gem

Figure 1.6 shows tooltips for the input and output, as well as the Gem itself. These seem a little more cryptic than the previous examples, but they are still very straightforward. The function being shown (`head`) returns the first item in a list. The input is the list, and a *type expression* with square brackets denotes a list of something, where the 'something' is shown inside the brackets. In this case, we can see that the 'something' is an 'a'. This seems like a daft name for a type, and in fact it doesn't represent a type at all (which would in any case start with an uppercase letter). Actually, this is a *type variable* and means 'anything'. Bearing this in mind, we can see that the `head` Gem takes a list of 'anythings' and returns an 'anything'. The fact that the same letter is used as the type variable tells us that whatever 'a' becomes will be consistent in both the input and output. In other words, if we use `head` on a list of double precision numbers (denoted: '[Double]'), we would expect the return type to be 'Double'. The feature of using type variables in the type expressions is called *parametric typing*.

## 4. Gem Composition

The act of creating a new Gem from a collection of other Gems is called 'composition'. To compose a Gem in the Gem Cutter, you simply drag the output of a 'source Gem' to an input on a 'destination' Gem. Actually, the concepts of source and destination are a little false, because what you end up doing is creating a new function whose meaning is a perfect combination of the two Gems (i.e. there's no 'first' and 'second' really).

Here's what this looks like in the Gem Cutter. We'll be composing a function (possibly a new Gem) which will add up the first *n* elements of a list. To do this, we first need a function which can extract the first *n* elements. This happens to exist in the standard library of Gems, and is called `take`. Figure 1.7 shows the `take` Gem:

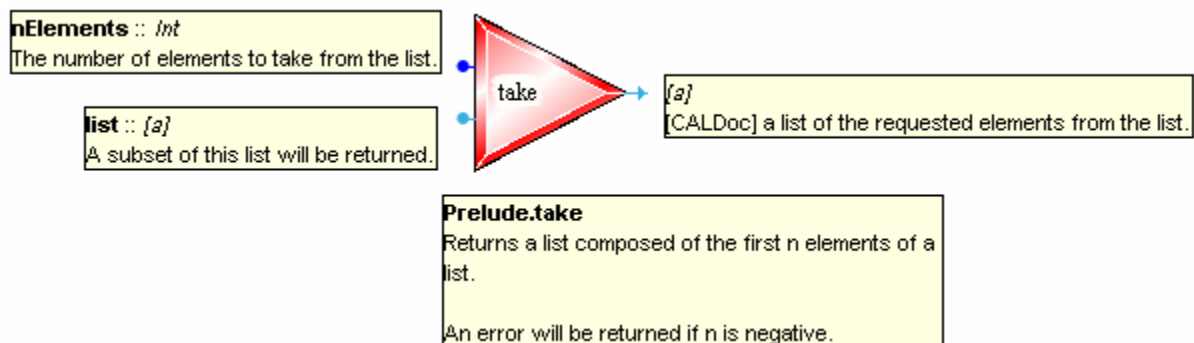


Figure 1.7. take Gem

In words, `take` requires a list of 'anythings' and an `Int` (integer) representing the number of elements to take from the list. It will provide a list of anythings (the list of the first `n` elements).

We also need a Gem that can sum a list. Again, we find a standard library Gem to do this, called `sum`. Its graphical representation is displayed in *Figure 1.8*:

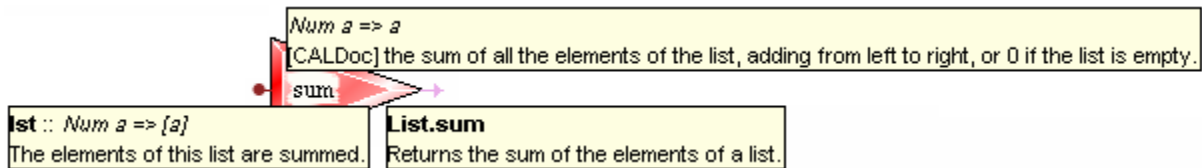


Figure 1.8. `sum` Gem

This Gem takes a list of numbers and returns a number (the total).

We want to compose a function that will take the first elements of a list and then sum them, so we know we need to attach the output of `take` into the input of `sum`. Notice that we can tell this is possible in two ways:

Firstly, the input type of `sum` is a more specific type of the output of `take`. The type `Num a => [a]` (a list of 'anythings', where 'anything' is a number value) is clearly a more specific version of `[a]`, which is the general list of 'anythings' (see the section *Chapter 12, Understanding Type Expressions* for more information). So we can infer that the Gems are connectible in the way we want. Technically, we say that the types are *unifiable*.

Secondly, we can experiment by trying to connect the Gems in the way we want. This is achieved by dragging a link from the output of `take` to the input of `sum`. *Figure 1.9* shows how it looks in the Gem Cutter:

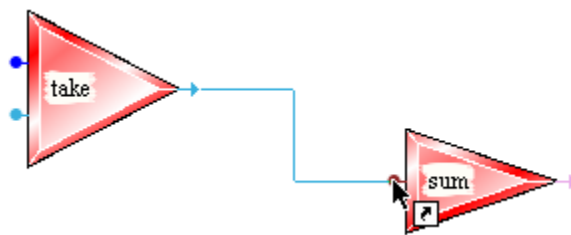
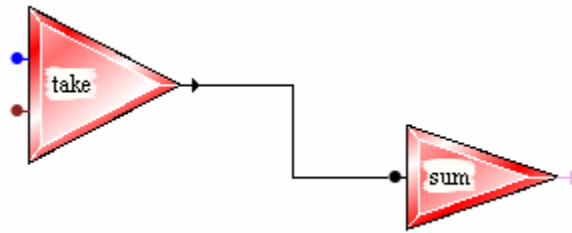


Figure 1.9. `take` Gem being connected to `sum` Gem

A light blue line extends from the light blue output of `take` to the desired input on `sum`. As the mouse nears the input, the cursor will change to one of two icons, to indicate either that the connection is possible (the types are compatible), or that this connection would be illegal. It is impossible to form an illegal connection between two Gems.

In this case, the types are compatible, and when we release the mouse button, the connectors involved turn black and a connecting line is drawn between the Gems, as shown in *Figure 1.9*:



**Figure 1.10. take Gem connected to sum Gem**

The change to black indicates that the two Gems are now bound. In fact, although the distinction between the two Gems is preserved, we have created a new function object, and in effect there is no longer a concept of an output on `take` and an input on `sum`. Therefore, there are no types associated with these any more, resulting in the black colouration.

An interesting point to note about the resulting graphical representation of the combination we just made is that the second input to `take` has changed colour to indicate that its type has changed. In fact it has changed to the same colour as the original input of `sum`. What has happened is that `take` 'realises' that it is required to produce a list of numbers as output (by being bound to `sum` which has this as an input requirement). This implies that it, in turn, must collect a list of numbers (instead of any list) in order to satisfy this requirement. It therefore advertises this fact with a change in its input type.

Another point of note is that a Gem output does not have to be the same colour as the input of another Gem for these two Gems to be connectible. In the previous example, the output of `take` is light blue, while the input of `sum` is brown. These Gems can still be connected because the input to `sum` is just a more specific type than the output of `take`. An output on one Gem coloured the same as the input on another Gem guarantees that these two Gem connectors can be connected, but Gem connectors with different colours may also be connectible if there is a type compatible with both connectors.

---

# Chapter 2. The Anatomy of the Gem Cutter

The Gem Cutter is the design-time environment for creating general purpose Gems. The Gem Cutter is designed for large scale development of libraries of Gems for general consumption by other tools.

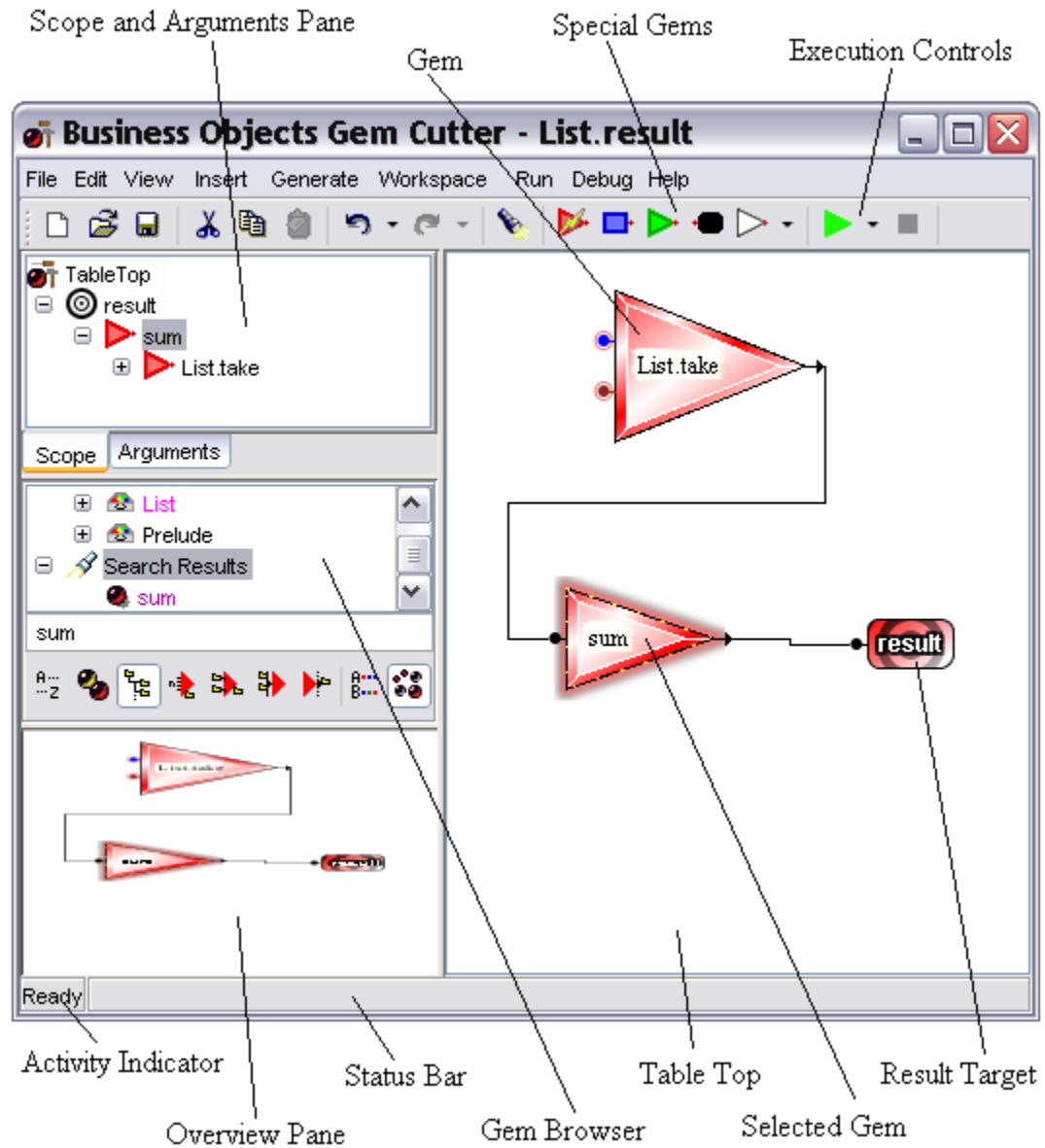
Note that when using the Gem Cutter to create Gems, you are essentially writing CAL code. Gems in the Gem Cutter are graphical representations of CAL functions. By default, Gems are saved into the `GemCutterSaveModule.cal` file, so looking in this file will show you the CAL code written by the Gem Cutter for any Gems you have saved here.

The Gem Cutter is simple in essence, being composed of two main sections:

The Sidebar	This area of the Gem Cutter contains the Gem Browser window, as well as windows providing information related to the current state of the Table Top. These additional windows, such as the Overview and the Argument Tree, can be switched on and off via the <b>View</b> menu.
The Table Top	This is where most of the activity is. Here, you will work with component Gems, composing them together to form a new Gem.

There is also a toolbar and a status bar.

*Figure 2.1* shows the main features of the Gem Cutter's graphical interface:



**Figure 2.1. Interface of the Gem Cutter**

The specific functions of each piece will be described elsewhere in this manual.



---

# Chapter 3. Where Do Gems Live?

Gems are conceptually stored in Vaults which are ultimately some form of physical storage mechanism (like a file or database). At this time, the Gem Cutter only supports one Vault – the standard vault. All Gem definitions are stored in the standard vault, which is simply a location in your file system containing CAL files. Each of these CAL files is referred to as a *Module*.

## 1. Modules and Workspaces

Modules are a system of organization for Gems. Each CAL file in a vault represents one module, and may contain one or more Gem definitions.

To use modules in the Gem Cutter, they must be present in the current workspace. A workspace represents the range of Gems being worked with at any given time, and contains one or more CAL modules. Together, the modules in the workspace contain all the Gems that are currently available for use in the Gem Cutter.

To add a module to the current workspace from the standard vault, the **Workspace → Add Module → From Standard Vault** menu option should be used. A dialog will then appear, allowing you to select which module to add to the workspace. The **Workspace → Add Module → Imported from Local File** menu option can be used to copy an existing CAL file from anywhere on the file system to the standard vault, allowing its use as a module in the Gem Cutter. Note that this will only work if the module being loaded is not already present in the standard vault.

To remove a module from the current workspace, use the **Workspace → Remove Module** menu option. This will display a dialog allowing you to select the module you would like to remove from the workspace.

The current module has its name displayed in bright pink text in the *Gem Browser* (see *Section 2, “How Do I Find Gems? The Gem Browser”*). This is the module that any Gems created by the user will be saved in. To change the current module, open the context menu for the module you would like to be the new current module, and select the **Change to this Module** option.

It is also possible to switch between different workspaces. The **File → Switch Workspace** menu option will allow you to select a different workspace declaration file to load a workspace from. Alternatively, you can select one or more modules from the standard vault to form the basis of a new workspace. Note that certain modules will be included in the workspace if selected modules depend on them. For example, selecting to include the `Math` module will also include the `Prelude` module, because the `Math` module depends on functionality contained in the `Prelude` module.

## 2. How Do I Find Gems? The Gem Browser

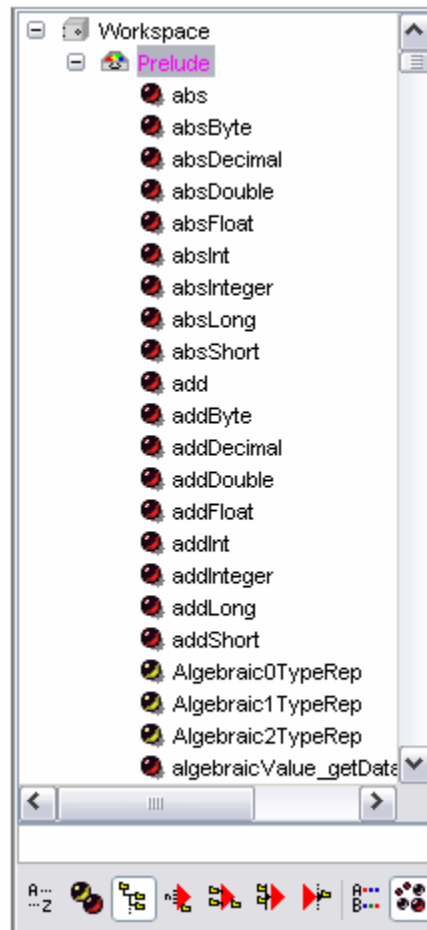
The *Gem Browser* is a component of the Gem Cutter that helps the user easily locate and select Gems. The Gem Browser will open the default workspace by default (which includes all the primitive Gems and data types), and will allow the user to locate and add new modules to its outline view to explore these other Gem repositories.

The Gem Browser is an outline control which displays the modules in the workspace. Inside the Drawers are the Gems themselves. Gems which display a yellow/orange icon (🟡) and have capitalised names are called Data Constructors (constructors for short). Gems which display a red icon (🔴) are Functions.

## Note

For more information on Data Constructors, please refer to the *CAL User's Guide* document.

*Figure 3.1* is a picture of the browser with the Prelude module open (where many basic, useful Gems are stored):



**Figure 3.1. The Gem Browser**

The Gem Browser allows the user to view the available Gems sorted in various ways, in order to facilitate finding Gems for a specific purpose. The following orderings are possible:

Sort Alphabetically

Gems are sorted by name in lexicographical order.

Sort Data Constructors then Functions

Gems are sorted by kind (yellow constructors, red functions), and then by lexicographical order.

Group by Module	Gems are categorised according to their containing module.
Group by Arity	Gems are categorised into folders by the maximum number of arguments they will consume. This is the number of input connectors visible on the Gem when it is on the Table Top.
Group by Gem Type	Gems are categorised into folders according to their type. This is useful for finding Gems that use certain kinds of data to produce specific kinds of data.
Group by Input Types	Gems are categorised into folders according to their input types. This is useful for finding Gems that operate on certain kinds of data.
Group by Output Type	Gems are categorised into folders according to their output type. This is useful for finding a Gem which will produce a certain kind of data.

You can switch between the above views by using the context menu on a Gem module in the browser window. The effects of the ordering can be performed at the module level, on an existing grouping, or at the workspace level. Applying an ordering at the workspace level will transcend the module organisation of the Gems and enable you to see sorted Gems 'across' several modules. This is very useful when you know something about a Gem but can't remember where it is stored.

To search the workspace, enter a string to search for in the text field below the pane containing the workspace tree. All Gems containing the string searched for will appear under the **Search Results** tree node in the workspace tree pane.

## 2.1. Browser View Quick Access Panel




For convenience a 'quick access' panel is displayed at the bottom of the Browser's outline view in order to quickly switch between Gem views at the workspace level. In order to perform sorting at the module level, the context menu must be used on the appropriate module in the Gem Browser.


The Quick Access Panel is displayed in *Figure 3.2*:




**Figure 3.2. The Quick Access Panel**

From left to right, the buttons in the left section of the panel perform:

-  Sort Alphabetically
-  Sort Data Constructors then Functions
-  Group by Module

 Group by Arity

 Group by Gem Type


 Group by Input Type

 Group by Output Type


The two buttons on the right side of the Quick Access Panel are used to show or hide additional information in the Gem Browser. These two buttons (from left to right) have the following functionality:

 Display Gem Type

Displays the type of each Gem next to its name in the Gem Browser.

 Display Gems from Unimported Modules

Displays Gems from unimported modules as well as those from currently imported modules.

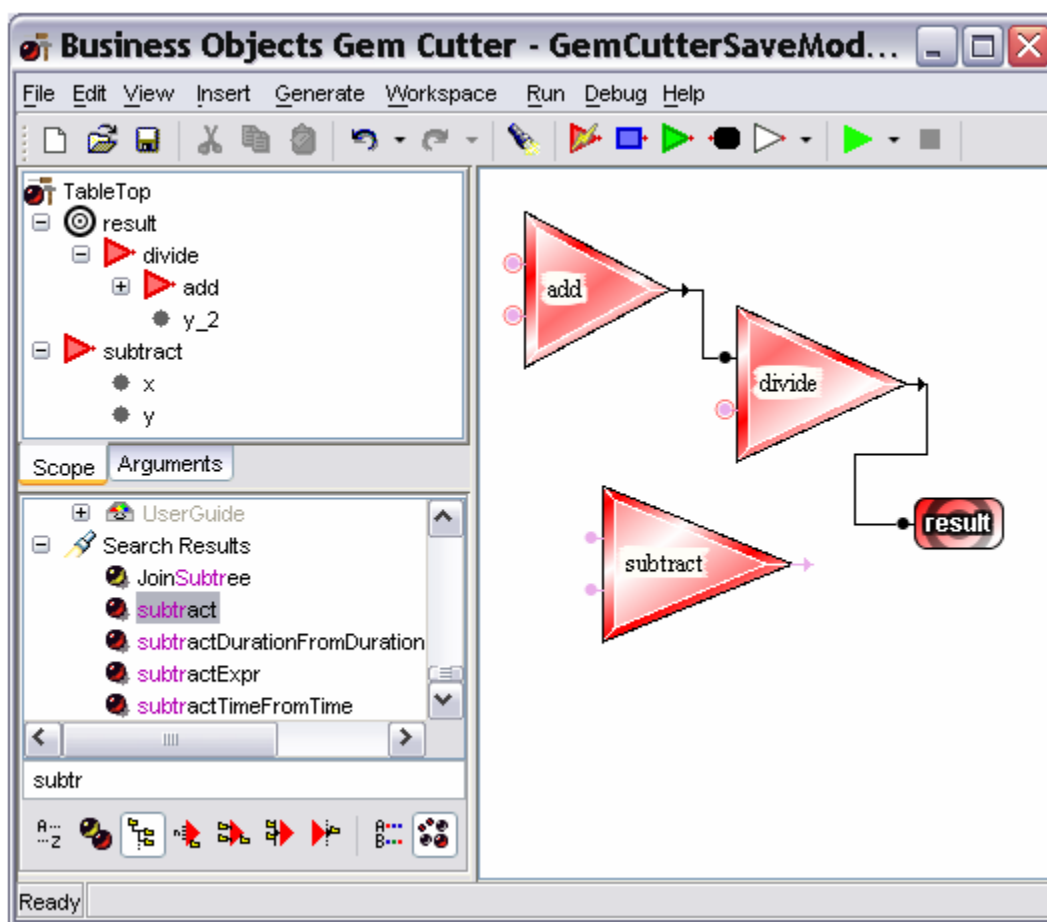
Gems are generally accessed by dragging and dropping the required Gem from the Gem Browser pane into the position in the environment where the Gem is required. In the Gem Cutter, Gems can be dropped onto the Table Top. Alternatively, the **Add Gem** () button on the Toolbar can be pressed, followed by clicking on the Table Top and selecting a Gem from the menu that appears (see *Section 1, "Getting Help Defining a New Gem: IntelliCut"*). When this is done, a representation of the Gem is drawn on the Table Top, showing an output connector, and any input connectors.

# Chapter 4. Defining a New Gem

In order to define a new Gem, the components from which the new Gem will be assembled must be collected onto the Table Top and composed together. As well as using Gems from a Vault, a variety of 'special' Gems are available on the toolbar which provide extra functionality. These are dealt with later in *Chapter 7, Special Gems*.

Gems are composed together to form the required function. Any inputs left unbound will automatically be treated as inputs to the newly created Gem. Once composition is complete, the new Gem is 'defined' by connecting the final output to a target. It is possible to have more Gems on the Table Top than are currently composed into a new Gem. Any Gems which are not connected into the 'tree' of Gems defined as having its root at the target are not considered a part of the new Gem. However, these Gems will still be saved as part of the Gem design if the Gem on the Table Top is saved.


Figure 4.1 shows an example of a new Gem defined in the Gem Cutter:



**Figure 4.1. A newly defined Gem in the Gem Cutter**

In this example, the `add` and `divide` Gems from the standard library have been composed together to form a new function. The output of this function has been connected to a target which

identifies this composition as being a new Gem. Notice that the `subtract` Gem is playing no part in the `add-divide` composition and is therefore not part of the Gem under construction.

At this point, the user may test the Gem using the green **Play** () button in the toolbar, or name the Gem and save it to a Vault.


Sometimes it is necessary to select Gems in order to target them for a certain operation. This can be achieved by clicking Gems in the following ways:

- Clicking on a Gem to select it alone. This will deselect already selected Gems.
- Holding a **Control** key and clicking on a Gem. This will 'toggle' the selection of the clicked Gem, whilst preserving the selection states of other Gems on the Table Top.

You can also select multiple Gems by dragging a rectangular outline around a set of Gems on the Table Top. Using this technique, selection can be performed in the following ways:

- Dragging an outline to completely enclose a set of Gems will select only these Gems. This will deselect already selected Gems not fully enclosed by the rectangular outline. The rectangular outline will appear solid.
- Holding a **Shift** key and dragging an outline around some Gems. This will preserve the selection of Gems not enclosed by the outline, and will select Gems enclosed by the rectangle.
- Holding a **Control** key and dragging an outline around some Gems. This will preserve the selection of Gems not enclosed by the outline, and will 'toggle' the selection of Gems enclosed by the rectangle.

Gems can be deleted from the Table Top by pressing the **Delete** key, selecting the **Edit → Delete** menu option, or by choosing the **Delete Gem(s)** option from the Gem context menu (by right clicking on a Gem).

To completely clear the Table Top (and start over), you can use the **File → New** menu option, or click the **New Table Top** () button on the toolbar.

To tidy the appearance of the layout of Gems on the Table Top, select the **View → Tidy Table Top** option from the menu bar, or press **Ctrl+G**. If tidying the appearance of only certain Gems is desired, select the Gems to be tidied, right-click to open a context menu and select the **Tidy Selection** option.

To attempt to fit the Gem layout into a smaller area on the Table Top, select the **View → Fit Table Top** option from the menu bar, or press **Ctrl+H**.

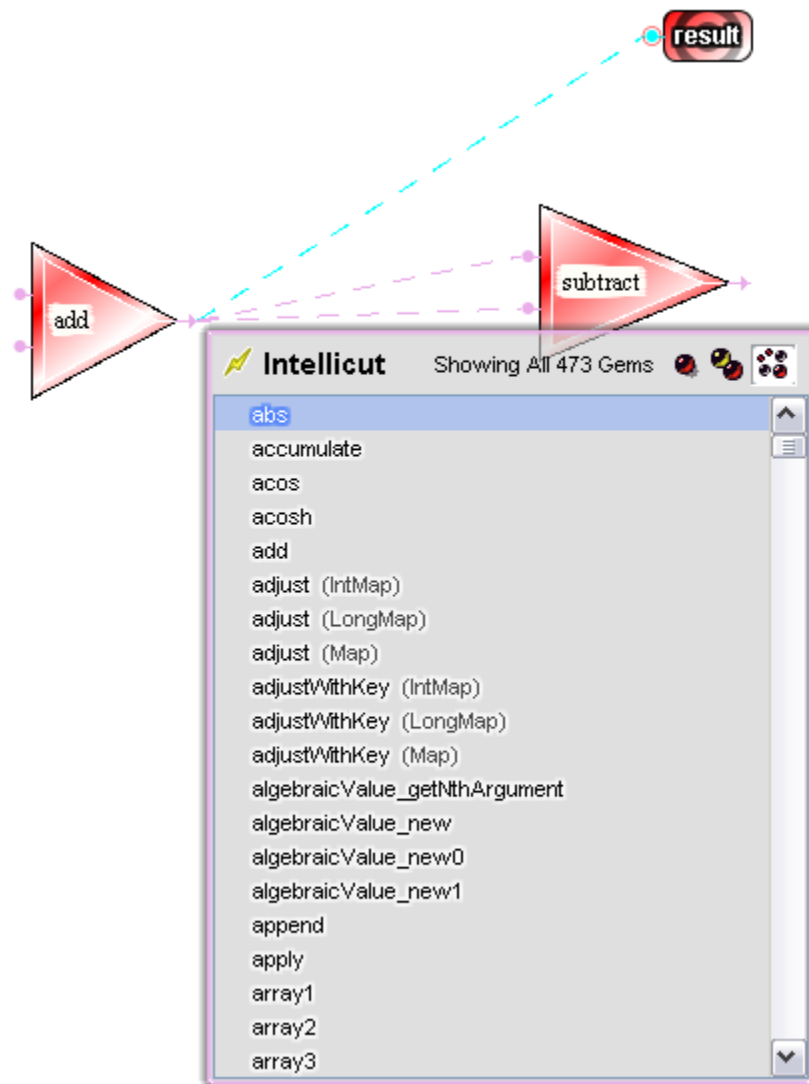
## 1. Getting Help Defining a New Gem: IntelliCut

The Gem Cutter has a feature called IntelliCut which will help you to build your Gem definitions. IntelliCut can be activated by letting the mouse pointer rest over a Gem connector (input or

output), by accessing the context menu associated with the connector, or by selecting the **Add Gem** (🔍) button on the toolbar and clicking on the connector. IntelliCut is designed to provide assistance in finding Gems that can legally be connected to the Gem connection the mouse is over. It does this in a number of ways:

- Suggesting connections which can be made to other Gems already on the Table Top.
- Showing you a list of Gems which can be brought out onto the Table Top and connected to the Gem from the modules currently open in the Gem Browser.

Figure 4.2 shows IntelliCut in action:

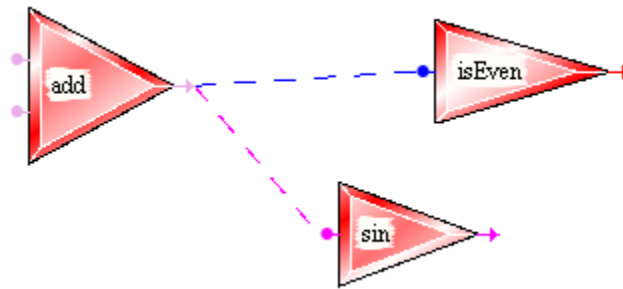


**Figure 4.2. IntelliCut in action**

The user has let the mouse pointer hover over the output of the add Gem. IntelliCut responds by showing that legal connections can be made to both the inputs of the subtract Gem, or to a list

of Gems not yet on the Table Top. At this point, the user can click on a Gem in the pop-up list to load a Gem onto the Table Top and automatically connect it to the output of `add`. Alternatively, the user could click on one of the indicated inputs to the `subtract` Gem or `result` Gem to form a connection to that Gem.




Naturally, IntelliCut understands the type system of Business Objects Gems, so it can correctly suggest valid connections which can be made between different but compatible types. This is shown in *Figure 4.3*:



**Figure 4.3. IntelliCut suggesting valid connections between Gems**

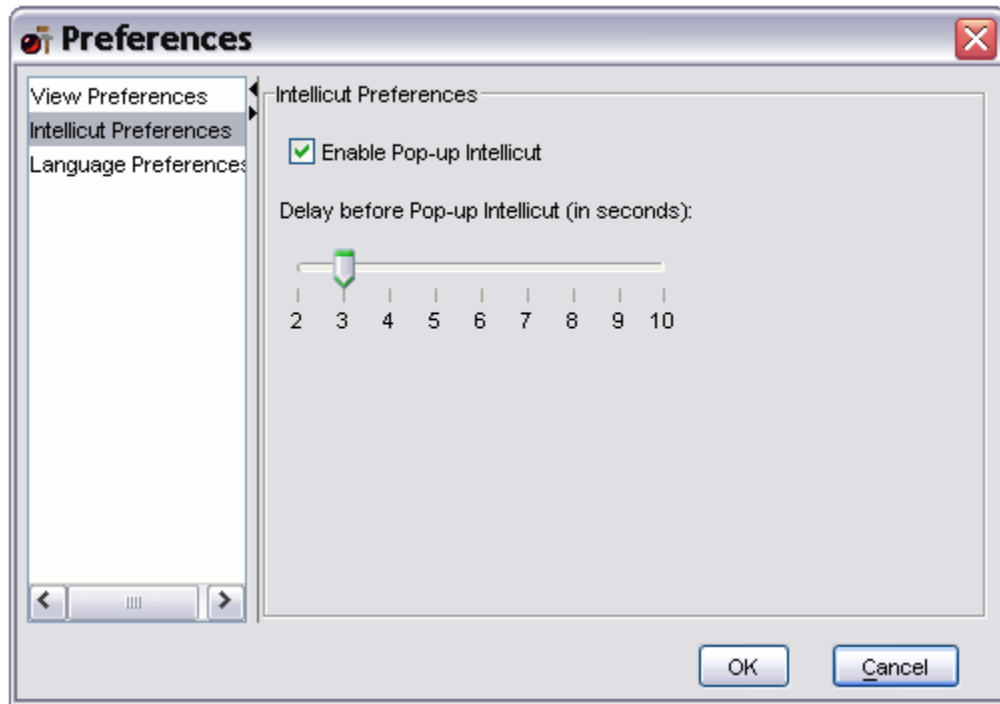
Here, IntelliCut is showing which Gems on the Table Top can be connected to the output of the `add` Gem. The `add` Gem's output is actually typed as a 'number' (type class `Num` - for information on type classes, see *Chapter 12, Understanding Type Expressions*) which includes both integers (type `Int`) and floating point numbers (type `Double`). IntelliCut determines that both the `isEven` Gem (taking `Ints`) and the `sin` Gem (taking a `Double`) can both be legally connected to the output of `add`. IntelliCut also uses the type colour of the 'destination' type to draw the 'potential connection' lines to indicate that these connections are compatible, but not the same, type.

In the top right corner of the IntelliCut window, there are three buttons which are used to control the Gems that are displayed. These buttons have the following functionality:

- |   |   |
|---|---|
|  <b>Show Best Gems</b>   | This button tells IntelliCut to display only the “best” Gems for the current connector. The Gems in this list will generally be very close in type to the current connector.  |
|  <b>Show Likely Gems</b> | This button tells IntelliCut to display Gems that are likely to be useful if connected to the current connector. This option will always display more Gems than the “Show Best Gems” option, but less than the “Show All Gems” option. Note that this option may not be present if there are only a small number of Gems that are connectible to the current connector. |
|  <b>Show All Gems</b>    | This button shows all Gems that are connectible to the current connector. No filtering of the list of Gems is performed.  |

The behaviour of IntelliCut can be modified using the IntelliCut Preference settings available through the **View** → **Preferences** menu option. The panel shown in *Figure 4.4* allows you to switch IntelliCut popup on or off, and to change the activation time period of IntelliCut:





**Figure 4.4. IntelliCut Preferences panel**

**Enable Pop-up IntelliCut** turns on or off the automatic operation of IntelliCut features. This will cause IntelliCut to operate if the mouse is allowed to hover over connectible Gem features (usually inputs and the output). If this is checked (on) then the setting of the slider bar represents the number of seconds before IntelliCut appears after the mouse pauses over a Gem connectible feature. If Pop-up IntelliCut is turned off, then IntelliCut can still be manually invoked by right-clicking on a connectible feature.

# Chapter 5. Testing a New Gem

Once a new Gem has been laid out on the Table Top, the new Gem is generally available for testing.

Figure 5.1 is a view of a sample Gem at this point, which adds two `Double`s then raises the result to the power of a third `Double`:

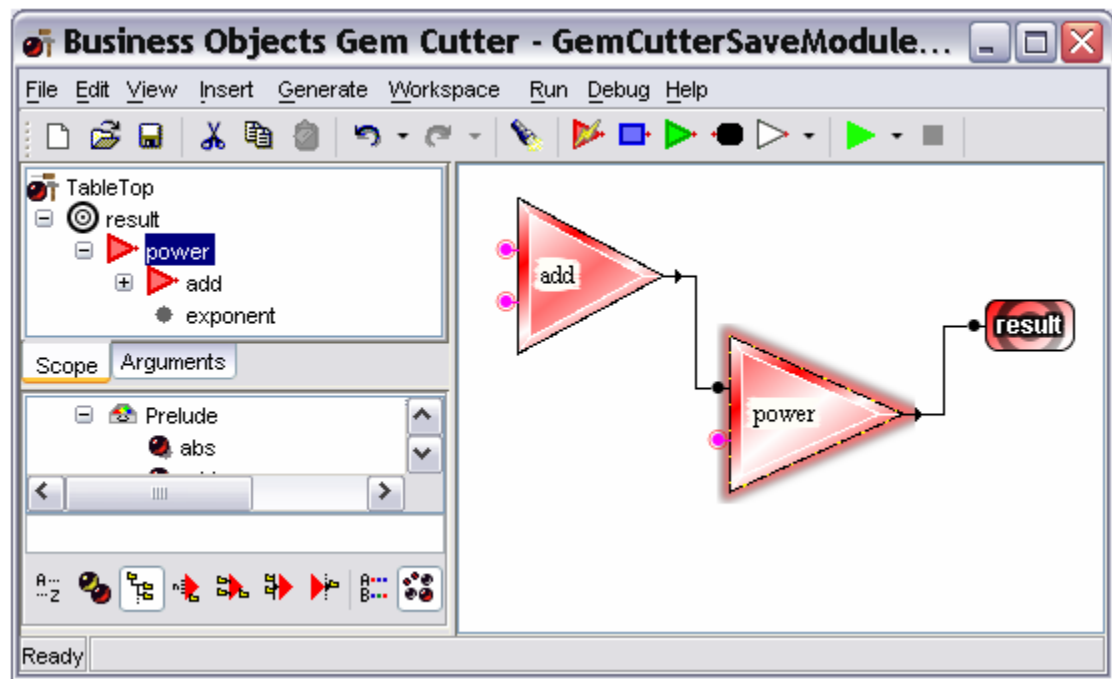


Figure 5.1. A sample Gem

The Gem developer is very likely to want to test the new Gem. In cases where the input types are relatively simple, this can be done by pressing on the 'play' button.

In cases where complex data values are expected by the Gem, it may be necessary to attach other Gems to the inputs of the Gem under testing in order to provide it with test data. The Gem Cutter will indicate if this is required by alerting the user with the dialog in Figure 5.2 when the **Play** (▶) button is pressed:

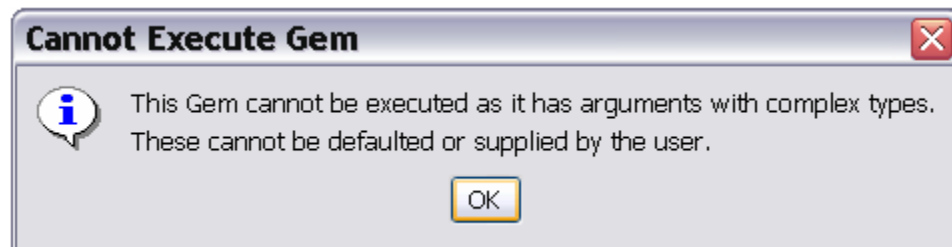
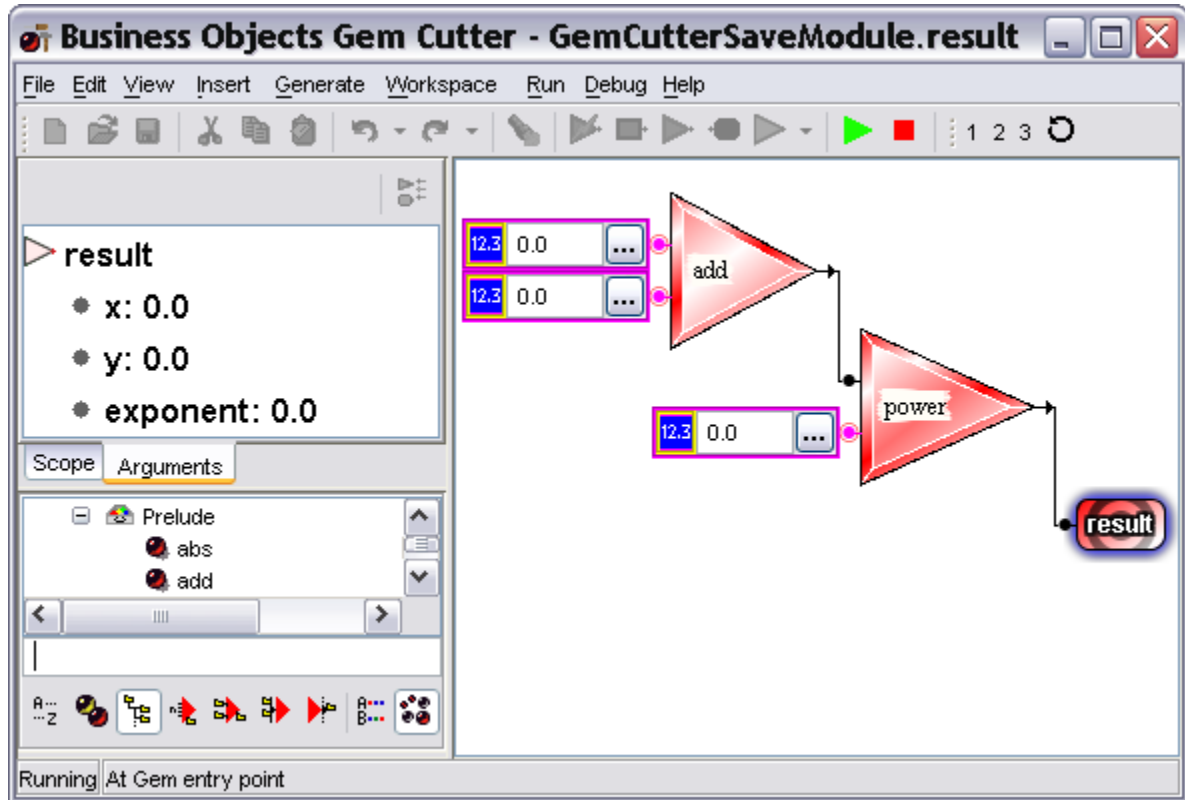


Figure 5.2. Cannot Execute Gem dialog

In the case of our simple `add-power` composition, we have very simple types, and the Gem Cutter is well able to deal with these. When the play button is pressed, the Gem Cutter realises that inputs are required in order to test the Gem and displays appropriate value entry panels to collect the required data. This is shown in *Figure 5.3*:



**Figure 5.3. Testing a Gem - Value Entry panels**

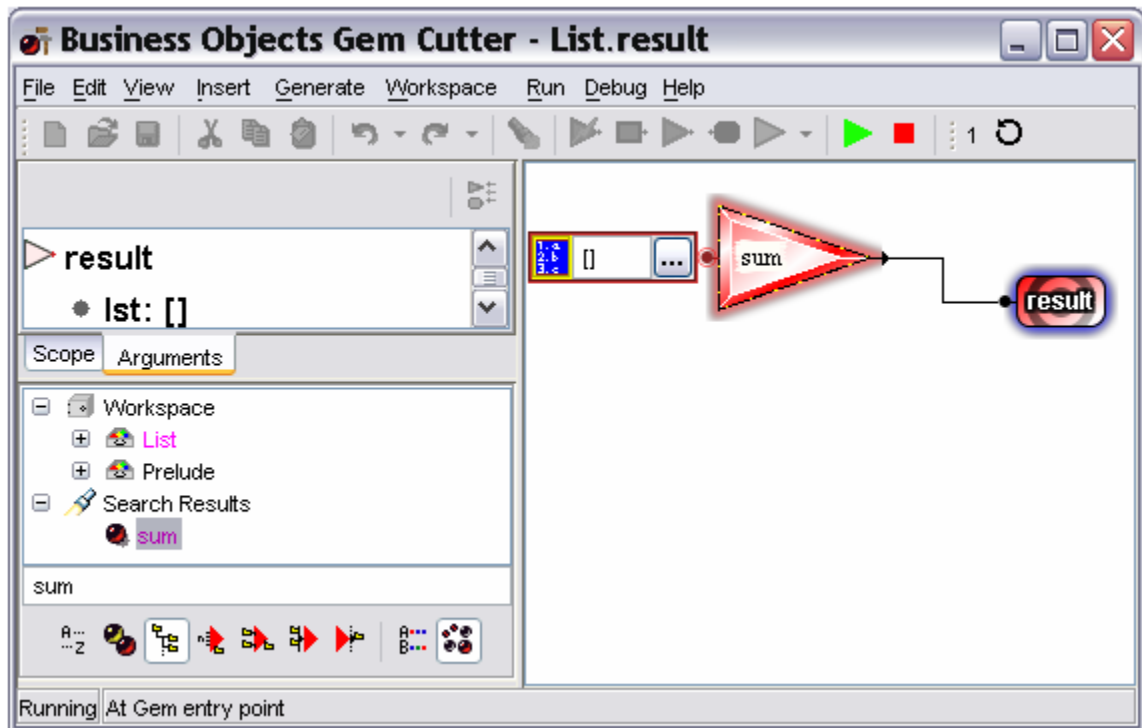
The status bar indicates that we are at the 'entry point' of the Gem. Notice that the target to be evaluated (in this case, `result`) gains a blue halo around it. Three value editors have appeared to collect values of the unbound inputs of the new Gem. In each case the required type of the input values is a double precision floating point number. To indicate this, the entry panels display an icon which hints at the appropriate type and each has a bright pink border to further associate them with the type `Double`. The arguments and their values are also displayed in the Arguments pane at the left of the screen.

The user can supply values in each panel for each respective argument. All input values are validated by the Gem Cutter to ensure they are legal for the required type. Once the right values have been entered, the user can run the Gem execution by using the play button. Values can be entered in the Arguments pane on the left as well as in the value editor panels themselves, if desired.

All of the built-in primitive types, as well as a number of more complex types, have value editors which can collect data for arguments. Lists, tuples and records are also supported, and value editors for the respective element types are presented accordingly. Alternatively, where an entry

panel has an ellipsis button to the right on the text entry field, the user may expand the entry panel another level. This generally makes available a more detailed form of entry panel specific to the required type.

For example, if we have a new Gem composed simply of the `sum` Gem from the standard library, and we attempt to run it, the first entry panel we will get will look like the one in *Figure 5.4*:



**Figure 5.4. Value Entry panel for `sum` Gem**

Notice that the icon on the entry panel indicates that the panel is expecting a list. We can expand the list entry panel a step by pressing its ellipsis button. This yields the view in *Figure 5.5*:

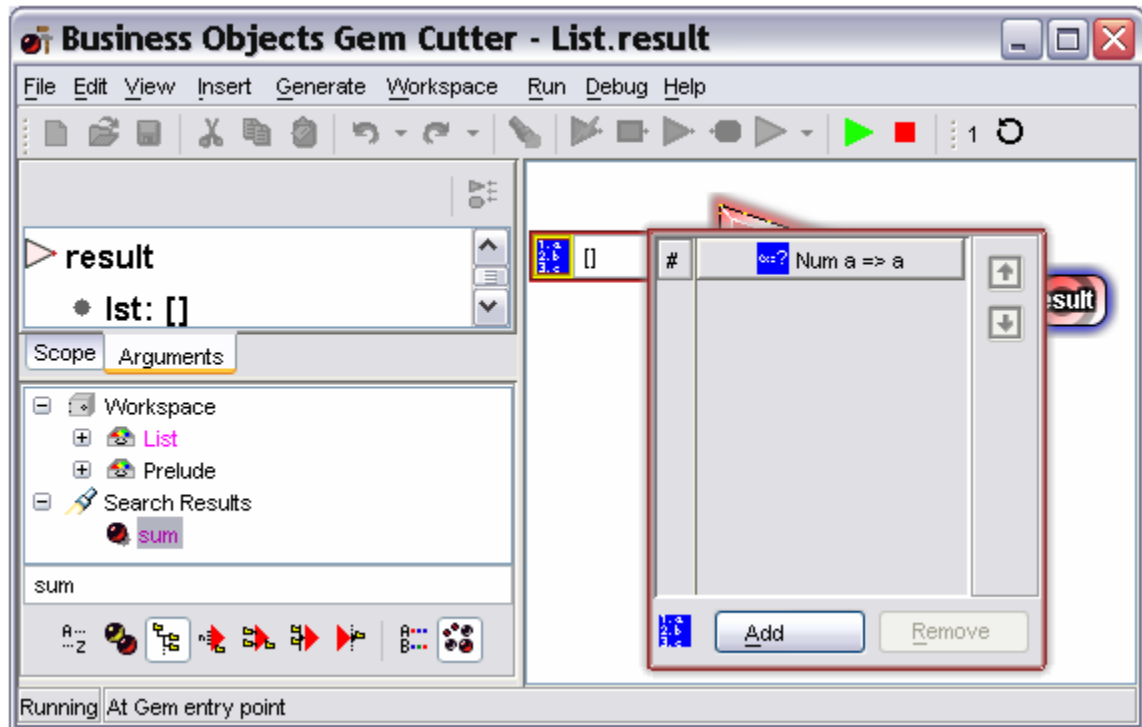


Figure 5.5. List Value Editor

A larger panel pops up (but with the same coloured border). This is the list entry panel. We can enter items into the list by pressing the **Add** button. The newly added entry's value can then be modified by clicking on the ellipsis button next to it.

In this case, the first time this is done, a window will appear allowing the type of the list to be selected. The `sum` Gem accepts a list of any type that is a member of the `Num` type class. Therefore, a type for the elements of the list must be chosen, such as `Int` or `Double`. This window is displayed in Figure 5.6:

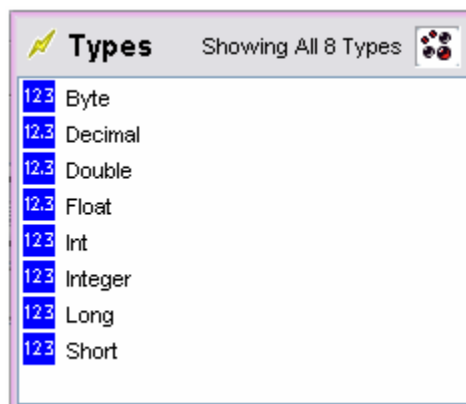
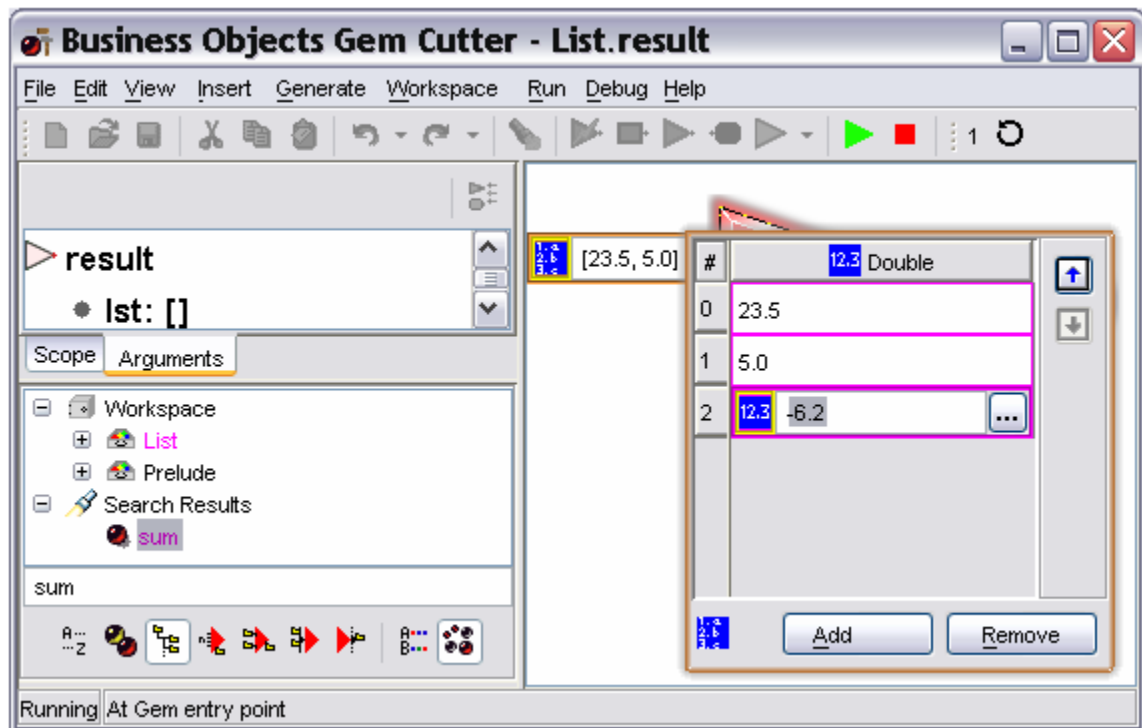


Figure 5.6. Type selection panel

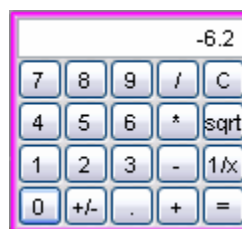
Selecting `Double` for the list type, we now add a few more items to the list. *Figure 5.7* shows the result:



**Figure 5.7. List Value Editor with `Double` values input**

We have added three list elements (23.5, 5, and -6.2). Each time the **Add** button was pressed, a new floating point number entry panel appeared in the list (which will scroll if there are too many to show concurrently). We can reorder the list elements using the up and down arrow buttons on the list panel.

As a convenience, for floating point numbers there is another level of panel that can be accessed for value entry. Pressing the ellipsis button on a floating point entry panel will pop up the panel in *Figure 5.8*:



**Figure 5.8. Calculator panel**

This is the Calculator Panel, and is provided in order to generate appropriate numeric values where these are constants derived from some arithmetic process. Notice how this panel indicates that it deals with 'bright pink' types (`Double`) by colourising its border accordingly.

Once argument values have been assigned the Gem is run by the user and a result is obtained. The result value may be of a simple type (such as a single numeric value), or it could be a data structure, such as a list. In each case, the result can be explored in the Results window which pops up. *Figure 5.9* shows the result of our `sum` Gem from before:



**Figure 5.9. Gem Results panel**

The result is shown in a familiar control – a Value Panel. If the result is of a compound type, then the result can be explored by clicking on the ellipsis button to the right of the result text. Doing so will 'expand' the result to show you more internal structure. For example, a list value will expand into a scrollable list control showing all the list members.

The Results window can be closed by clicking on the window close button in the top-right corner. However, you may leave the window open and continue to test your Gem for the purpose of comparing future results with those just obtained.

You may have noticed that when the green 'play' button is pressed for the first time, the buttons in the Toolbar change to look like those in *Figure 5.10*:



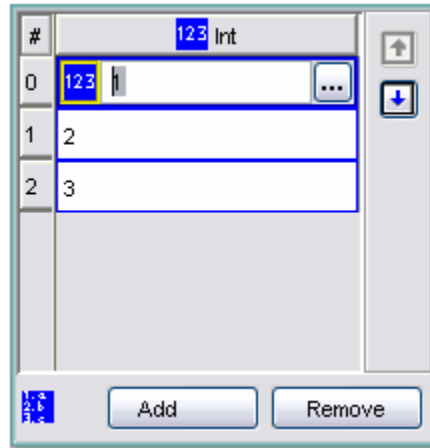
**Figure 5.10. Toolbar buttons when testing a Gem**

The red **Stop** (■) button is used to stop the running of the Gem and return to design mode.


The numbered buttons are used to select the Gem's arguments. In this case, the Gem constructed on the Table Top has only one input argument – the list that is an input to the `sum` Gem.

The **Reset Argument Values** (↺) button clears all the Gem's argument values, while remaining in test mode so that new argument values can be entered.

Occasionally, you may need to change the type of a value being used as a test input to a Gem. For instance, you may have tested a Gem with a list of `Ints` and you want to test it again with a list of `Doubles`. In this scenario, you would start with test input value entry panels looking like those shown in *Figure 5.11*:



**Figure 5.11. List Value Editor for Ints**

Notice that the type icon (in this case, the icon for `Int`: ) in the panel is highlighted with a yellow border. This indicates that the type is modifiable. Given that we want to change the type of the list element from an `Int` to a `Double`, we can click on the active list element's type icon to bring up a type list. The list that pops up shows us the options we have for morphing the original type into a new one. Also, you always have the option of deleting the type constraint completely (and effectively starting again from scratch).

Once the new element type has been assigned, you can enter values of the new type and test the Gem again.

## 1. Viewing the Gem Definition

Sometimes it may be desirable to see the generated CAL code for a Gem before it has been saved to a module. To view the CAL code the Gem Cutter would generate for the Gem design currently on the Table Top, select the **Debug** → **Dump Gem Definition** option from the menu bar. This will output the CAL definition of the Gem to the console.

Let's revisit the example shown at the beginning of *Chapter 5, Testing a New Gem*, in *Figure 5.3*. Using the **Dump Gem Definition** feature outputs the following text to the program console:

```
Gem Definition: private result x y exponent = Math.power (x + y) exponent;
```

Observe that the left-hand side of the Gem definition indicates that the result Gem takes three arguments: *x*, *y* and *exponent*. The functionality of the Gem displayed on the right states that the Gem will add the arguments *x* and *y* together, then raise this to the power of the *exponent* argument. This is identical to the Gem design currently set up on the Table Top.

An alternative way to get the CAL code generated for the Gem design currently on the Table Top is to right-click on the Table Top and select the **Copy Special** → **Target Gem Source** option.



## Chapter 6. Burning Inputs

So far we have considered fairly simple compositions of Gems where the simple output value is connected into the input of another Gem. However, not all Gem compositions are quite as straightforward. To illustrate this, we'll walk through a simple example.

Suppose we wish to construct a Gem which will tell us if any number in a list is larger than `pi`. In order to do this, we can call upon the services of the `any` Gem in the `List` module. This Gem takes a list of any type and returns a `Boolean` value of `True` if any value in the list causes a supplied predicate function to return `True`. A predicate function is a function which will produce a `Boolean` value based on an input. In this case, the predicate required will examine a single element in the original list and indicate `True` if this element is greater than `pi` or `False` otherwise.

Let's verify that `any` does indeed perform this function by examining its 'anatomy'. Dropping an `any` Gem on the Table Top allows us to inspect its inputs and output using the mouse to display tooltips (see *Figure 6.1*):

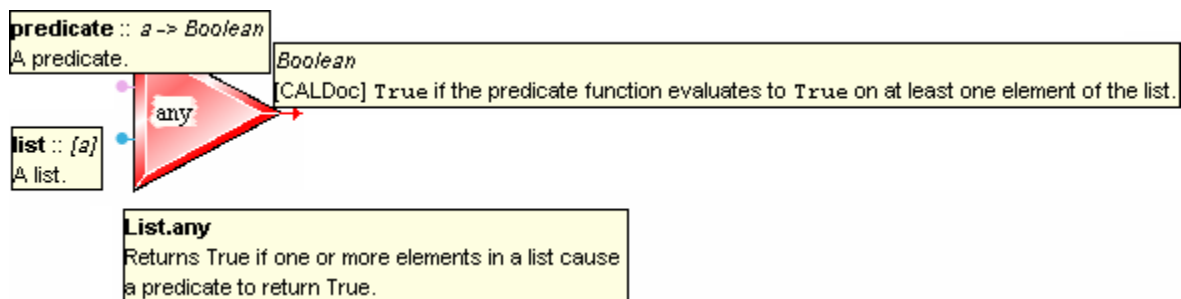


Figure 6.1. `any` Gem with tooltips displayed

The predicate function we want is very simple (just a test for being greater than `pi`). Again, we have a standard library Gem which will do the work for us, namely `greaterThan`. We can drop this onto the Table Top as well and perform a similar inspection (see *Figure 6.2*):

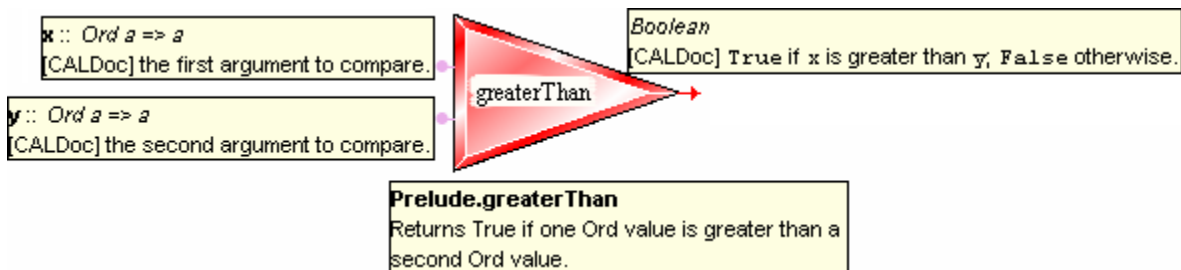
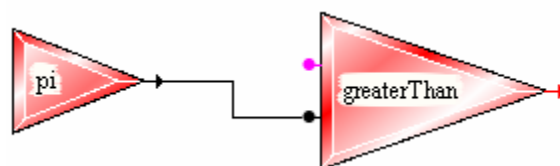


Figure 6.2. `greaterThan` Gem with tooltips displayed

This tells us that `greaterThan` takes two values belonging to the class `Orderable` and produces a `Boolean` value.

We can produce a function which will always test for a value being greater than `pi` by attaching a `pi` Gem to the second argument of `greaterThan`. The `pi` Gem takes no arguments and simply

returns the double value `pi`. After dropping a `pi` Gem on the Table Top and connecting it up to the `greaterThan` Gem, we should end up with something that looks like *Figure 6.3*:

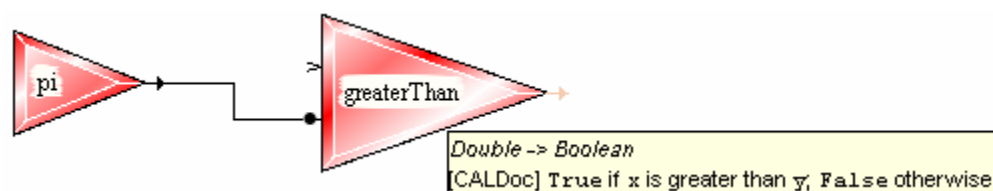


**Figure 6.3. `pi` Gem connected to `greaterThan` Gem**

We're almost there! We've built a function which will do the actual value comparison we want, and we have the Gem which performs that actual work of checking a list based on such a comparison (the `any` Gem). All that remains is to connect these two up.

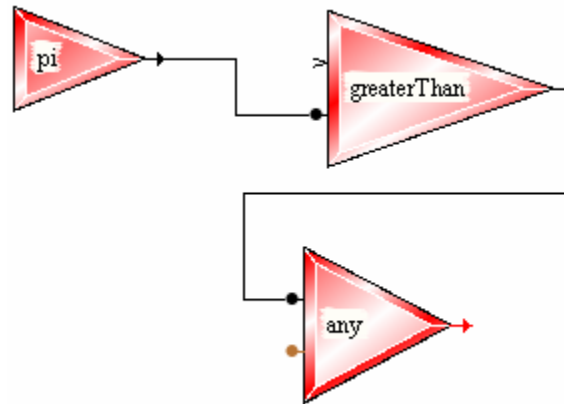
It's about at this point that we realise why things aren't always that simple! The output of our '`greaterThan pi`' composite function is a `Boolean` value, but the first input to the `any` Gem asserts that it takes a function that takes a value and returns a `Boolean`. This would imply that we can't take the output of this predicate composite and simply connect it into the `any` function as we might think we would want to. Actually, this makes perfect sense, because what we actually want is for the whole predicate function to be internally provided with values from the elements of the input list to `any`. So, actually, we don't want to externally provide the argument to compare to `pi` at all. How do we communicate this fact to the Gem Cutter?

The way to achieve what we want is by using 'input burning'. By burning the first argument to the `greaterThan` Gem, we are telling the Gem Cutter that we are not intending that this should have a single value provided to it, but that this value should be 'collected' later on when this function is applied to data (in this case within the `any` Gem). In actual fact, we want to create a functional result from the combination of the `greaterThan` Gem and the constant `pi`. This is achieved in the Gem Cutter by double-clicking an input to be burned or unburned. Doing this with the first argument of `greaterThan`, in our current example, will produce the result shown in *Figure 6.4*:



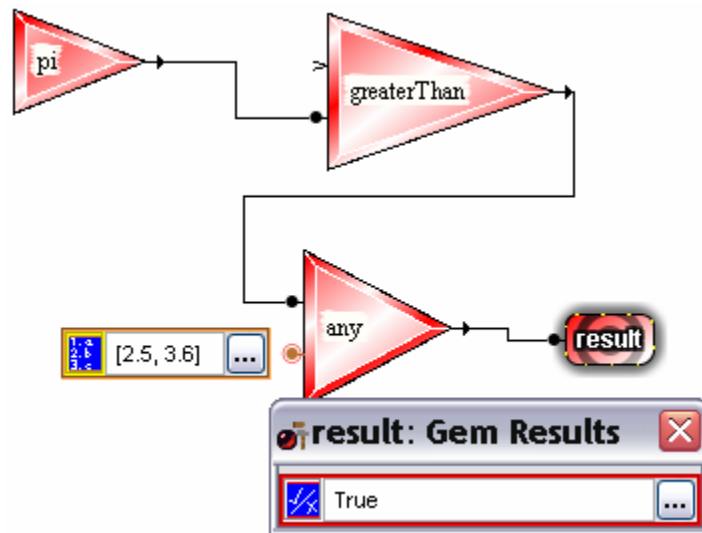
**Figure 6.4. Predicate function created by input burning**

Notice how the input is now rendered with a 'burnt twig' look to tell us that it has been burned. More importantly, note that the output type of the combination has changed to a functional result (a function which takes a `Double` and returns a `Boolean`). This is exactly what we need for our predicate function for `any`. Because this is now a type match for the first argument of the `any` function, the '`greaterThan pi`' combination can now straightforwardly be connected to the first argument of `any`, giving us the Gem graph shown in *Figure 6.5*:



**Figure 6.5. Predicate function connected to the any Gem**

We can examine the remaining unconnected inputs and output to see that we have what we set out to produce. We can test the new Gem to verify that we have the correct behaviour. An example test is shown in *Figure 6.6*:

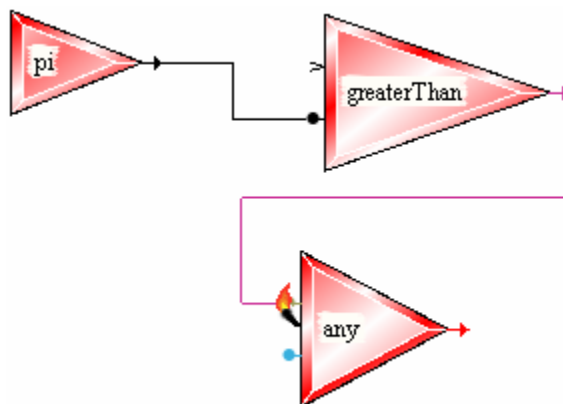


**Figure 6.6. Testing a Gem with input burning**

While we only had to burn a single input in this example, other scenarios may call for the burning of more than one input (including all the inputs of a Gem).

Now that we've introduced the concept of burning, it's safe to mention that in many cases, the mechanics of manually burning the inputs are unnecessary! Because complete and correct type information is always known for Gem components, there are many occasions where burning is required, and the correct input burn combination can be unambiguously inferred by the Gem Cutter itself. In these situations, the Gem Cutter will allow the user to establish a connection between apparently incompatible Gem input-output combinations providing there are one or more inputs that can automatically be burnt to make the connection legal. In the example we

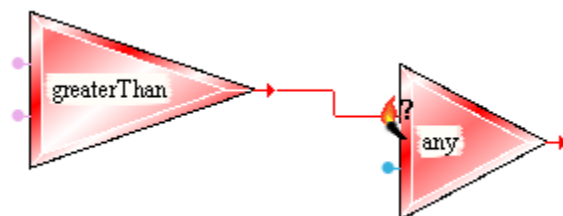
have just examined, this would manifest as the view in *Figure 6.7* when attempting to connect the unburned `greaterThan` Gem to the `any` Gem:



**Figure 6.7. Automatic input burning**

Before we release the mouse button while dragging out the connection between the two Gems, and while hovering over the input to `any`, the Gem Cutter will indicate that a connection is possible if burning is performed on the `greaterThan` Gem. This is communicated by the cursor changing, and by the Gem Cutter provisionally burning the correct input of the `greaterThan` Gem. Completing the connection by releasing the mouse button in this configuration will establish the burnt argument(s) and the required connection.

In cases where the Gem Cutter is not able to unambiguously establish a connection in this way through burning, it will indicate that a connection is still possible if the user can determine the burn configuration they need prior to connecting. For example, observe *Figure 6.8*:

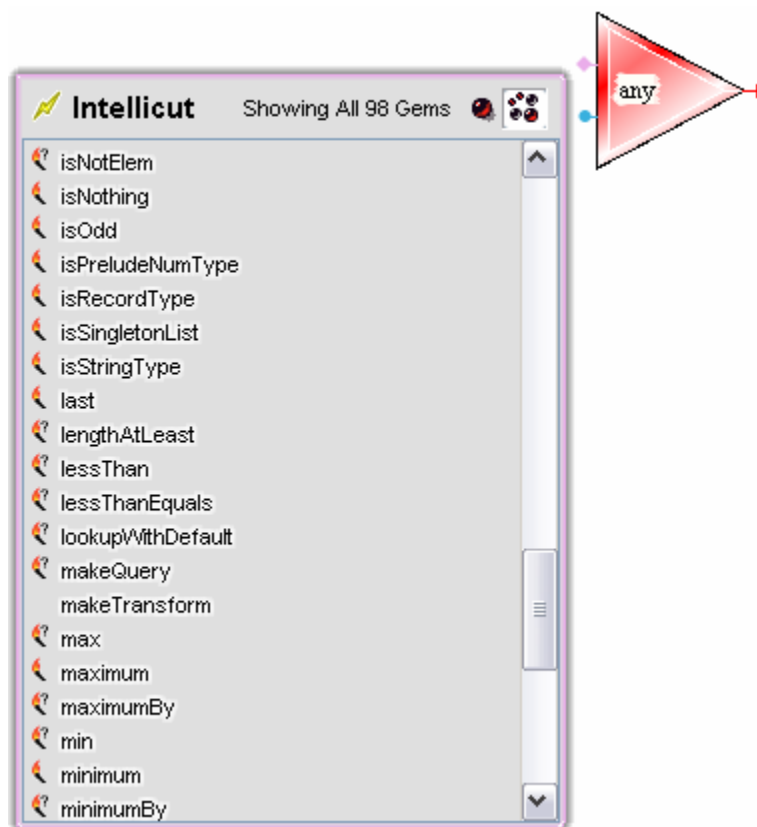


**Figure 6.8. The Gem Cutter cannot determine which input to burn**

In this case, we have tried to connect the output of the `greaterThan` Gem to the input of the `any` Gem before connecting anything to the inputs of the `greaterThan` Gem. The Gem Cutter has no way of knowing if we want its first or second input to be burnt, so the cursor changes to a burn icon with a question mark next to it. To connect the Gems in this manner, one of the inputs to the `greaterThan` Gem must be manually burnt.

The IntelliCut feature of the Gem Cutter is also smart enough to handle cases where connections can be established between Gems when burning is required. Hovering over a Gem input will cause the IntelliCut list to popup after the appropriate pause, and this will include Gems which

can be connected to the input if burning is performed on one or more of the new Gem's inputs. *Figure 6.9* shows an example:



**Figure 6.9. IntelliCut showing Gems that can be connected with input burning**

Notice that the Gem names appearing in the IntelliCut list can have a burn icon before them. Names appearing with no icon represent Gems which can simply be connected with no need for burning. Gems with a burn icon indicate Gems which can be connected to the input, but for which some burning is unambiguously required. If the user clicks on one of these, the chosen Gem will be added to the Table Top and connected to the appropriate input with the correct burning of the new Gem's inputs to make the connection meaningful (and legal!). Gem names that are followed by a burn icon containing a question mark are Gems which can be connected to the input, but for which there are multiple possible combinations of input burning which would result in a legal connection. Clicking on one of these will add the Gem to the Table Top, but will not connect it, as user intervention is required to burn the correct inputs on this Gem to obtain the user's desired functionality.

---

# Chapter 7. Special Gems

Special Gems are not actually 'special' at all, at least not at the point that a new Gem is saved. However, whilst the Gem is being created, Special Gems indicate a special type of contribution to the overall Gem definition. That is to say that they define something about the Gem that is not available from the libraries of Gems available through the Gem Browser.

Special Gems are added to the Table Top by clicking on a button on the toolbar, rather than dragging from the Gem Browser. The toolbar section shown in *Figure 7.1* includes buttons for each of the Special Gems:



**Figure 7.1. Special Gems Toolbar buttons**

Clicking on a Special Gem button will cause the cursor to change to an arrow with a picture of the selected Gem type beside it. Clicking on the Table Top will then place the required Special Gem at that location.

It is also possible to add certain Special Gems from the context menu that appears when right-clicking on a Gem connector. The Special Gems that are available to add from the context menu depend on the particular connector.

We will take each Special Gem in turn, describe its functionality and provide an example.

## 1. The Value Gem

The blue Value Gem simply defines a constant. It has a specific type and value which can be connected to a compatible input of a Gem to bind this value to the Gem. Value Gems appear as rectangular, blue Gems when they are added to the Table Top.

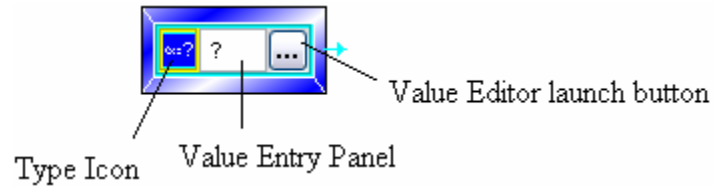
Value Gems are often required to 'de-generalise' a Gem, fixing it to deal with a reduced (or specific) set of cases. For example, a general purpose filter could be fixed to only look for values above 1000, rather than values above any input.

Attaching Value Gems to inputs will remove that input from the set of inputs which will be exposed by the Gem under construction when it is saved.

Value Gems use the value entry panels to collect the constant values of the particular type that is asserted by the Gem. For example, if the Gem is set up to be a constant list of double precision floating point numbers, then the list entry panel is used for elements of type `Double`.

We will take a look at some examples of using the Value Gem.

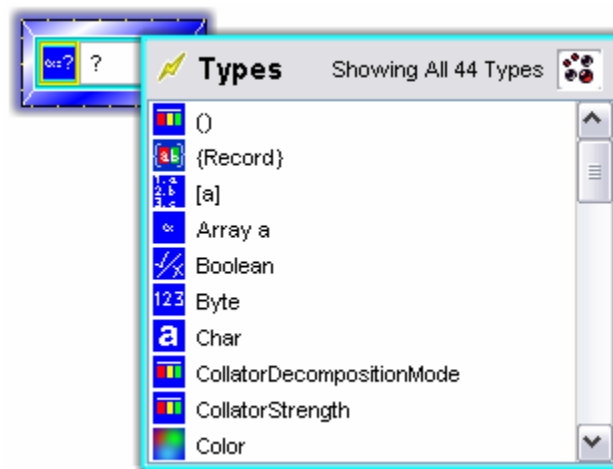
*Figure 7.2* shows a Value Gem as it appears when placed on the Table Top, with its sections labelled:



**Figure 7.2. Value Gem with sections labelled**

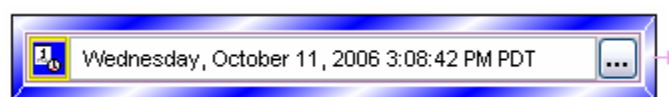
The Type Icon displays the type of the value represented by the Value Gem. The Value Entry panel displays the current value represented by the Value Gem, and allows the user to enter a new value. The Value Editor launch button launches a custom editor designed to edit the type of value the Value Gem is currently representing. Value Gems allow the user to set a value as part of the Gem definition, rather than leaving the Gem to collect the value when it is executed. Therefore, the value-collecting mechanism is reused in Value Gems.

Initially a Value Gem does not know what type of value it is supposed to be representing. This is indicated by the type icon and value field showing a question mark. To begin with, we need to indicate what type the constant will be. To do this, we click on the ellipsis button to the right of the value entry panel, which displays the image in *Figure 7.3*:



**Figure 7.3. Value Gem type selection panel**

We are now presented with a list of possible types for the value. This list is composed from all the types that are currently visible from the modules that are open in the Gem Cutter. Let's create a `Time` value. To do this, we must double click on the `Time` item in the types list. The look of the Value Gem now changes to that in *Figure 7.4*:



**Figure 7.4. Value Gem representing the `time` type**

The Value Gem now knows what type of value it is supposed to be representing, and changes to represent an editor for values of this type. Notice the following differences:

- The type icon has changed to tell us that we are now dealing with a `Time` type.
- The value field is now showing a default value for this type (in this case, the current time and date).
- The colour of the Value Gem's output connector has changed. It will become the colour which will be used in this session of the Gem Cutter to represent all values of type `Time`.

At this point, we can edit the actual value that the Value Gem represents. The way to edit the value depends on the type, but in general, you can alter the value by typing into the value field, clicking on the value field and using the cursor keys (up move values 'up' or 'down'), or by invoking a custom editor specific to the type by clicking on the 'ellipsis' button on the right. In any case, the technique for altering the value is specific to the particular value entry panel for the selected type. It is therefore consistent with the situation where you are entering values of this type during Gem testing.

Once you have set the desired value in the Value Gem, the output of the Value Gem can be connected to another Gem in the usual way. This will apply the value to the input of the other Gem.

Business Objects Gems have an advanced type system that allows for complex types to be described. In keeping with the flexibility of this type system, Value Gems let you create compound, highly nested values. If you select a type which is a 'container' for other values (possibly of different types), the Value Gem value editors will let you continue to select nested types and therefore build up the 'shape' of the value you need.

Here's an example of building a constant value which is a record containing an integer, a date and a colour value.

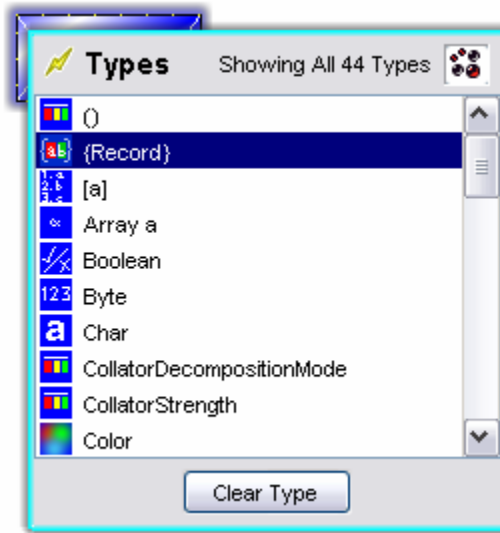
As usual, we start with a completely unspecialised constant, as shown in *Figure 7.5*:



**Figure 7.5. Unspecialized Value Gem**

The first thing we need to say about our value is that it is a record of three values. The name of the type which describes this is `Record`. We'll select `{Record}` from the list of available types, as displayed in *Figure 7.6*:





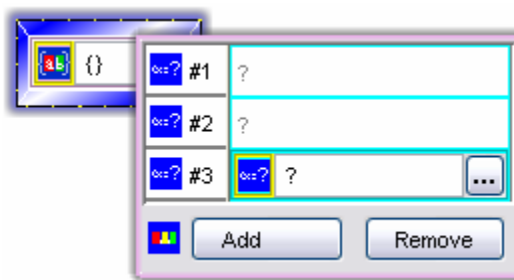
**Figure 7.6. Selecting the record type for a Value Gem**

Doing this changes the Value Gem, as before, to reflect the fact that we are now dealing with a Record value. This is displayed in *Figure 7.7*:



**Figure 7.7. Value Gem representing a record type**

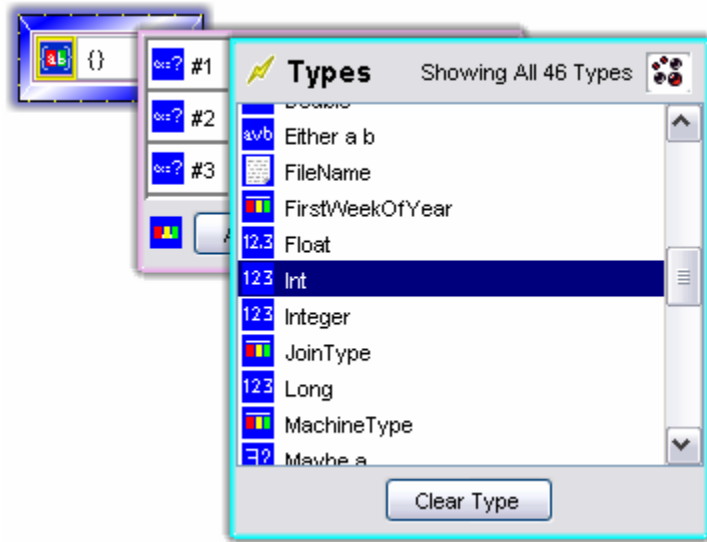
The icon and value field both tell us we are dealing with a record type. Now, we need to specify that the `Record` should contain three items. To do this, click on the ellipsis button, and click the **Add** button three times. The result should look like *Figure 7.8*:



**Figure 7.8. Record editor**

The three question marks indicate that three values of unknown type are contained in this record. Now we need to tell the Value Gem what the types and values of each of the elements are.

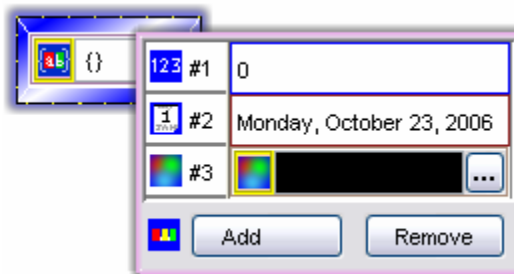
We want the first value to be an integer, so we click on the editor box for the first value and click the blue button outlined in yellow. Next, we select `Int` from the type menu shown in *Figure 7.9* which appears.



**Figure 7.9. Selecting a type for a record component**

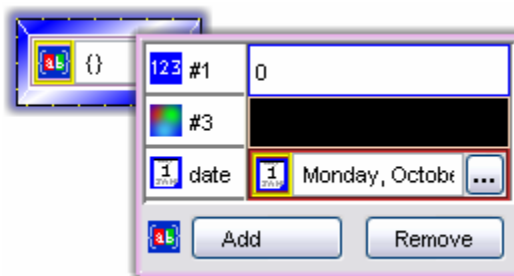
This changes the type of the first value in the record, and sets it to the default value (zero).

We can repeat this process to set the types (and default values) for the second and third values contained in the `Record` type. We will end up with the view in *Figure 7.10*:



**Figure 7.10. Record containing an `Int`, `Date` and `Color` value**

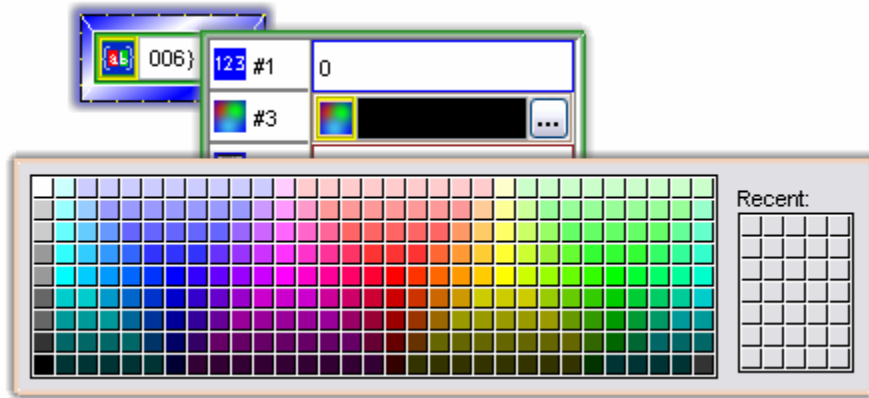
The names of the fields can also be changed. For example, we can change the name of the second field from “#2” to “date” by clicking on the box on the left with “#2” in it, and typing “date”. The result looks like *Figure 7.11*:



**Figure 7.11. Record after renaming the `Date` type field**

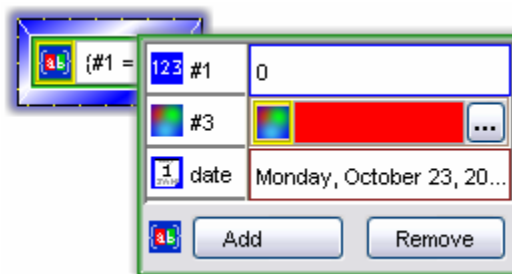
Note that the entries in the record are automatically organised by the alphanumeric ordering of their names. This is just for convenience for the user, as records are essentially a map-like entity – order is not significant. Valid names include strings starting with a lower case letter and containing only alphanumeric characters and the underscore character, and the pound sign (#) followed by an integer.

The Value Gem can always be revisited in order to edit the value of each part of the record. For instance, if we want to change the colour from black to red, we first click on the Value Gem, and select the ellipsis button next to the colour editor. This displays *Figure 7.12*:



**Figure 7.12. Changing the color field**

Selecting an appropriate shade of red changes the Value Gem to display the new settings, as shown in *Figure 7.13*:



**Figure 7.13. Record with color changed to red**

Naturally, it is possible to set types for component values which are themselves compound, creating an even deeper nesting for the resultant value.

If you make a mistake, or simply wish to change the design on the Gem being constructed to include a constant of a different type, you can change the type of a Value Gem. In order to change the type, the Gem Cutter must be in 'design mode' (i.e. you cannot change the design of a Gem if you are testing it). To change the type of a Value Gem, simply click on the type icon on the left of the Gem and select a new type from the menu that appears.

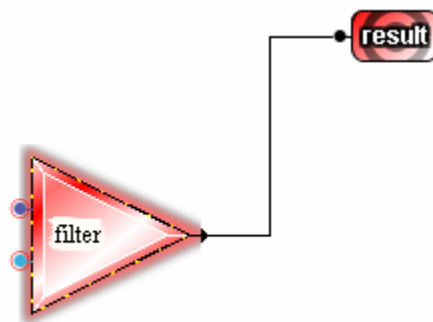
## 2. The Code Gem

Code Gems are used to inject a specific expression into the Gem under construction. In fact, Code Gems are actually code editors for the *CAL* language which underlies the Business Objects Gems concept. A Code Gem can therefore be made to represent any user defined function, including complex functions involving local variables and calls to external functions in the visible set of modules.

Code Gems are functions and therefore appear as triangular Gems on the Table Top, but are green instead of the regular red or yellow.

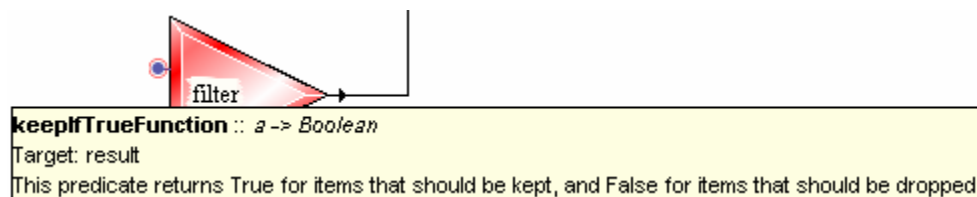
We'll now walk through an example of Code Gems to demonstrate how they might be used by creating a customised filter Gem. This Gem will find all entries in a list which have a value greater than 1000.

Let's start by taking the `filter` Gem from the `List` module. We know that the output from this will be the output of the Gem we're designing, so we can go ahead and connect it to the Result Target. *Figure 7.14* shows our Gem so far:



**Figure 7.14. `filter` Gem connected to result target**

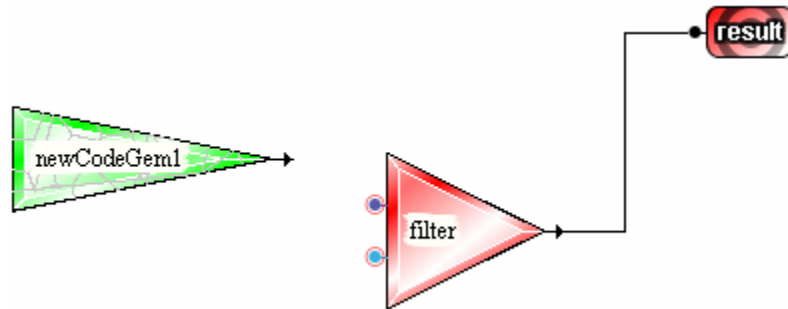
Taking a look at the `filter` Gem tells us that it takes a predicate function as its first argument. This is a function that will return a `Boolean` type, which will indicate whether to accept or reject a list element into the set that `filter` will return. Hovering the mouse over the dark blue input (Figure 7.15) shows us its type:



**Figure 7.15. `filter` Gem with tooltip**

We can see that the type required is a function which takes anything (the 'a') and returns a `Boolean`. This definition sets the 'target' type that we have to achieve in our Code Gem in order to make it a compatible predicate function for `filter`.

We'll now create a Code Gem that we'll edit to be a predicate function to achieve the desired effect. Pressing the **Code Gem** (▶) button on the toolbar and clicking in an empty area on the Table Top adds a new Code Gem, as shown in *Figure 7.16*.

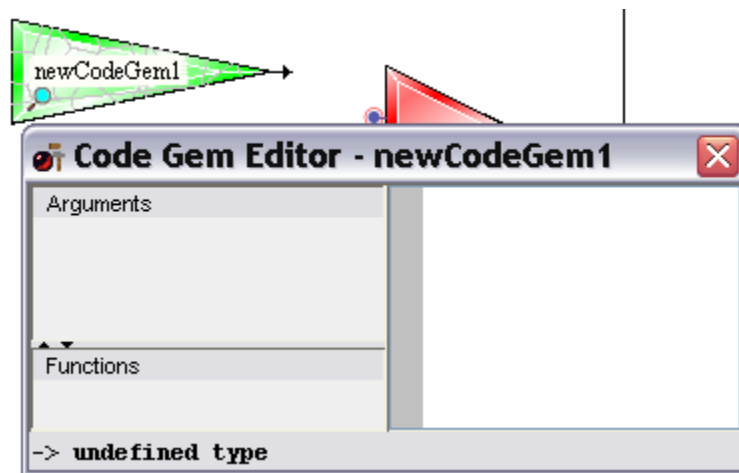


**Figure 7.16.** New Code Gem added to Table Top

The new Code Gem is automatically named for us, although this is mostly for the user's benefit. Technically speaking, we will be creating a lambda expression, which means an anonymous function.

Notice that the Code Gem appears broken at the moment. The cracks on its surface are an indication that the Gem is not 'good', that it requires some attention to be useable in the Gem we are constructing. At the moment, this is simply because we haven't actually given it a definition!

By default, double-clicking a Code Gem will open its Code Gem Editor. *Figure 7.17* shows the editor for the new Gem:



**Figure 7.17.** Code Gem Editor

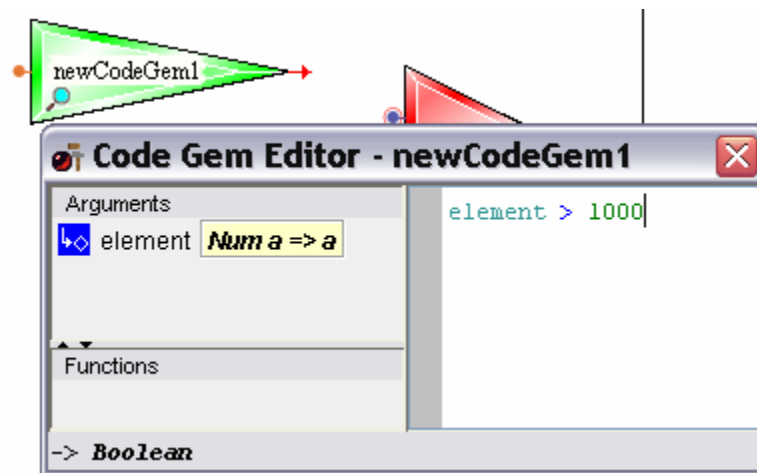
The Code Gem Editor associates itself with the Code Gem by name and by proximity (initially). Also, note that an image of a magnifying glass has appeared on the Code Gem. This indicates that this Code Gem's associated editor window is open. The blue colour of the lens means that

this Code Gem's editor window has the focus currently. If the lens appears white, it means that this Gem's editor window is open, but does not currently have the focus. This is useful to help show which Code Gem is currently being edited when multiple editor windows are open at once.

The function we want to describe is a test for a value (which will be our list element) being greater than 1000. No big surprises here, the code we want is the expression:

```
element > 1000
```

The Code Gem Editor should look like *Figure 7.18* after typing in this piece of code:



**Figure 7.18. Code Gem Editor containing CAL code**

As soon as we paused typing, the Code Gem Editor took the code and checked it. It was checked for syntactical correctness according to the CAL language specification, and it was checked for 'type correctness'. All this takes a tiny fraction of a second, so it is done every time the user pauses typing. This means that you can stop if you are unsure and see what the Code Gem Editor has to say.

Notice that various fields have appeared or changed on the Code Gem Editor to indicate the meaning of the expression we just typed.

Firstly, an item has appeared in a panel to the left of the typing area. This is telling us that the name `element` has been inferred to be an argument (the icon on the left tells us this). It also tells us that the type of this argument is inferred as a `Num` (a number).

At the bottom of the editor, we see the text `"-> Boolean"`. This tells us that the output type of the function has been inferred to be a `Boolean` type. Along with the argument, we can see that we have produced a function that takes a `Num` and produces a `Boolean`. The type expression of this would be `(Num a => a -> Boolean)`. This is perfectly compatible with the required type `(a -> Boolean)`, so we have succeeded in creating our required predicate function.

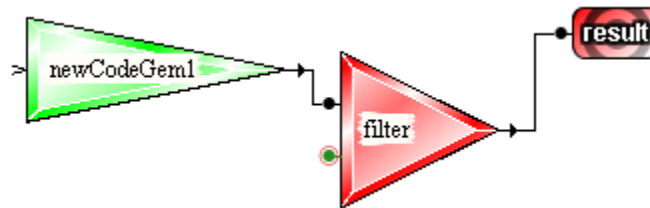
What has happened so far illustrates one of the major features of Business Objects Gems and the CAL language, which is that the system will attempt to be as general as possible. In the

absence of any information to the contrary, the CAL compiler has inferred that the most general type that the code will deal with is a number. If we were to force a more specific interpretation by changing the 1000 to 1000.0 (making the number into a definite floating point `Double`), we would see the type of element change to `Double`. Go ahead and experiment with this, then return to the original source code.

Syntax highlighting of the expression text itself gives further visual feedback about how the expression has been interpreted.

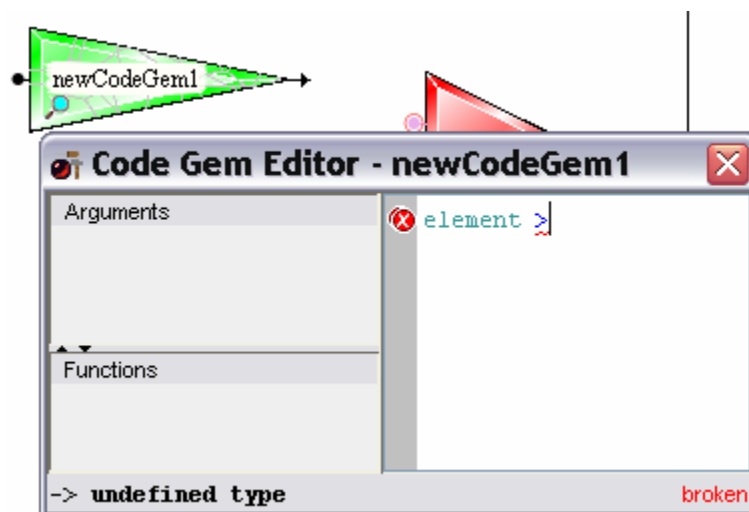
Finally, we can tell the expression made sense because the Gem (behind the editor) has now been 'repaired' and is showing no signs of cracking!

Now, to complete our filter, simply connect the output of the new Code Gem to the first input of the `filter` Gem. This will result in the input to the Code Gem being automatically burnt as per the requirements of the `filter` Gem. The filter is shown in *Figure 7.19*:



**Figure 7.19. Gem to filter elements greater than 1000 from a list**

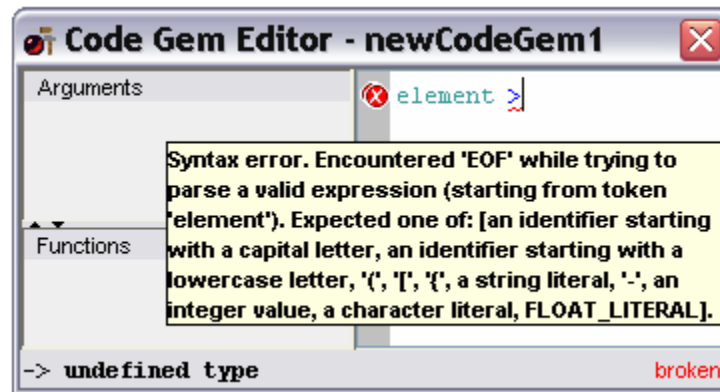
Now, suppose that when we were coding the functionality of the new Code Gem, we made a mistake. *Figure 7.20* shows an example where we got the expression wrong:



**Figure 7.20. Code Gem Editor containing incorrect code**

This time, the CAL compiler cannot infer that `element` is an argument, or give us a resultant type. Instead, the word “broken” appears in red text in the bottom right corner of the Code Gem Editor

window, and the errors are underlined in red. The Gem has cracked again, showing that we have some work to do to 'fix it'. To view a more detailed definition of a specific error, place the mouse pointer over the underlined code, as in *Figure 7.21*:



**Figure 7.21.** Code Gem Editor with tooltip describing code error

In this case, we haven't finished the expression correctly.

Another effect that was achieved once we had valid source in the Code Gem Editor was the fact that the Code Gem updated itself to show inputs and outputs of the correct type (at the same time that it became uncracked). The fixed Code Gem is displayed in *Figure 7.22*, along with its tooltips:



**Figure 7.22.** Code Gem with tooltips

Hovering the mouse of these new inputs and outputs will show us that the Gem believes that it takes a `Num` and returns a `Boolean`, just as expected. We can now go ahead and connect this Code Gem to the predicate input of the `filter` Gem and proceed to test or save our creation.

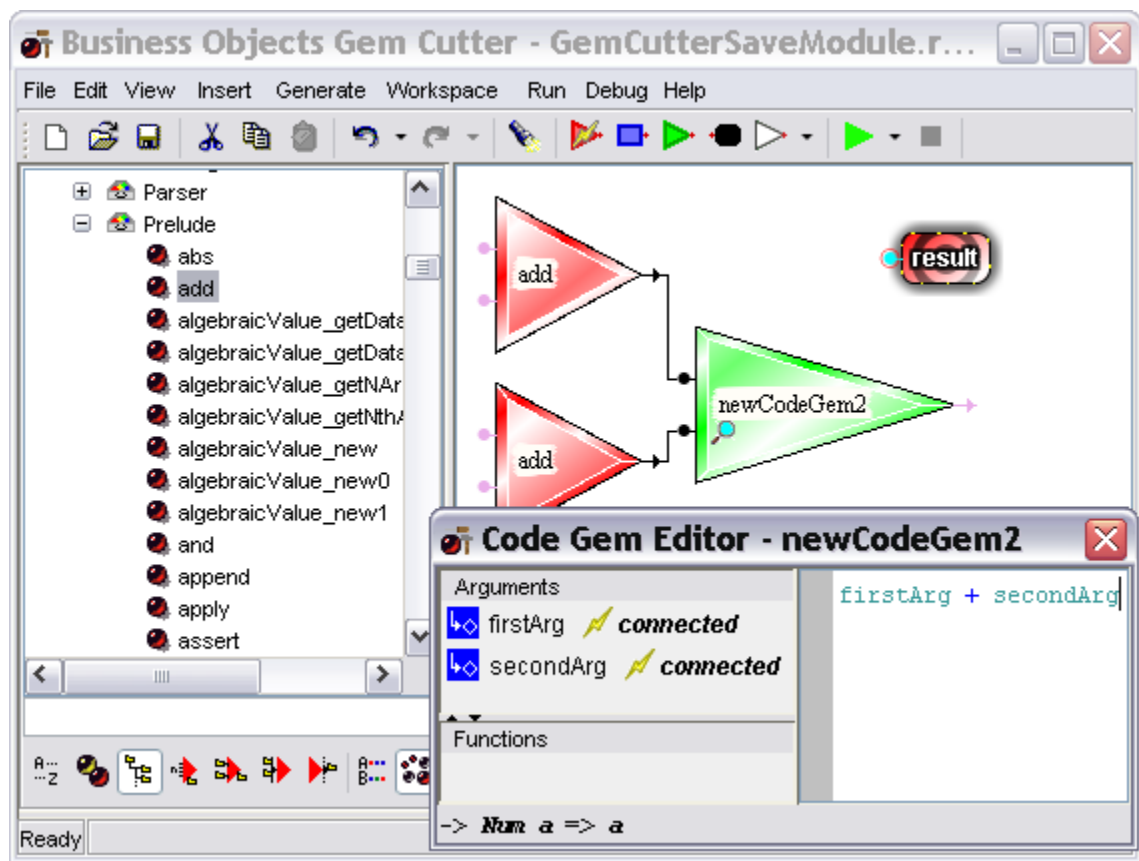
You can continue to make changes to your Code Gem after having connected it to other Gems (on its inputs or output). However, doing so will establish type constraints for the Code Gem, so it is often better to develop the code without establishing connections until the code looks correct and the inferred types are what is expected.

If you do connect the Code Gem and then alter the code, you will see a number of possible effects.

Here is an example of constraints on code after composition with other Gems. We start with a simple Code Gem to add two numbers together. We have connected two `add` Gems to the inputs of the new Code Gem. This changes the type description of the arguments to show that they are

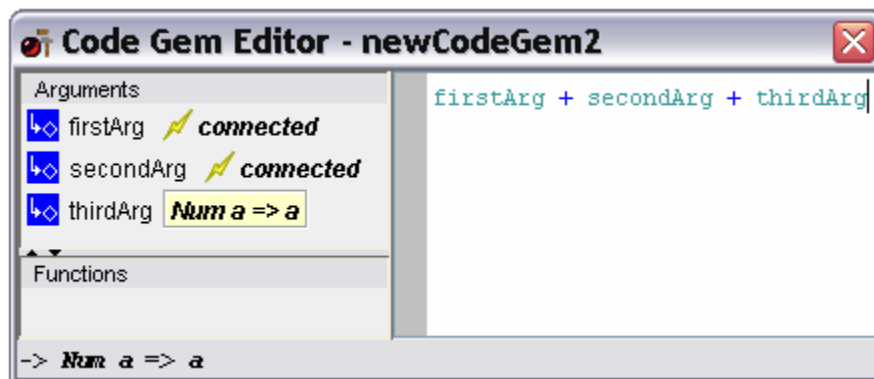


connected. The Code Gem Editor has inferred that the output is a Num. The situation is displayed in *Figure 7.23*:



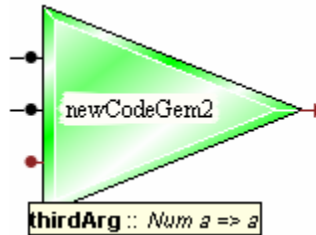
**Figure 7.23.** Two add Gems connected to a Code Gem

At this point, we will attempt to edit our CAL code to add a third parameter (added to the first two). The result is shown in *Figure 7.24*:



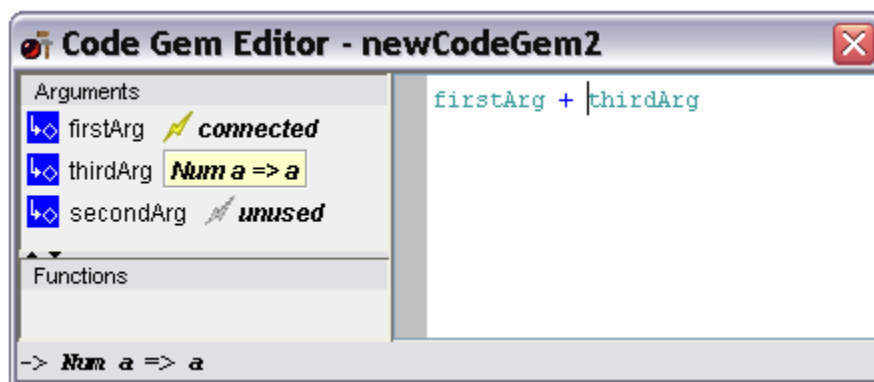
**Figure 7.24.** Code Gem Editor with three parameters

The new `thirdArg` argument is recognised and inferred as a `Num` because of its relationship to the other arguments (through the `+` operator). The display of the Gem on the Table Top also changes to include the new argument, as shown in *Figure 7.25*:



**Figure 7.25. Code Gem with added third argument**

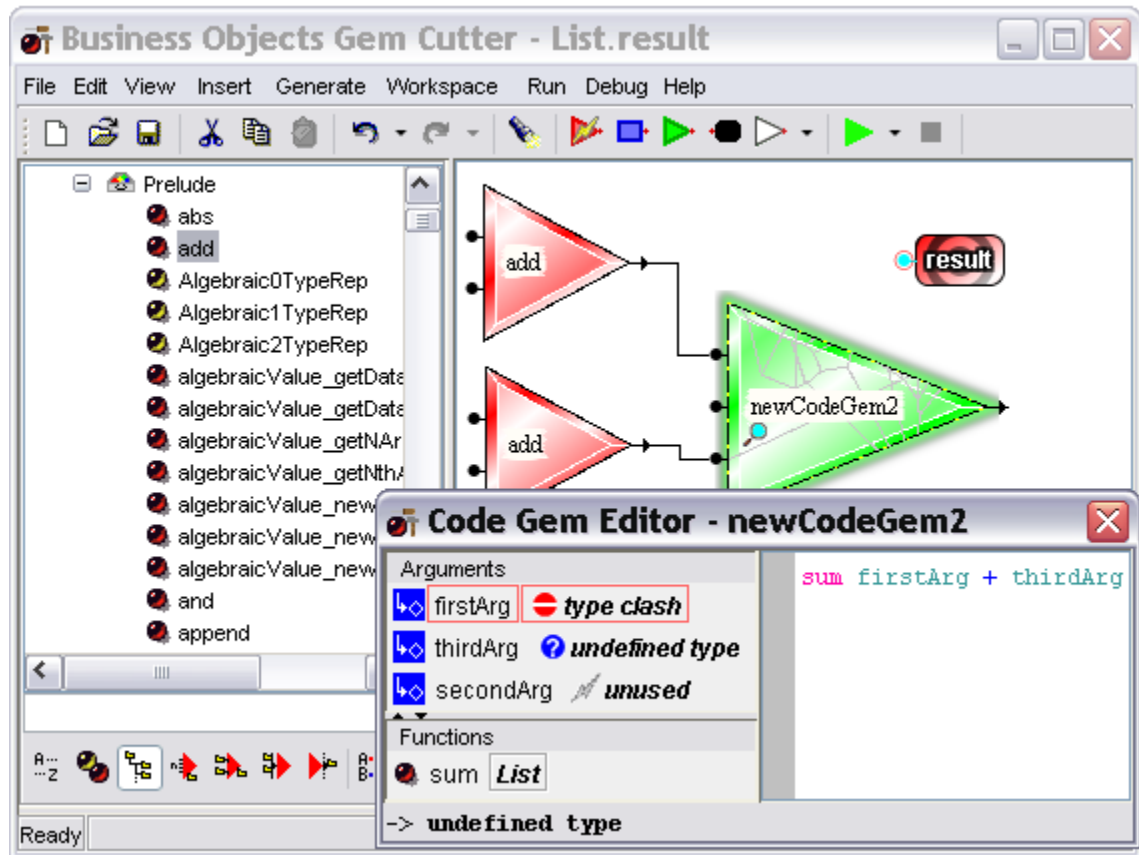
What happens if we try to remove a connected argument reference from the code? We'll try this by deleting the reference to `secondArg`. *Figure 7.26* shows the result:



**Figure 7.26. Code Gem Editor after removing `secondArg`**

The `firstArg` and `secondArg` arguments are held locked by the virtue of being connected to the add Gems. The `secondArg` argument is now marked as being connected, but unused. The definition of the Code Gem tells us that the result of the Gem will be the addition of the `firstArg` and `thirdArg` arguments, but the resultant Code Gem will continue to have three arguments (the unused argument will appear in the argument list, even though it is never used by the function). Anything connected to this defunct argument will effectively be 'pruned away' by the CAL runtime when the Gem is executed as `secondArg` can clearly have no impact on the result of the program. The Code Gem does rearrange order of the arguments slightly. It does this to try to keep the order of the arguments on the resulting function the same as the order of their appearance in the function definition. This is just a sensible 'default', and we'll see later that the argument order can easily be changed.

Now we'll take a look at what happens if we change the inferred type of a locked argument. Let's change the code such that it implies that first must be a list, as shown in *Figure 7.27*:



**Figure 7.27. Argument type clash in the Code Gem Editor**

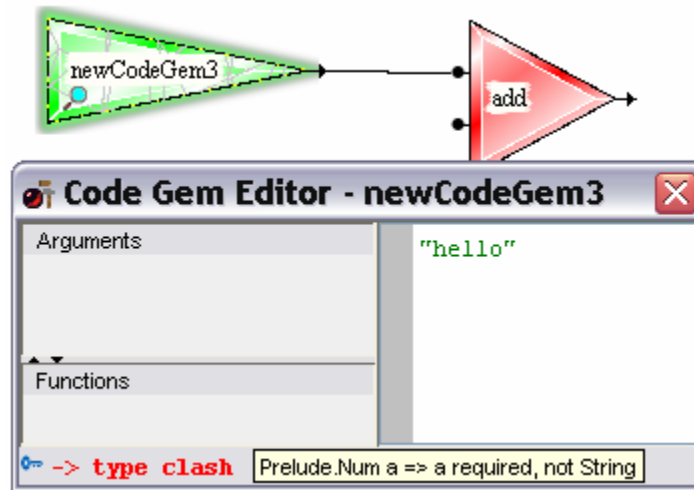
Notice that the argument `firstArg` is now flagged with a type clash error. The tooltip for this error in Figure 7.28 shows us that the argument was locked to a `Num` and that a list of `Num`s is incompatible with this type constraint:

`Prelude.Num a => a required, not Prelude.Num a => [a]`

**Figure 7.28. Argument type clash error tooltip**

This problem has also caused our Gem to 'break'.

A similar situation exists for the output of a Code Gem. If we connect the output to another Gem, we establish a type constraint for the output type. Failing to comply with this constraint will result in a 'broken' Code Gem and a flagged type error. This time, the type error will be displayed on the status line where we normally see the inferred output type, as shown in Figure 7.29:

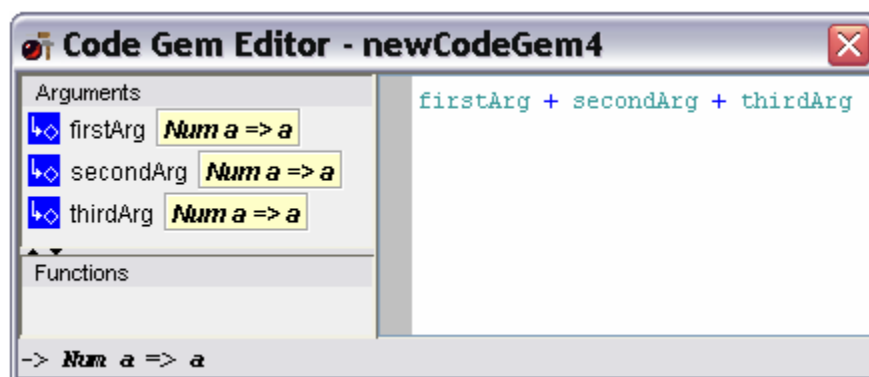


**Figure 7.29. Output type clash in the Code Gem Editor**

Here, we can see that the code has been changed to return a string, but an output type constraint exists for the Code Gem as it is connected to an `add` Gem. The type clash error is flagged on the status line and the tooltip tells us the details about the type incompatibility.

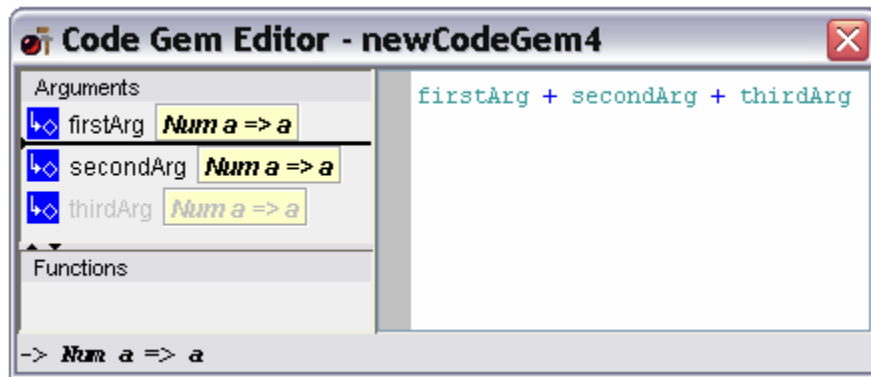
Returning to our original definition, adding three arguments together, it is often important to be able to reorder the arguments of a function. This becomes important if we want to allow our resulting Gem to be easily used in partial evaluation. As such, we normally want the arguments that are more likely to be left unbound to occur at the end of the argument list. So, it's good practice to think about this when developing a Code Gem. How do we force the order of the arguments? Well, one way is simply to ensure that the arguments' variable names appear in the desired order in the syntax of the Gem's CAL code. Doing this will allow the default ordering to take care of making sure the arguments are in the correct order. However, it is not always convenient, or even possible, to force the code to reflect the desired argument order. Instead, the Code Gem editor allows you to order the arguments manually.

Starting with default ordering and our earlier example, the Code Gem editor will appear like *Figure 7.30*:



**Figure 7.30. Code Gem Editor before argument reordering**

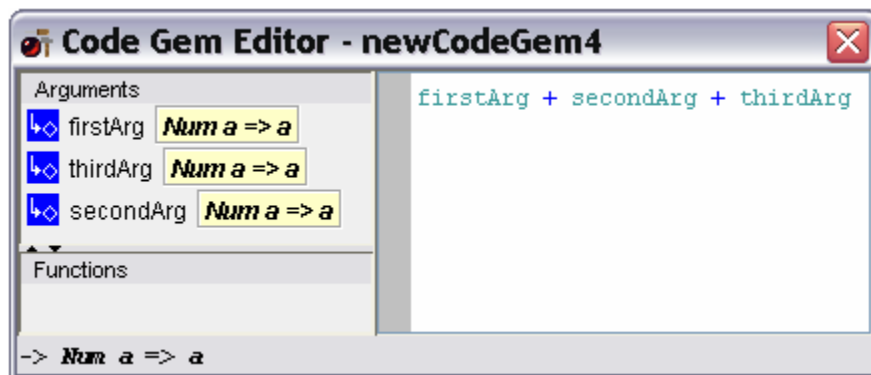
Let's suppose that for reasons of being able to use the function naturally in future applications, we decide that we would like to have *thirdArg* come before *secondArg* in the argument list. To achieve this, click on the *thirdArg* entry in the variables panel on the left and drag it up above the *secondArg* entry. As we drag, an insertion line will indicate where the argument will be positioned if we release the mouse button. We want to release it when it looks like the image in *Figure 7.31*:



**Figure 7.31. Arguments being reordered in the Code Gem Editor**

Notice the item we are dragging appears grey and an insertion line is drawn above the *secondArg* entry. The cursor also changes to indicate that we are dragging.

When the mouse is released in the situation shown above, *thirdArg* is repositioned in the variable panel to indicate that it is now the second argument of the function. This situation is shown in *Figure 7.32*:



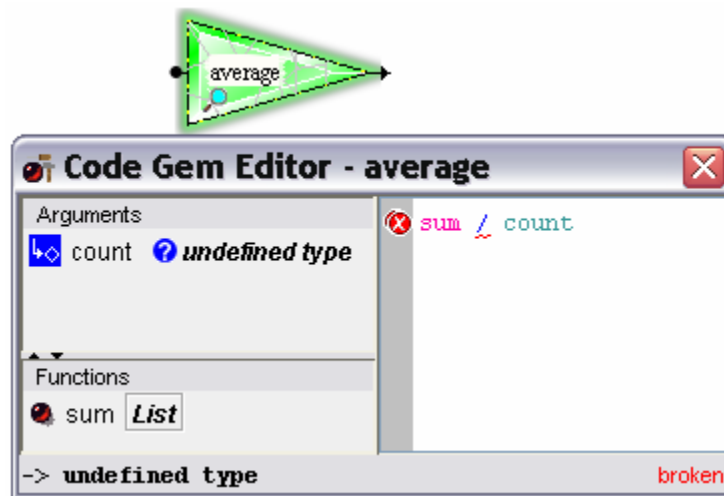
**Figure 7.32. Code Gem Editor after argument reordering**

Function calls appear in the panel on the left below the Arguments panel. These indicate that the variable has been inferred or set by the user to mean a call to another function. The fact that this is a call will also be indicated in the code through syntax highlighting.

In some cases, the inferred class of the variable (argument or function call) will not be correct. This happens because the editor assumes that any variable names it discovers which match the

names of other recognised functions are calls to that function. If this were always the case, then you would not be able to reuse the names of functions for arguments, which could be very awkward! Instead, where an incorrect assumption has been made about a call to an external function, the user is allowed to assert that the variable is actually an argument. This can be done simply by clicking on the name in the Arguments or Functions panel, and dragging it to the other panel. Alternatively, right-clicking on the argument or function name in one of the panels on the left will display a context menu allowing the user to choose whether this identifier should be treated as an argument or a function.

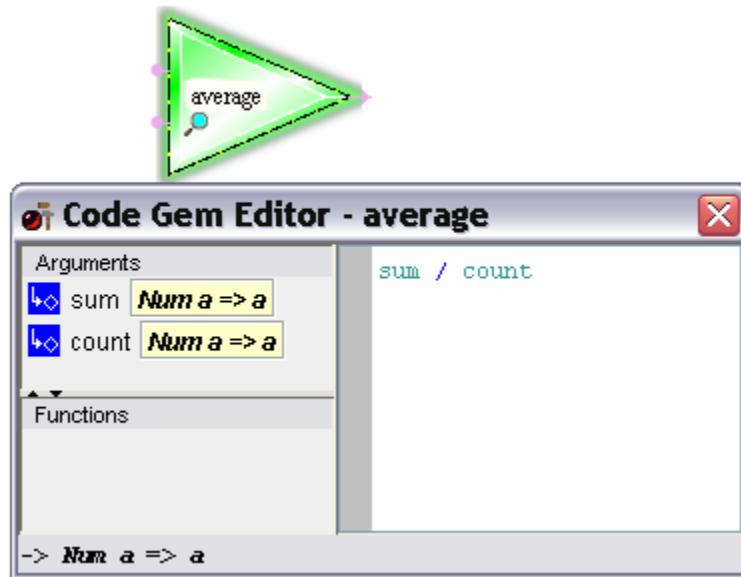
For example, suppose we have the Code Gem in *Figure 7.33*:



**Figure 7.33.** average Code Gem and editor

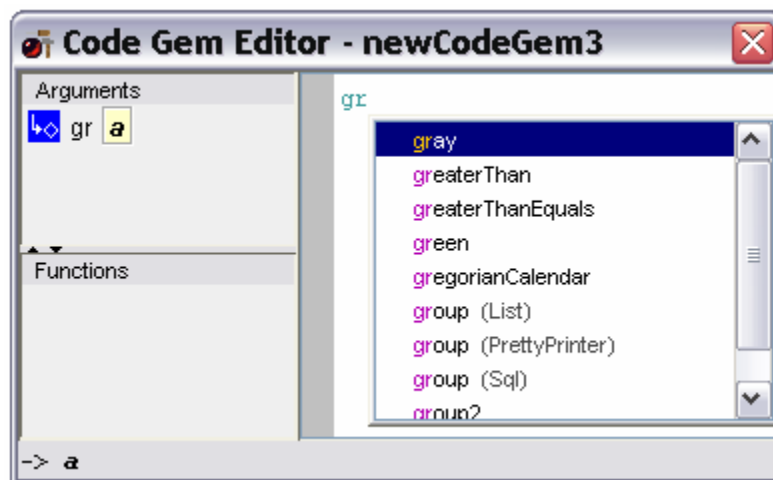
The desired functionality of the Gem is that it will take a sum of some values and a count of the number of values as inputs, and divide the sum by the count to produce an average value. However, the Gem Cutter has inferred that the variable `sum` refers to the function `List.sum` (a function to sum the values in a list), instead of treating it as an argument to the Code Gem, as we want. This has caused the Gem to break.

Fixing this is done by dragging `sum` from the Functions panel to the Arguments panel, or by right-clicking it and selecting the **Argument** option from the context menu. *Figure 7.34* shows the completed Code Gem:



**Figure 7.34.** *average* Code Gem after changing function name to argument

The Code Gem Editor also offers code completion for function and data constructor names. To invoke this, press Ctrl+Space while typing in the editor window. A window will pop up and give a list of all the possible functions and data constructors that match the text that has already been typed. An example is shown in *Figure 7.35*:

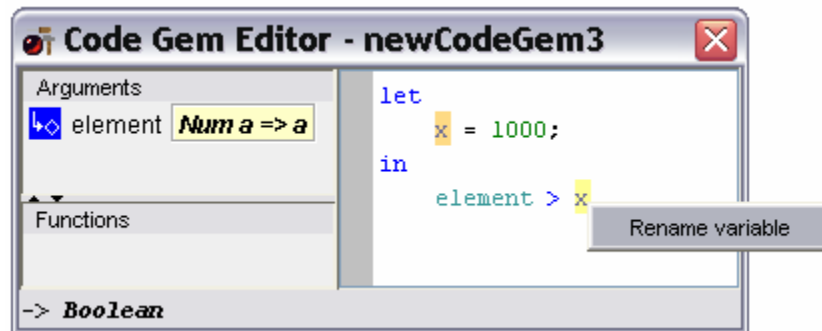


**Figure 7.35.** Code completion in the Code Gem Editor

Any of the functions in the list can be selected and inserted into the code by clicking the desired function with the mouse, or by moving to the desired function with the arrow keys and pressing **Enter**.

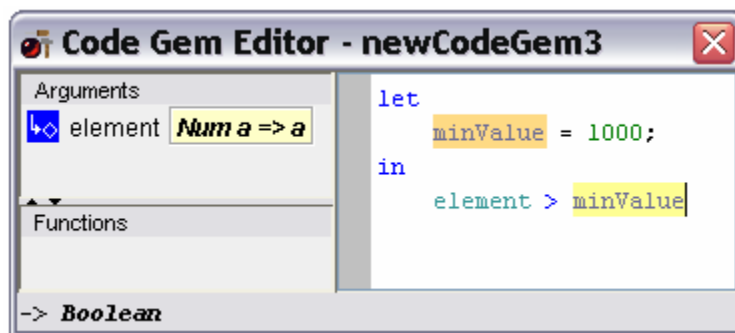
Another convenient way to insert functions into the code in the Code Gem Editor is by dragging and dropping them from the Gem Browser. This is done the same way as dropping a Gem on the Table Top, except that you drop it in the appropriate place in your code in the editor window.

The Code Gem Editor includes an easy method to rename local variables (variables defined in a `let` expression). To rename a local variable, right-click on an instance of the variable to open its context menu, shown in *Figure 7.36*:



**Figure 7.36. Renaming a variable in the Code Gem Editor**

Select the **Rename variable** option. Typing a new variable name will now cause every instance of this variable name in the code to be updated. For example, we can change variable `x` to `minValue` as displayed in *Figure 7.37*:



**Figure 7.37. Code Gem Editor after renaming a variable**

Both of the locations of `x` in the code are immediately replaced with the new variable name.

A feature available from the Table Top of the Gem Cutter is the ability to copy the code from Code Gems into the system buffer. To access this feature, right-click on the Table Top to open a context menu and select the **Copy Special → Code Gems** option. This copies the code from all the Code Gems into the system buffer so that it can be pasted into another location. Alternatively, the code from a single Code Gem, or a selection of Code Gems, can be copied by selecting the desired Code Gems, then right-clicking on a selected Code Gem and invoking the **Copy Special → Code Gems** option.

### 3. Collectors and Emitters

*Collectors* are special gems which collect a result. The purpose of a Collector is to identify a result, and to name the result. Collectors can be run, causing the result to be evaluated. The main

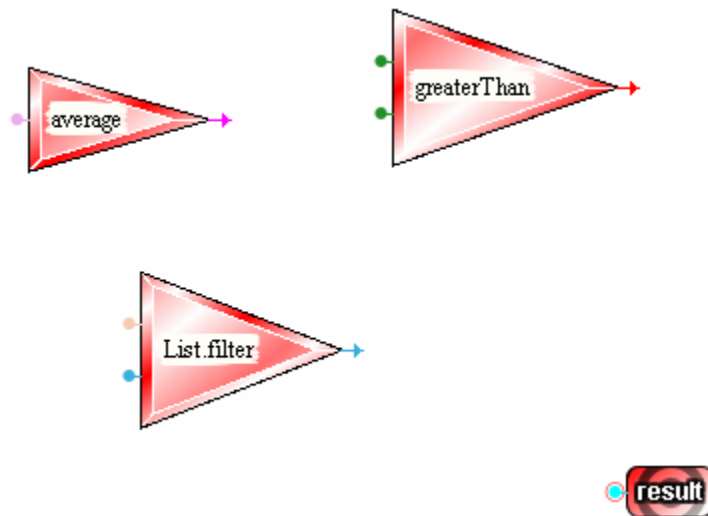


use of a Collector, however, is to give a name to a value so that the value can be used as an input to other Gems.

Having created a Collector and named it, and connected it to the output of a Gem, the user has effectively created a 'variable'. This will store the result. In order to use the same result in further computation within the Gem being constructed, the user creates *Emitters* as required. Each Emitter can then be used to connect the result to other Gem inputs. This effectively allows one output to be 'split' in order to connect to multiple inputs of other Gems.

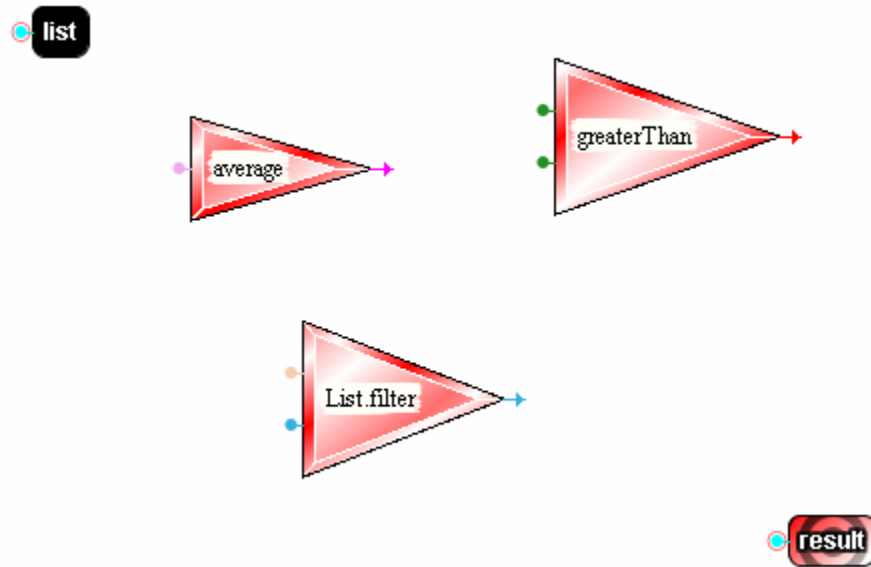
Here's a simple example. Let's say we want to filter out all the elements of a numeric list which are above the average (of all the elements in the list). Even with this simple definition, we can see that 'elements of a numeric list' occurs twice. This is a clue to the fact that we will require a variable.

Let's start by getting the obvious components, a *filter*, a *greaterThan* and an *average* Gem, and placing them on the Table Top such as in *Figure 7.38*:



**Figure 7.38. Gems on Table Top**

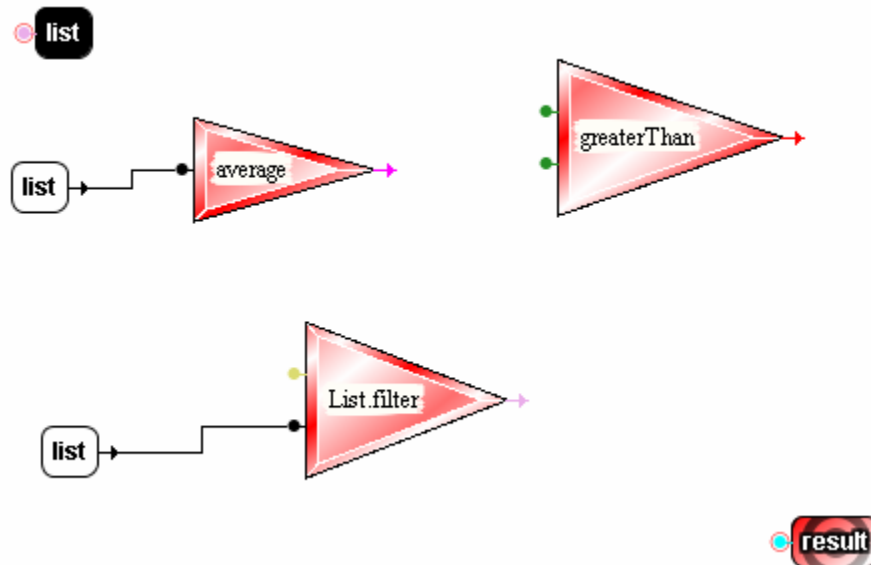
We want to create a 'variable' representing 'the elements of a list'. To do this, we place a Collector Gem on the Table Top by selecting the **Add Collector Gem** (●) button in the Toolbar, and clicking a blank area on the Table Top. Next, we rename the new Collector Gem to `list` by right-clicking on the Gem and selecting the **Rename Gem** option, or simply by double-clicking the Collector Gem. The result looks like *Figure 7.39*:



**Figure 7.39. Table Top with Collector Gem added**

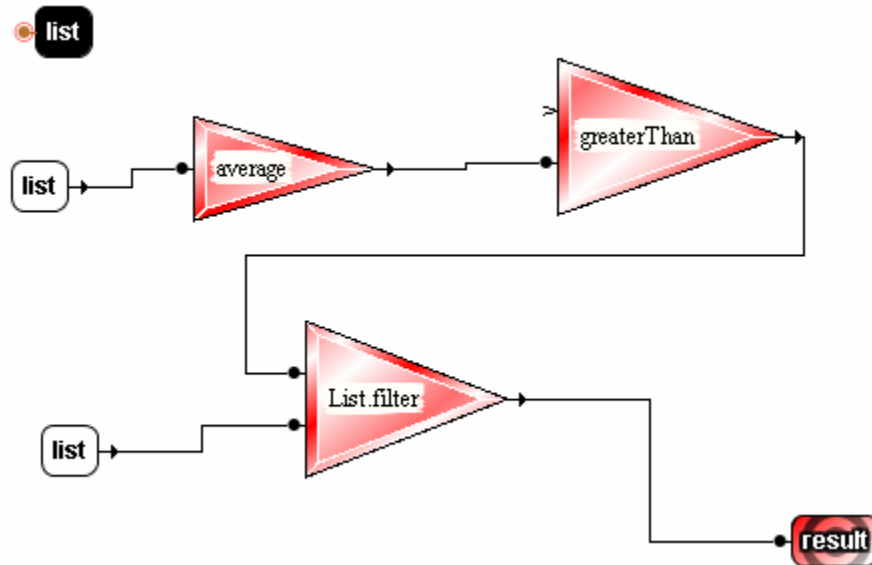
The next step is to find the average of our list. To do this, we place an Emitter Gem by clicking on the down arrow next to the **Add Emitter Gem** (▶) button in the Toolbar, selecting `list`, and clicking on a blank area in the Table Top. Once the new Emitter has been placed, we connect it to the input of the `average` Gem.

We also need to filter this list. In order to set our list as input to the `filter` Gem, we must place another Emitter Gem, and connect it to the second input of the `filter` Gem. The result should look like *Figure 7.40*:



**Figure 7.40. Emitter Gems connected on Table Top**

We want the list to be filtered so that only the elements greater than the average are given in the result. Attaching the output of the `average` Gem to the second input of the `greaterThan` Gem sets the `greaterThan` Gem up to compare numbers to the average value of the list. The output of the `greaterThan` Gem can now be connected to the first input of the `filter` Gem. Note that when this is done, the first input to the `greaterThan` Gem is automatically burnt (see *Chapter 6, Burning Inputs*) so that it is a function of the correct type required by the `filter` Gem. Lastly, the output of the `filter` Gem is connected to the result target. The completed Gem looks like *Figure 7.41*:

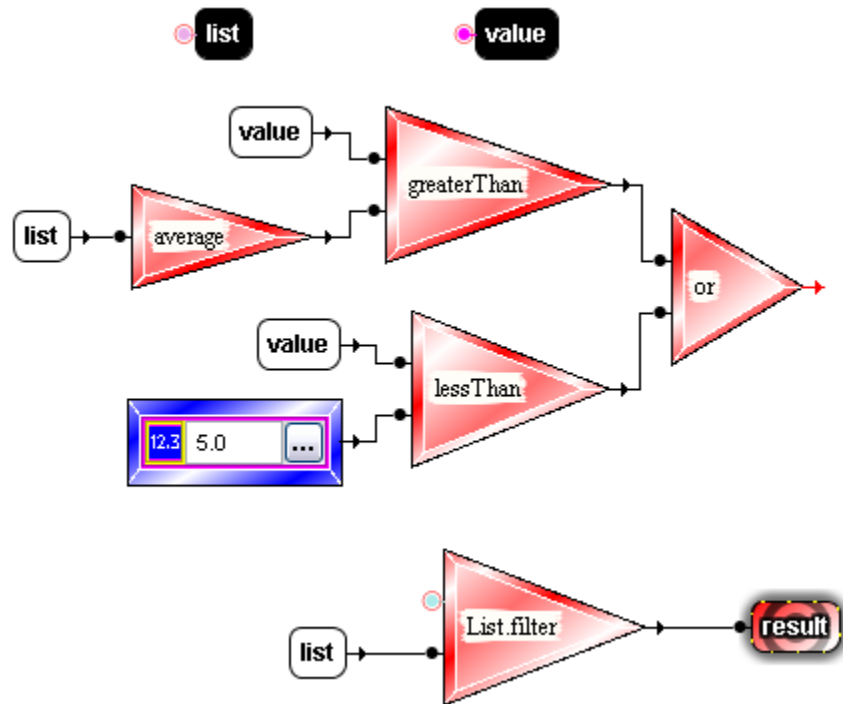


**Figure 7.41. Completed filtering Gem layout**

## 3.1. Creating Local Functions

In the previous example demonstrating the use of Collector and Emitter Gems, we created a Gem which could filter a list and return only list elements greater than the average of the list. What if we wanted to modify the filtering function so that elements will be returned if they are greater than the list average, or if they are less than five?

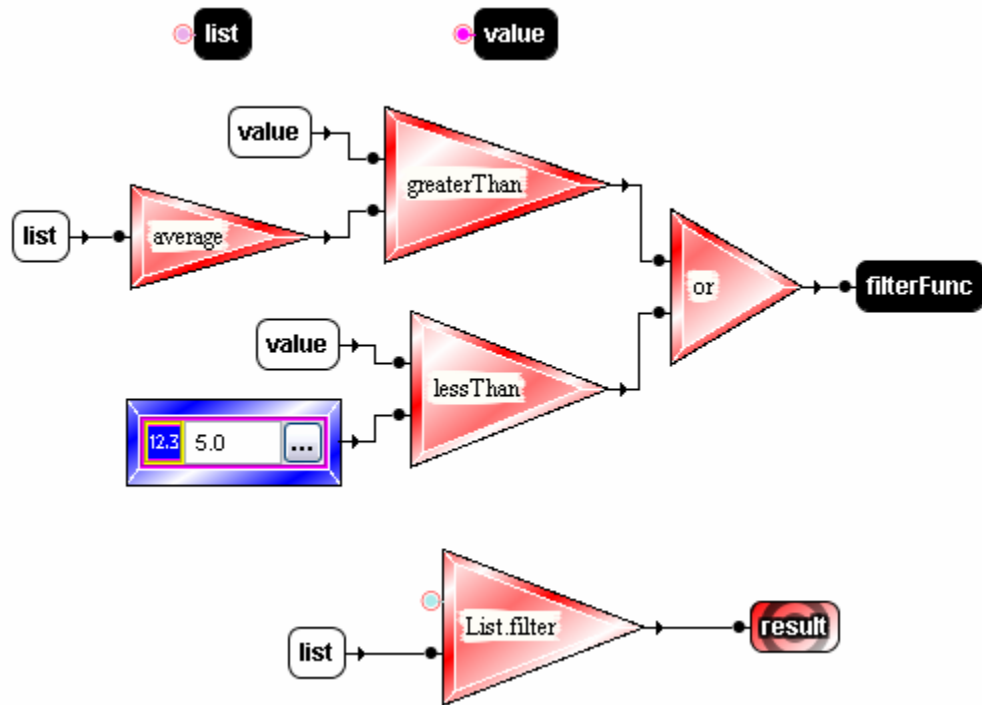
The Gem layout in *Figure 7.41* can be modified to look like *Figure 7.42*:



**Figure 7.42. Modified filtering Gem layout**

We would like to somehow connect the `or` Gem to the first input of the `filter` Gem, as was done in the previous example with the `greaterThan` Gem. This time, however, we have a problem: the input we would like to burn is not on the `or` Gem at all, but instead is represented by the collector labelled `value`. How, then, do we get the Gem Cutter to recognize that we want the tree of Gems connected to the `or` Gem to be treated as the predicate function to be input to the `filter` Gem, with the `value` input burnt?

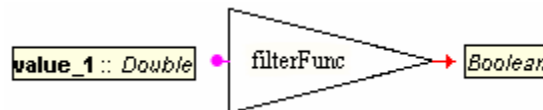
The solution is to create a local function. To do this, first connect a Collector Gem to the output of the `or` Gem, and rename it with an appropriate name for the local function. In this case, it has been named `filterFunc`, as in *Figure 7.43*:



**Figure 7.43. Modified filtering Gem layout with `filterFunc` collector**

Now, it is necessary to specify that the input to the Collector named `value` should be treated as an input to our local function, `filterFunc`, instead of to the result target. To do this, right-click on the input to the `value` Collector, select the **Retarget Input** option from the menu, and select `filterFunc` from the submenu that appears. Note that the circle around the input on the `value` Collector changes colour from red to black, indicating that this input is now targeting another collector instead of the result target (compare *Figure 7.43* with *Figure 7.45*).

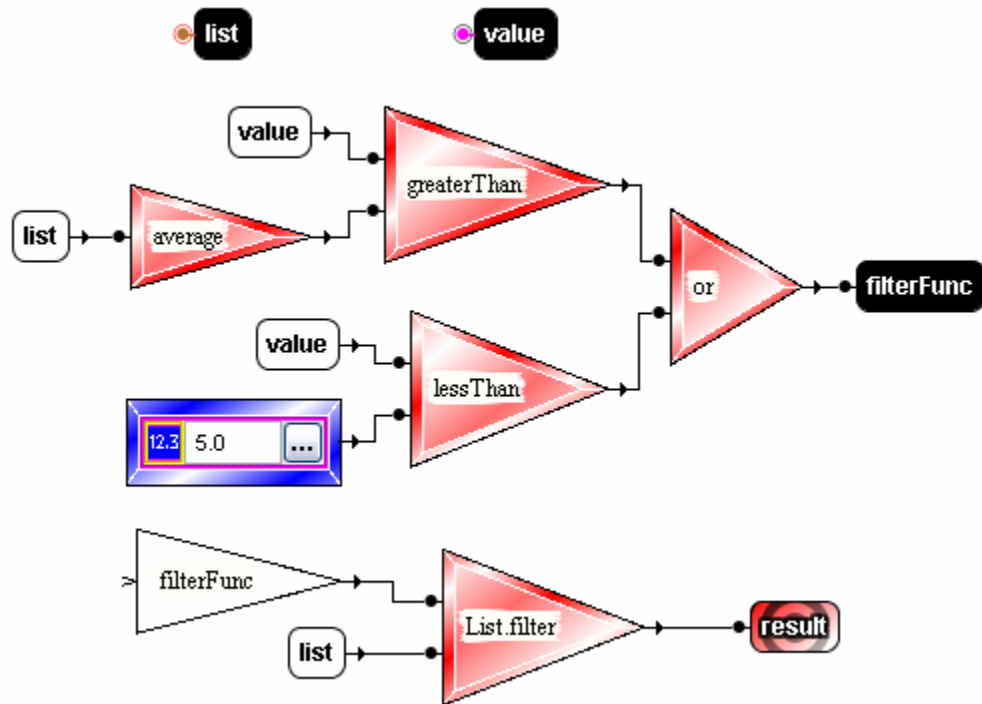
The local function is now complete! To use it, simply place an Emitter Gem for `filterFunc`. *Figure 7.44* shows what it will look like:



**Figure 7.44. `filterFunc` Emitter Gem**

You will notice that the Emitter now has a triangle shape like a regular function Gem, and has an input in addition to its output. Instead of representing a value, this Emitter now represents a function that takes a `Double` as input and returns a `Boolean`.

To complete the Gem, all that is left to do is burn the input to the `filterFunc` Emitter Gem, and connect it to the first input of the `filter` Gem. Here is the finished Gem in *Figure 7.45*:



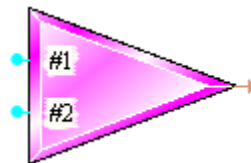
**Figure 7.45. Completed modified filtering Gem using local function**

Local functions are very useful when a simple function is needed inside the function (Gem) being designed, and it is not desirable to save this inner function separately.

## 4. The Record Creation Gem

The *Record Creation Gem* is used to create a record via connections to its inputs.

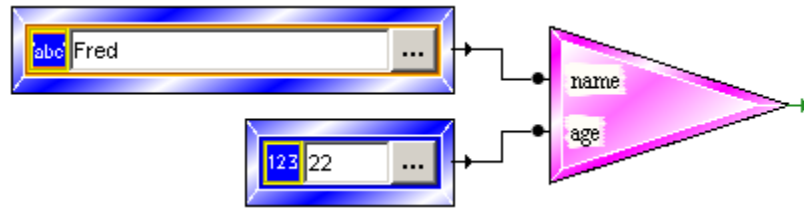
The Record Creation Gem is shown in *Figure 7.46*:



**Figure 7.46. Record Creation Gem**

The Record Creation Gem has one or more labeled inputs. Each of these labeled inputs represents a record field whose name is the label on the input, and whose value is any connection to that input. The output of the Gem is a record value comprised of fields whose names and values correspond to the Record Creation Gem's input labels and connections.

For instance, the output of the gem in *Figure 7.47* is a record value with two fields: the first field is named "name" and has a value of "Fred", and the second is named "age" with a value of "22". This represents the record value {name="Fred", value=22}.



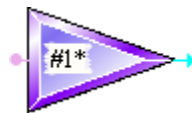
**Figure 7.47. Creation of a record with "name" and "age" fields**

When a record creation gem is first placed it has a two inputs, each of which are labeled with a default field name. An input's field name can be changed by double-clicking on its label and entering a new field name. Additional actions are available in the context menu when right-clicking on the Gem: **Add Field**, **Delete Field**, and **Rename Field** can be used to add new fields, delete existing fields, or rename fields (similar to double-clicking a field name) respectively.

## 5. The Record Field Selection Gem

The *Record Field Selection Gem* is a triangular purple Gem. It is often used when working with records. Its function is to obtain the value of a particular field from an existing record.

The Record Field Selection Gem is shown in *Figure 7.48*:



**Figure 7.48. Record Field Selection Gem**

The Record Field Selection Gem always has exactly one input connector. The name displayed on the Gem represents the name of the field that will be selected from a record that is input to the Gem.

When a record field selection gem is first created the field to select is automatic. This is indicated by an asterisk appended to the end of the field name. It means that any type of record can be connected to the input, providing any output constraints can be satisfied. When a connection is made, a combo box is displayed allowing the user to select a valid field. Once a field has been selected, the field becomes fixed, and can not change again automatically, even if the gem is disconnected and reconnected. In this state, only records which contain the appropriate field can be connected. However, the field can be modified explicitly at any time, either by double-clicking on the Gem, or by right-clicking on the Gem and selecting the **Change Field to Select** option from the context menu. This will allow the user to enter or select a field that is consistent with the input and output connections.

---

# Chapter 8. The Properties Browser

The Properties Browser is a section of the Gem Cutter that displays information about all the different modules and Gems in the current workspace. The Properties Browser is a useful tool for examining all the different Gems available for use. It can also be used to modify Gem metadata (data about the Gems), or write new metadata for newly created Gems.

## 1. Viewing Gem and Module Metadata

To open the Properties Browser, select the **View → Properties Browser...** menu option. Alternatively, to display the properties of a Gem or module directly, right-click on the Gem or module in the Gem Browser, or a Gem on the Table Top. From the context menu, select the **View Properties...** option.

*Figure 8.1* shows what the Properties Browser looks like:





Figure 8.1. The Properties Browser

The panel on the left side of the Properties Browser is called the Navigation Panel. The Navigation Panel allows for easy browsing of all the modules in the workspace, and all the Gems they contain. The layout of the Navigation Panel is similar to the Gem Browser, except that the Navigation Panel organizes the contents of each module into folders that organize Gems according to their CAL representations. These folders are:

- |           |   |
|-----------|---|
| Functions | The Functions folder contains a list of all the function (red) Gems in the module.  |
| Types     | The Types folder contains a list of all the data types defined in the module. Any type that has data constructors (yellow Gems) for it will have an expandable node for the type name, with the data constructors listed. |

Classes	The Classes folder contains a list of all the type classes defined in the module. Each type class has a list of class methods accessible by expanding the node containing the name of the type class.
Instances	The Instances folder contains a list of all the specific types that implement functions from type classes. These functions each have their own metadata, which gives details about the implementation of a type class function for a specific type.

The panel on the right side of the Properties Browser is where the information related to the Gem or module selected in the Navigation panel is displayed. At the top of this panel is a toolbar that looks like *Figure 8.2*:



**Figure 8.2. Properties Browser toolbar**

This toolbar contains **Back** and **Forward** buttons that function similarly to the buttons in a web browser, allowing the user to navigate back and forth between previously viewed pages in the Properties Browser. It also contains a button to toggle display of the **Navigation Panel** on or off, and a **Print** button to print the currently displayed metadata. The **Edit** button is used to edit the metadata for the item currently displayed (see *Section 2, “Editing Metadata”* for more information). Finally, basic search functionality is available in the Properties Browser. Typing a search string in the **Search** field and pressing Enter will search all metadata in the current workspace, and display the results below with hyperlinks to each topic found.

Under this is a bar displaying where the current page of metadata is in the tree. It also provides links to different sections of the current page of metadata. It is displayed in *Figure 8.3*:

```
Workspace >> Prelude >> Num >> add(Class Method)
DESCRIPTION | RETURN VALUE | ARGUMENTS | REQUIRED METHOD | EXAMPLES | GENERAL | RELATED FEATUR
```

**Figure 8.3. Section of Properties Browser navigation bar**

This particular screenshot indicates that the metadata currently being viewed is for the `add` function. This function is a method of the `Num` type class, which is part of the `Prelude` module in the workspace. The `Workspace`, `Prelude` and `Num` text in this example provide links directly to each of these sections.

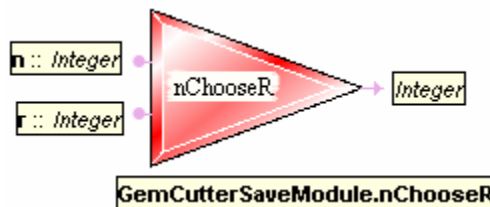
The links in the second row of this section of the Properties Browser direct the user to different sections in the current page of metadata.

## 2. Editing Metadata

The Properties Browser provides an easy way to edit metadata for Gems and modules. This is a particularly useful feature when you are creating new Gems, and want to provide documentation about their functionality.

To start editing the metadata of a Gem or module, open the Properties Browser to the page for the entity you want to edit, and click the **Edit** button in the toolbar. Alternatively, right-clicking on an entity in the Gem Browser window, and selecting the **Edit Properties** option from the context menu will open the Properties Browser in editing mode.

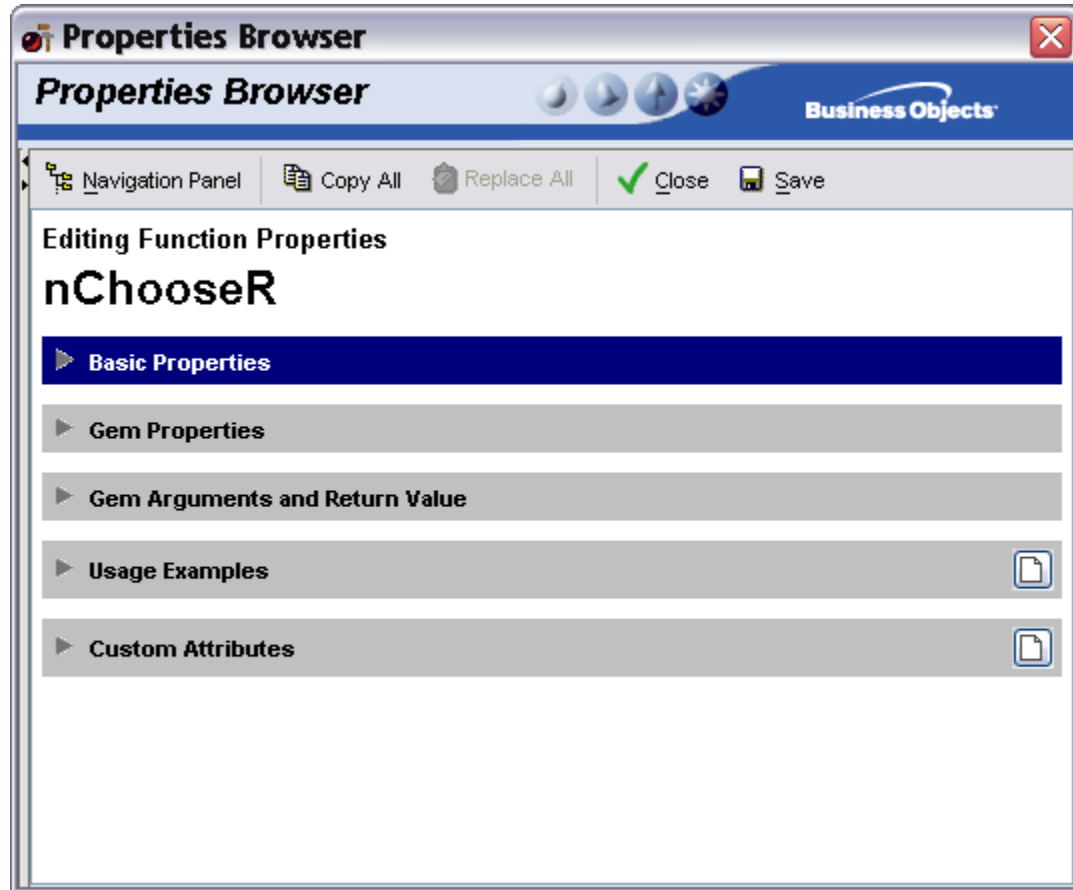
Let's take a look at an example of creating metadata for a newly designed Gem. In this example, a Gem called `nChooseR` has just been designed and saved to the `GemCutterSaveModule` module. The function of this Gem is to calculate the number of combinations of  $r$  elements that can be selected from a group of  $n$  elements. The Gem with its tooltips displayed is shown in *Figure 8.4*:



**Figure 8.4.** `nChooseR` Gem with default tooltips

Note that the default tooltips are not very informative. Adding metadata to this Gem is desirable, in part, because it will add information to these tooltips to better explain the functionality of the Gem to someone who has never used it before.

Entering Edit mode, the Properties Browser looks like *Figure 8.5*:



**Figure 8.5. Properties Browser in edit mode**

Each of the five sections listed can be expanded in order to edit the metadata contained within. Clicking on the **Basic Properties** option displays the pane in *Figure 8.6*:

**Figure 8.6. Basic Properties panel**

The **Basic Properties** section is used to edit basic information about the Gem. Editing each of the fields has the following effects:

Display Name	Changes the displayed name of the Gem in the Properties Browser. If nothing is entered, the displayed name defaults to the name given to the Gem when it was saved. This field is useful if it is desired to have a Gem with spaces in its name in metadata, something which the Gem Cutter does not allow in its name for a Gem.
Short Description	A brief description of the Gem. This description will be displayed as a tooltip when the mouse pointer is moved over the Gem in the Gem Browser or on the Table Top.
Long Description	A more detailed description of the Gem. This description will appear in the Properties Browser when viewing the Gem's properties. It will also display as a tooltip if the <b>Short Description</b> section is left blank.

Author	The author of the Gem.
Version	The version of the Gem.
Preferred	Metadata attribute indicating if this is a “preferred” Gem – can be used by other applications to filter Gems.
Expert	Metadata attribute indicating if this is an “expert” Gem – can be used by other applications to filter Gems.
Related Features	A list of other Gem Cutter features that are related to the current entity being edited. Modules, types, type classes and Gems can be selected from the list on the left (the current workspace) and added to the list on the right, which represents this Gem’s related features.

Displayed in *Figure 8.7* is the pane after entering relevant information for the `nChooseR` Gem:

**Basic Properties**

Display Name:

Short Description:

Long Description:

Author:

Version:

Preferred: ☐ Yes ☒ No

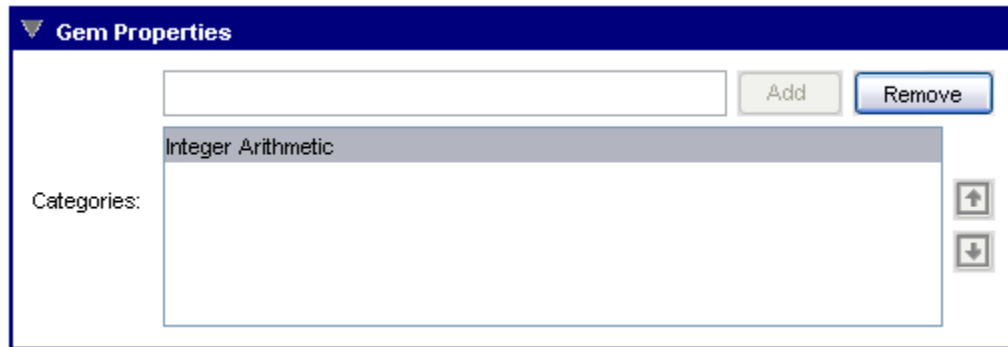
Expert: ☐ Yes ☒ No

Related Features:

- ☐ Accumulate
- ☐ Array
- ☐ ArrayPrimitives
- ☐ Bits
- ☐ Char
- ☐ Color
- ☐ DataGems
- ☐ DatabaseMetadata
- ☐ Debug

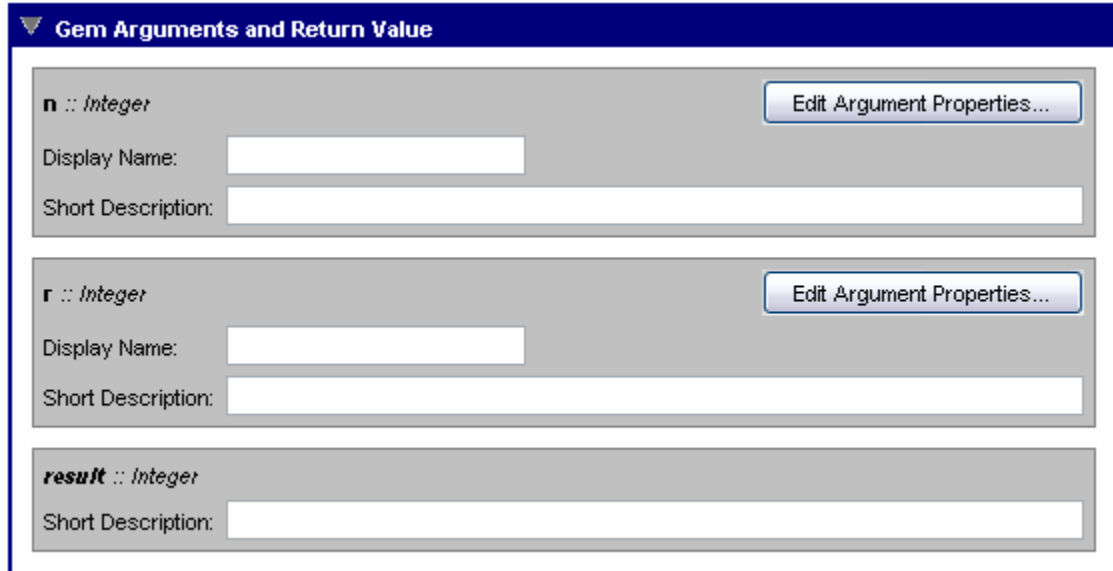
**Figure 8.7. Basic Properties panel with metadata entered**

The next entry section is **Gem Properties**. The purpose of this section is to specify any categories the Gem is a part of. These categories can then be used by other applications that are organizing Gems by examining their metadata. To add a category, type its name in the box at the top and click the **Add** button. To remove a category from the list, select it and click the **Remove** button. *Figure 8.8* shows the **Gem Properties** panel after adding a category:



**Figure 8.8. Gem Properties panel**

The **Gem Arguments and Return Value** section allows the user to modify metadata pertaining to the arguments and return value of the Gem. This is displayed in *Figure 8.9*:



**Figure 8.9. Gem Arguments and Return Value panel**

For each argument, a display name and short description can be entered. These two options function the same way for arguments as they do for Gems (see above). Clicking on the **Edit Argument Properties...** button will open up a whole new edit page for the argument, which will be similar to the page for Gems.

The return type is the argument at the bottom, and can be given a short description that will appear as a tooltip.

After entering results, the **Gem Arguments and Return Value** panel looks like *Figure 8.10* for the `nChooseR` Gem:

**Gem Arguments and Return Value**

**n :: Integer** Edit Argument Properties...

Display Name:

Short Description:

**r :: Integer** Edit Argument Properties...

Display Name:

Short Description:

**result :: Integer**

Short Description:

**Figure 8.10. Gem Arguments and Return Value panel with metadata entered**

The next pane of data is titled **Usage Examples**. This section allows examples of code using this Gem to be displayed and calculated. Initially, this section is empty, as shown in *Figure 8.11*:

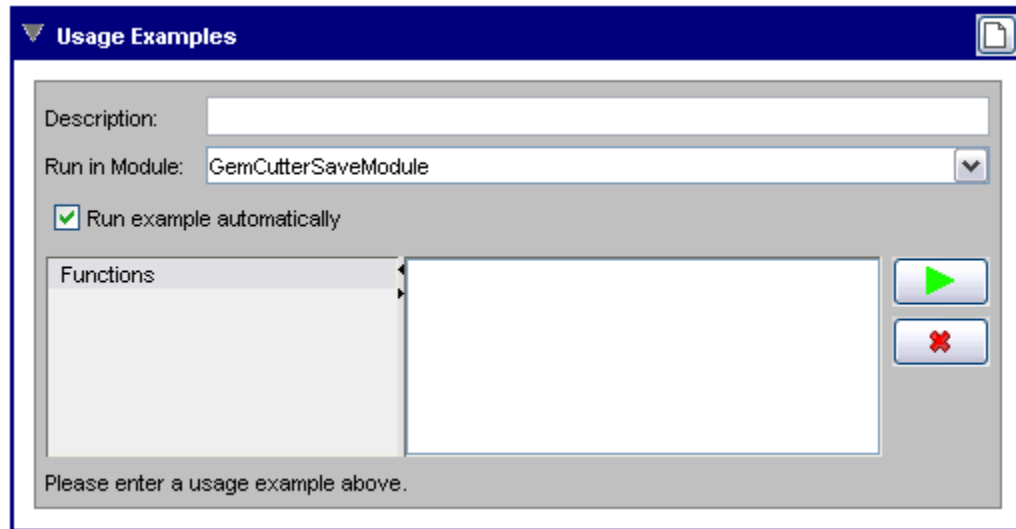
**Usage Examples** New

There are no existing usage examples. Click the button to add a new example.

**Figure 8.11. Usage Examples panel**

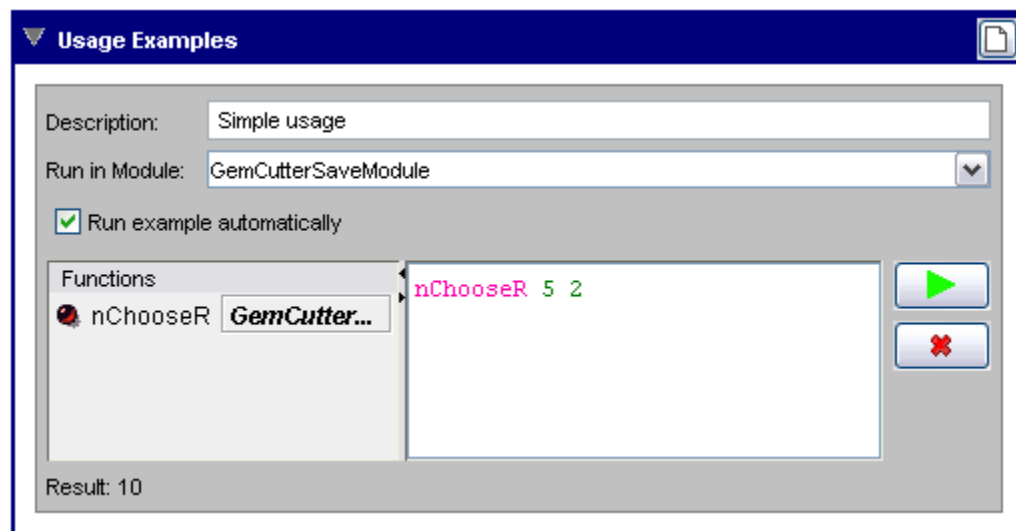
Clicking on the **New** (📄) button in the top right corner will add a new example to the section, shown in *Figure 8.12*:





**Figure 8.12. Usage Examples panel with new example**

Notice that the section at the bottom appears similar to the Code Gem editor. A piece of CAL code can be entered here to demonstrate the functionality of the Gem. If the play button (green arrow) is pressed, the editor will display the result at the bottom of the pane, if possible. Our new usage example is shown in *Figure 8.13*:



**Figure 8.13. Usage Examples panel with completed example**

The final pane that appears is the **Custom Attributes** pane. This pane allows custom pairs of names and values to be added to the Gem metadata. To add a custom attribute, click the **New** button, and enter a name and value for the attribute. We will not add any custom attributes to the `nChooseR` Gem.

When editing of the Gem metadata is complete, click the **Save** button in the toolbar to save the metadata. Clicking the **Close** button will exit the metadata editor, and display the metadata panel for the `nChooseR` Gem, which should look like *Figure 8.14*:

## nChooseR

### Description

Combination function. Returns the number of combinations of elements of size  $r$  that can be selected from a size  $n$  group of elements. Also known as the binomial coefficient function.

### Return Value

**result** :: *Integer* - Number of unordered combinations of  $r$  elements chosen from set of size  $n$

### Arguments

**n** :: *Integer* - Number of elements in set to choose from

**r** :: *Integer* - Number of elements to choose from set

### Examples

#### Example 1

Description:	Simple example
Expression:	nChooseR 5 2
Result:	10

### General

Creation Date:	Monday, October 23, 2006 3:53:06 PM
Modification Date:	Monday, November 27, 2006 4:56:16 PM
Author:	Business Objects
Version:	1
Visibility:	public
Categories:	Integer Arithmetic

### Related Features

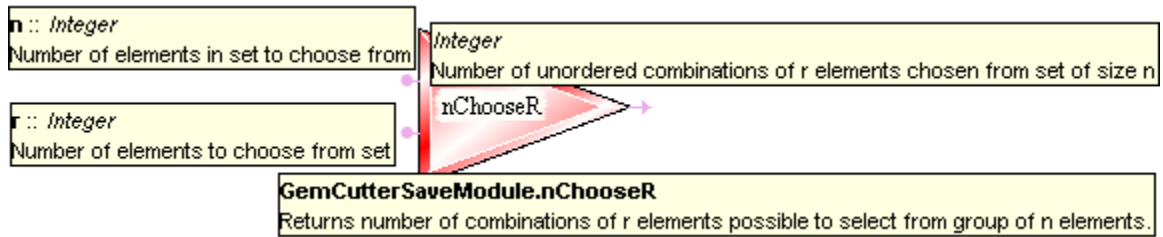
none

### Custom Attributes

none

**Figure 8.14. Metadata for the nChooseR Gem**

All the Gem metadata is now displayed in the standard format used by the Properties Browser. Also, *Figure 8.15* shows the Gem with its tooltips displayed after updating the metadata:



**Figure 8.15. nChooseR Gem with new tooltips**

This is now far more informative than the default tooltips for the Gem. The functionality of this Gem is now much clearer for someone who has never used this Gem before.

---

# Chapter 9. Miscellaneous Features

This section describes a number of useful miscellaneous Gem Cutter features.

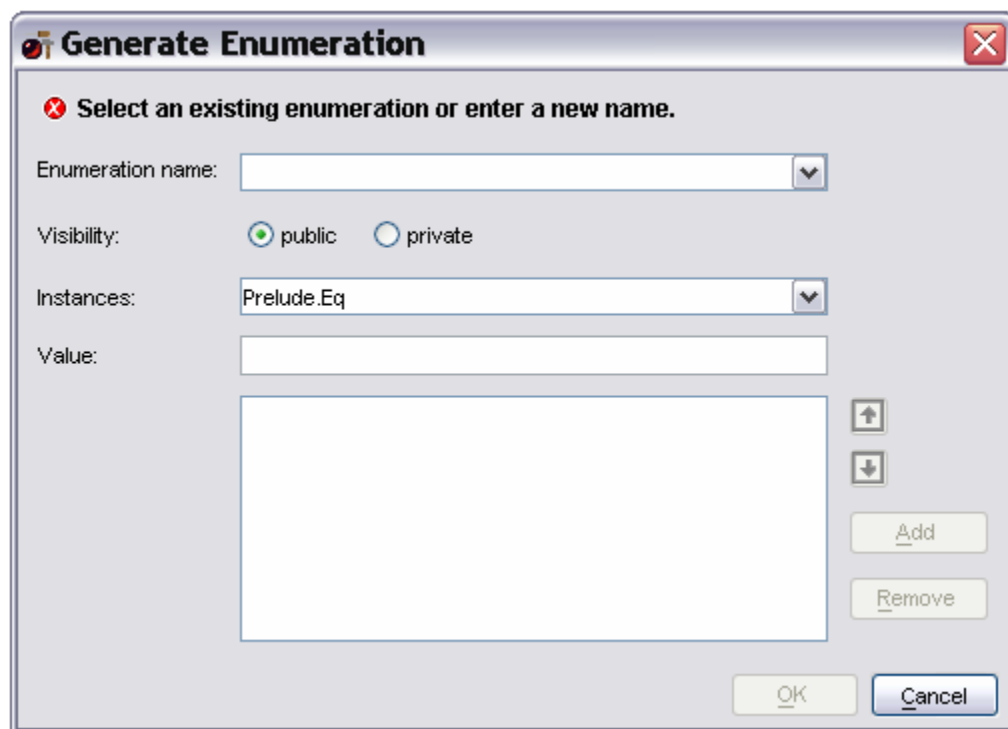
## 1. Renaming Entities

The Gem Cutter can rename an existing Gem, type, type class or module, and update all workspace references to the entity being renamed in Gem definitions and in metadata. To use this feature, select the **Workspace** → **Rename** menu option, and select the kind of entity to rename. A dialog will appear in which you can input the existing name of the entity to rename, and the new name. The Gem Cutter will then go through all source and metadata resources, and perform the requested change.

An alternative way to accomplish this for Gems and modules is to right-click on the Gem or module to be renamed in the Gem Browser, and select the **Rename (Gem/Module)...** option from the context menu.

## 2. Generating Enumerations

An enumeration is a type that contains a set of values that are part of the type, and whose data constructors have no arguments. To generate an enumeration, select the **Generate** → **Enumeration...** menu option. This will display the dialog box in *Figure 9.1*:

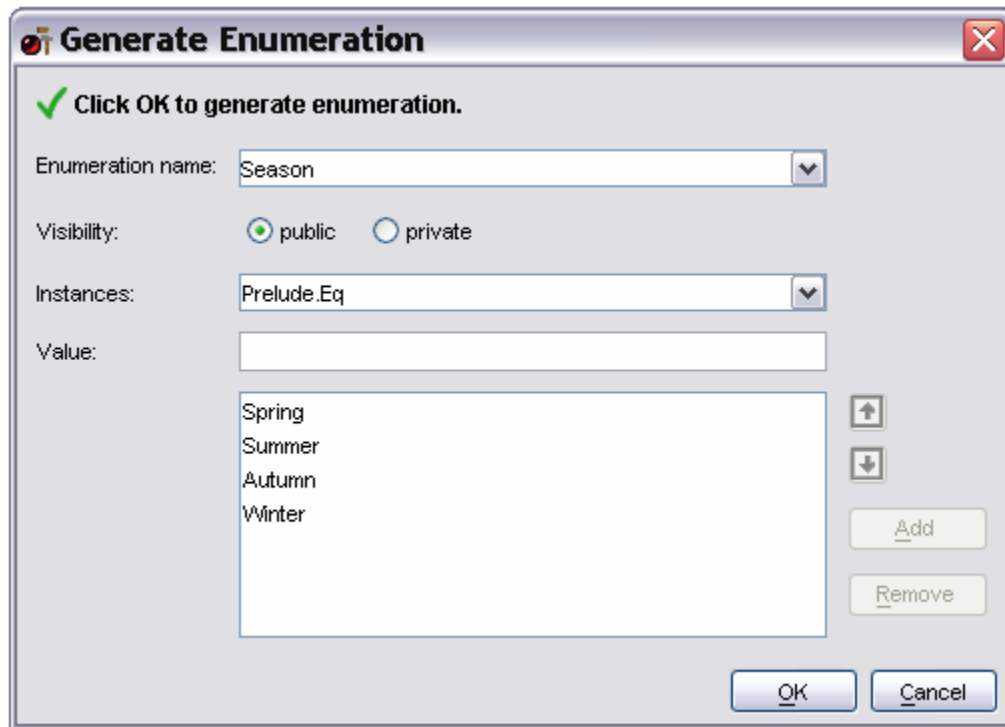


**Figure 9.1. Generate Enumeration dialog box**

Entering an existing enumeration name will allow modification of the enumeration. Entering a new enumeration name will allow a new enumeration to be created. An enumeration is created by naming it, then adding values by entering them in the **Value** field and pressing the **Add** button.

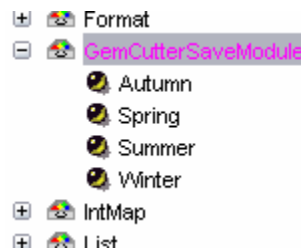
The **Instances** drop-down box is used to select which type classes the enumeration will implement. For example, if the enumeration implements `Prelude.Ord`, the elements in the enumeration will have an "order", and will be comparable with Gems such as `greaterThan` and `lessThan`.

For example, suppose we want to create an enumeration called `Season` that will enumerate the seasons in the year. After entering the enumeration name and values, the dialog looks like *Figure 9.2*:



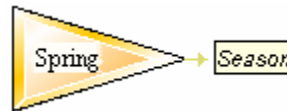
**Figure 9.2. Generating a Season enumeration**

Upon clicking **OK**, the enumeration is generated. A data constructor for each possible value in the enumeration is created and placed in the current module, as shown in the Gem Browser in *Figure 9.3*:



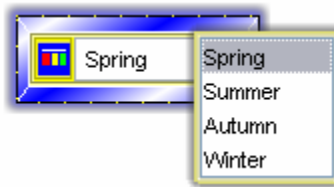
**Figure 9.3. Season enumeration in the Gem Browser**

*Figure 9.4* shows a sample data constructor for the enumeration:



**Figure 9.4.** `spring` data constructor

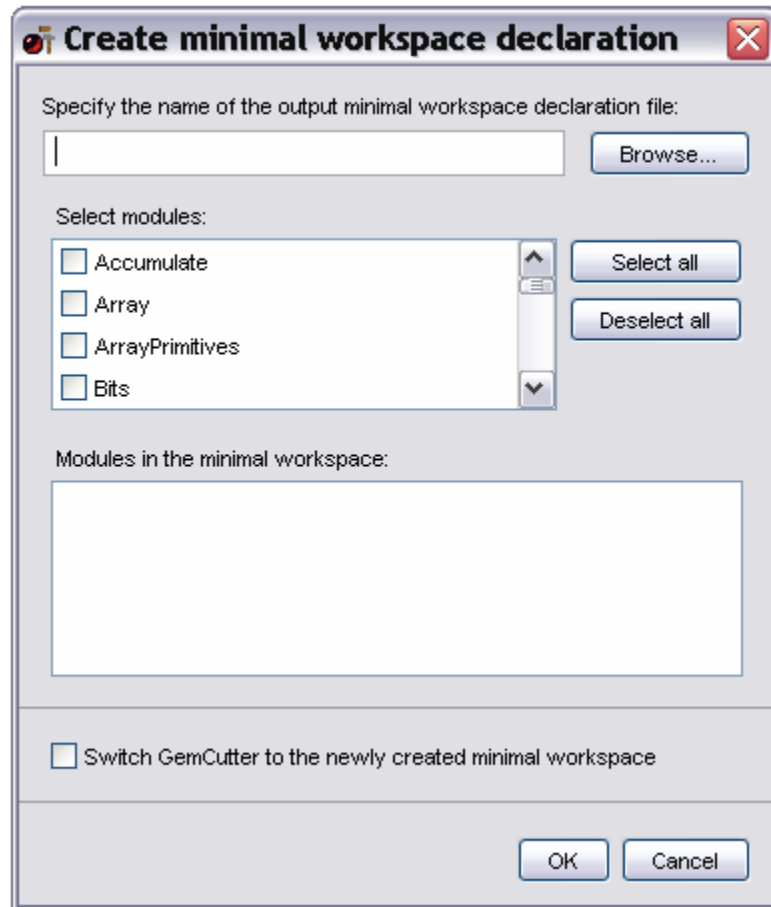
The enumeration can now be used in the construction of new Gems. Also, note from *Figure 9.5* that the enumeration is now a valid type to use in a Value Gem:



**Figure 9.5.** Value Gem representing the `Season` data type

### 3. Creating a New Minimal Workspace Declaration

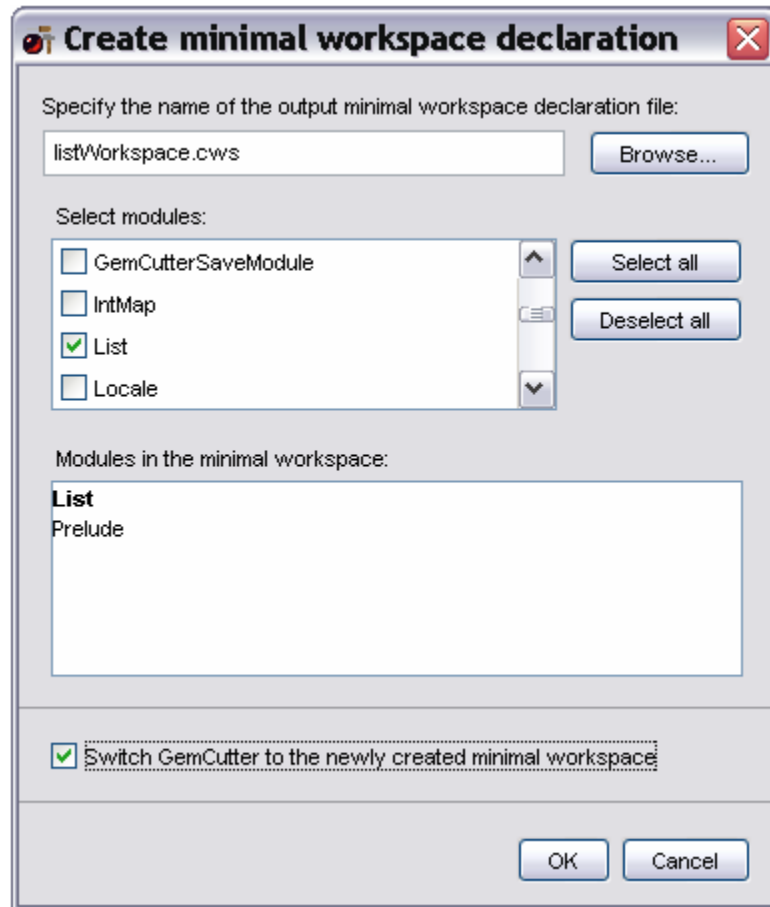
The Gem Cutter provides a feature to create a new workspace declaration. To create a new workspace declaration, select the **Workspace** → **Create Minimal Workspace Declaration...** option from the menu bar. The dialog in *Figure 9.6* will appear:



**Figure 9.6. Create Minimal Workspace Declaration dialog box**

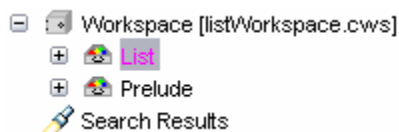
Suppose we want to create a workspace only containing the `List` module (the `Prelude` module will also be included, as it is required by all other modules). To do this, we enter a name for the workspace declaration file in the field at the top, or click the **Browse...** button to choose exactly where the workspace declaration file will be saved on the file system. Next, we check the box next to the `List` module in the **Select Modules** box. We can also check the checkbox at the bottom of the dialog to tell the Gem Cutter to switch to the new workspace immediately after it is created. The dialog now looks like *Figure 9.7*:





**Figure 9.7. Creating a minimal workspace including the `List` module**

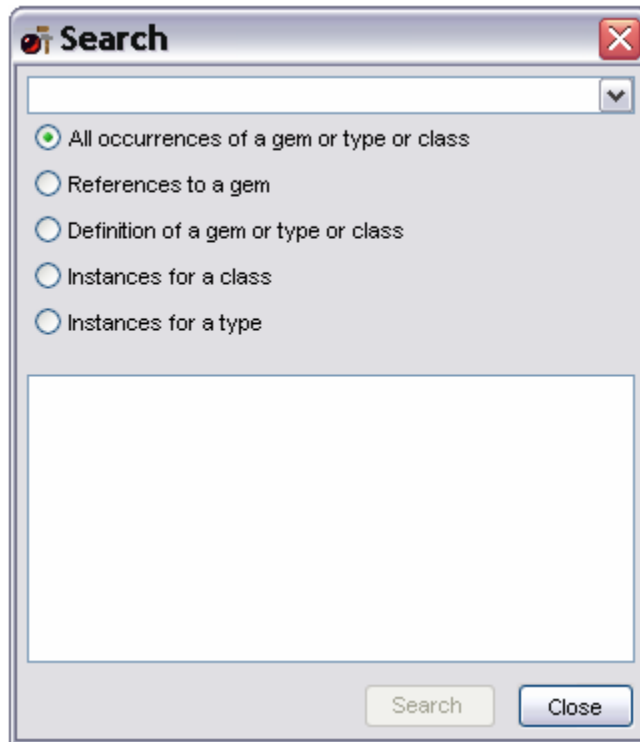
Clicking **OK** now generates the new workspace declaration file, and switches to the new workspace. The Gem Browser (*Figure 9.8*) now shows the contents of the new workspace:



**Figure 9.8. New minimal workspace in the Gem Browser**

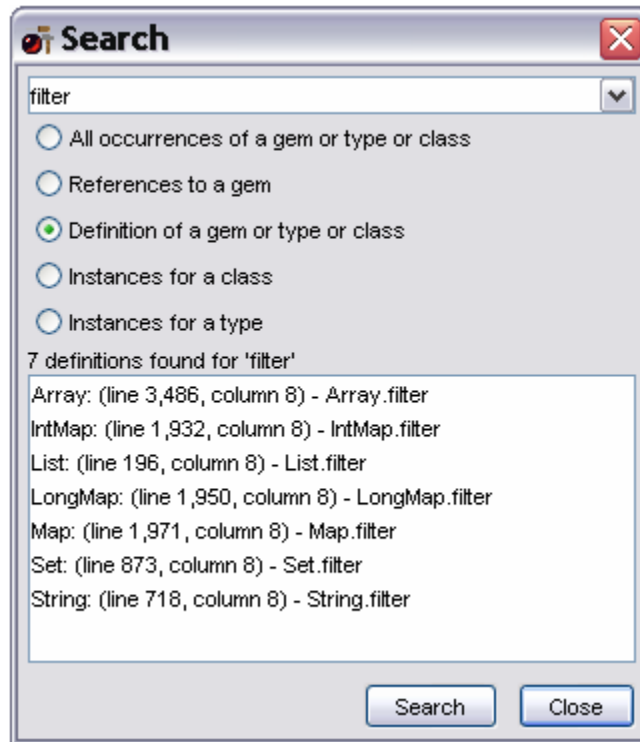
## 4. Searching Module Code

The Gem Cutter has a search feature which can be used to search the CAL code of all modules. To open the search dialog, select the **Edit** → **Search...** option from the menu bar, click the flashlight button on the toolbar, or press Ctrl+F. The search dialog shown in *Figure 9.9* will appear:



**Figure 9.9. Search dialog box**

To search for something, select the category of what to search for with the radio buttons, enter a search term in the field at the top of the dialog, and press **Enter**. The frame at the bottom of the window will display any occurrences of the entity that was searched for. *Figure 9.10* shows an example:



**Figure 9.10. Using the Search dialog box**

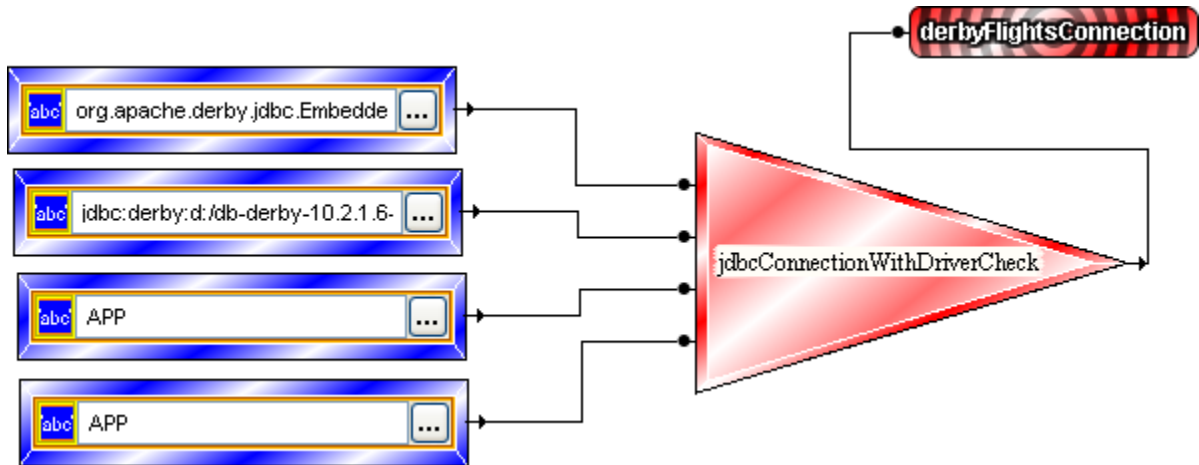
Clicking on one of the items in the list of locations found will open a text editor displaying the contents of the module file at the selected line.

## 5. Generating a JDBC Data Source

The Gem Cutter has a feature to allow easy creation of JDBC data sources. This is very useful for processing database records within the Gem Cutter. In order to use this feature you must first create a database connection gem. The easiest way to do this is to use the `DataGems.jdbcConnectionWithDriverCheck` gem. This gem has four inputs:

- JDBC driver location (fully qualified class name),
- the url of a JDBC database resource,
- the database login,
- and the database password.

To make a new database connection add this gem and four value gems to the table top. Enter the four values for your database into the value gems, and connect them to the four inputs. Connect the output to the target gem. Rename the target gem to something that is appropriate for your database connection, and save it. An example database connection gem design is shown in *Figure 9.11*



**Figure 9.11. Example Data Connection Gem Design**

Once you have created a database connection you are ready to create the JDBC datasource that uses it. To launch the graphical interface for the JDBC data source generator, select **Generate** → **JDBC Data Source...** in the Gem Cutter.

You should be presented with the dialog box shown in *Figure 9.12*:

**Figure 9.12. Generate JDBC Resultset Gem dialog box**

Enter an appropriate name for the data source and check the **Include record extractor gem** check box. This will create a gem that presents the data as a list of CAL records in addition to

the default which is a ResultSet. Click **Next** and select the connection gem created above and the corresponding SQL builder gem for your database type. Click **Next** and select a table from the list displayed. Click **Next** and choose the table fields that you want to include. Click **Next** and select fields to use to sort the records, if desired. Click **Next** and you can review the SQL select statement that will be used to select data from your database. Click **Next** again and you will see a sample of the results obtained by running this query. Click **Finish** and two gems will be created in your current module. One will be of type ResultSet and the other will be a list of records.

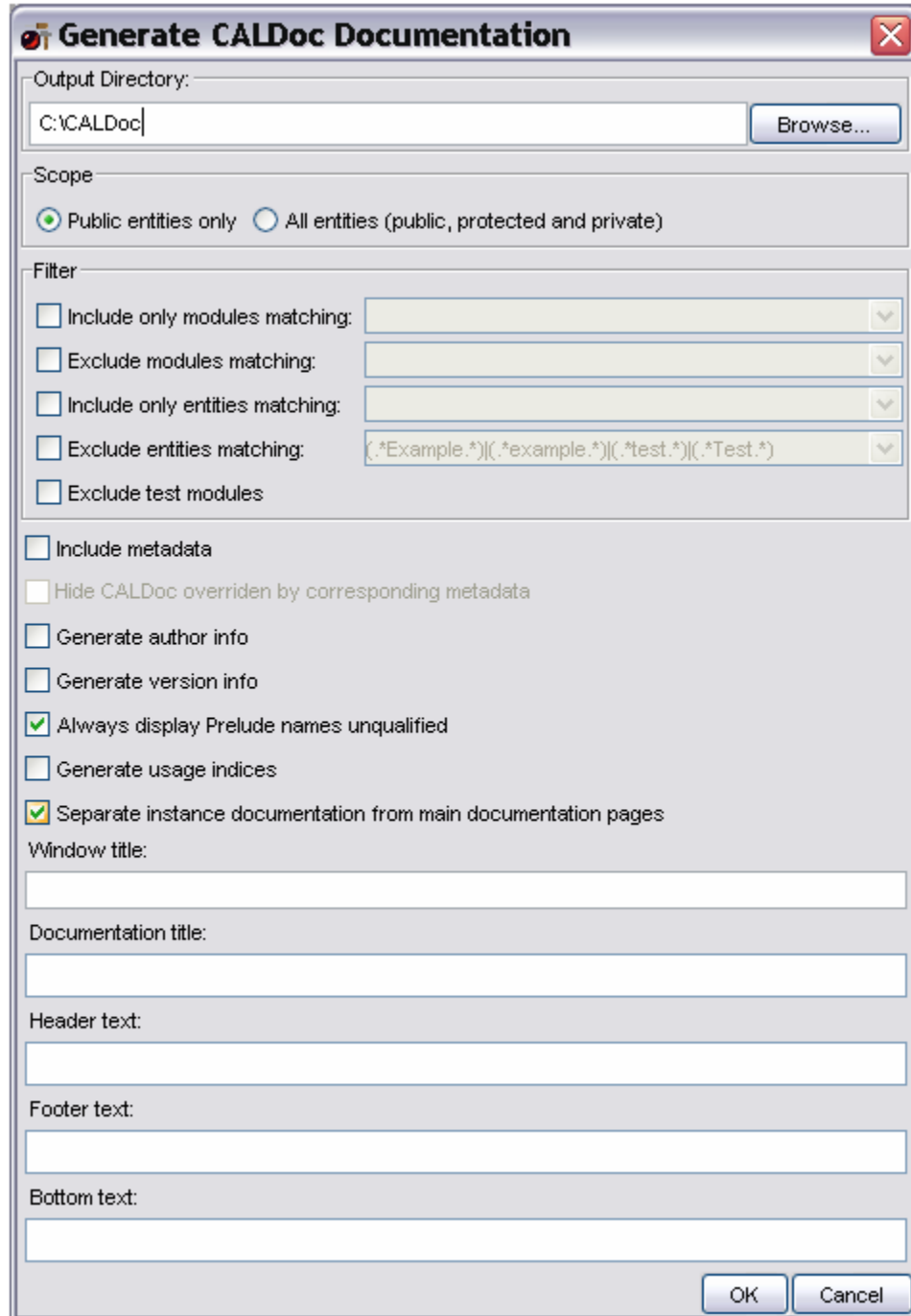
---

# Chapter 10. Using the Documentation Generator in the Gem Cutter

The Quark Platform comes with a documentation generator for producing HTML documentation from CALDoc comments and metadata for the various entities in a CAL workspace. This generator can be accessed via a graphical interface in the Gem Cutter.

To launch the graphical interface for the documentation generator, select **Generate → CALDoc Documentation...** in the Gem Cutter.

You should be presented with the dialog box in *Figure 10.1*:



The dialog box is titled "Generate CALDoc Documentation". It contains several sections for configuring the documentation generation process:

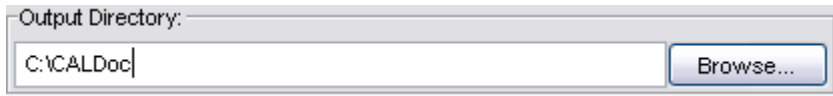
- Output Directory:** A text field with "C:\CALDoc" and a "Browse..." button.
- Scope:** Two radio buttons: "Public entities only" (selected) and "All entities (public, protected and private)".
- Filter:** A group of checkboxes and text fields:
  - ☐ Include only modules matching: [text field]
  - ☐ Exclude modules matching: [text field]
  - ☐ Include only entities matching: [text field]
  - ☐ Exclude entities matching: [text field containing "(. \*Example. \*) (. \*example. \*) (. \*test. \*) (. \*Test. \*)"]
  - ☐ Exclude test modules
- Options:** A list of checkboxes:
  - ☐ Include metadata
  - ☐ Hide CALDoc overridden by corresponding metadata
  - ☐ Generate author info
  - ☐ Generate version info
  - ☒ Always display Prelude names unqualified
  - ☐ Generate usage indices
  - ☒ Separate instance documentation from main documentation pages
- Text Fields:** Four empty text fields labeled "Window title:", "Documentation title:", "Header text:", and "Footer text:", followed by a "Bottom text:" label and an empty text field.
- Buttons:** "OK" and "Cancel" buttons at the bottom right.

**Figure 10.1. Generate CALDoc Documentation dialog box**

This dialog box lists the options that can be specified for the generation of the documentation. Once the desired options have been specified, the generation process begins immediately upon clicking the **OK** button.

# 1. CALDoc Generation Options

Output Directory


A dialog box titled "Output Directory:" with a text input field containing "C:\CALDoc" and a "Browse..." button to its right.

**Figure 10.2. Output Directory section**

Specifies the output directory where the generator saves the generated HTML files. Leaving this option blank causes the files to be saved to the current directory. The directory can be absolute, or relative to the current working directory. The output directory is automatically created if it does not already exist.

The **Browse...** button on the right launches a directory chooser for interactively selecting the output directory.

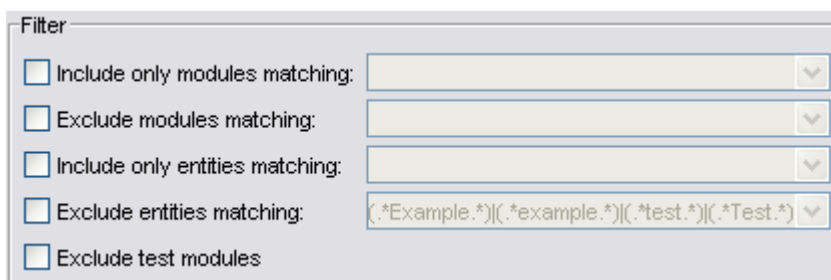
Scope

A dialog box titled "Scope" with two radio button options: "Public entities only" (which is selected) and "All entities (public, protected and private)".

**Figure 10.3. Scope section**

Specifies whether documentation should be generated for public entities only, or for all entities.

Filtering Options

A dialog box titled "Filter" with five filter options, each with a checkbox and a text field for a regular expression. The options are: "Include only modules matching:" (checkbox unchecked, field empty), "Exclude modules matching:" (checkbox unchecked, field empty), "Include only entities matching:" (checkbox unchecked, field empty), "Exclude entities matching:" (checkbox unchecked, field containing "(\*Example.\*)"(\*example.\*)"(\*test.\*)"(\*Test.\*)" ), and "Exclude test modules" (checkbox unchecked, no field).

**Figure 10.4. Filtering options section**

The five filtering options shown in *Figure 10.4* provide the ability to control whether documentation should be generated for certain modules and/or certain entities. These filters can be applied in any combination.

The first four filters are based on matching regular expressions. When the check box for one of these filters is selected, the corresponding field on the right is enabled to allow for specifying the regular expression to be used.



The regular expressions are used in a case-sensitive manner by the filters.

From top to bottom, these four filters are:

- **Include only modules matching:** (*regular expression*)
  - The documentation generator will only generate documentation for modules whose names match the specified regular expression.
  - For example, specifying the regular expression `File.*` would cause the generator to only generate documentation for modules whose names begin with the prefix `File`.
- **Exclude modules matching:** (*regular expression*)
  - The documentation generator will not generate documentation for modules whose names match the specified regular expression.
  - For example, specifying the regular expression `.*_Tests` would cause the generator to exclude from generation any module whose name ends with the suffix `_Tests`.
- **Include only entities matching:** (*regular expression*)
  - The documentation generator will exclude from the documentation any functions, data constructors, and class methods whose names do not match the specified regular expression.
- **Exclude entities matching:** (*regular expression*)
  - The documentation generator will exclude from the documentation any functions, data constructors, and class methods whose names match the specified regular expression.
  - For example, specifying the regular expression `(.*Example.*) | (.*example.*) | (.*test.*) | (.*Test.*)` would cause the generator to exclude entities whose names contain the string `Example`, `example`, `test`, or `Test`.

The fifth filtering option, **Exclude test modules**, indicates to the documentation generator that test modules are to be excluded from the generated documentation. A test module is defined to be a module whose source file has a directory named “test” as part of its path.

## Other Options

Include Metadata

This option selects whether or not to include CAL metadata in the generated documentation.

Hide CALDoc overridden by corresponding metadata	This option specifies that the CALDoc for a CAL entity should not be included in the documentation if the corresponding metadata is present. This option is only enabled if the <b>Include Metadata</b> option is selected.
Generate author info	Specifies that author information should be included in the generated documentation.
Generate version info	Specifies that version information should be included in the generated documentation.
Always display Prelude names unqualified	Specifies that the names of <code>Prelude</code> entities should always be shown in their unqualified form, e.g. displaying <code>Int</code> rather than <code>Prelude.Int</code> throughout the generated documentation.
Generate usage indices	Specifies that one usage page should be generated for each documented type and type class. The page describes which functions, data constructors, and class methods refer to the type/type class in their type signatures, as well as any associated class instances.
Separate instance documentation from main documentation pages	Specifies that documentation for instances should be generated as separate HTML pages, instead of being included with the main documentation pages.
Window title	Specifies the title to be placed in the HTML <code>&lt;title&gt;</code> tag. This appears in the window title and in any browser bookmarks (favorite places) that someone creates for this page. This title should not contain any HTML tags, as the browser will not properly interpret them.
Documentation title	Specifies the title to be placed near the top of the overview file. The title may contain HTML tags and white space.
Header text	Specifies the header text to be placed at the top of each output file. The header will be placed just below the upper navigation bar. The text may contain HTML tags and white space.
Footer text	Specifies the footer text to be placed at the bottom of each output file. The footer will be placed just above the lower navigation bar. The text may contain HTML tags and white space.
Bottom text	Specifies the fine-print text to be placed at the bottom of each output file. The text will be placed at the bottom of the page, below the lower navigation bar. The text may contain HTML tags and white space.

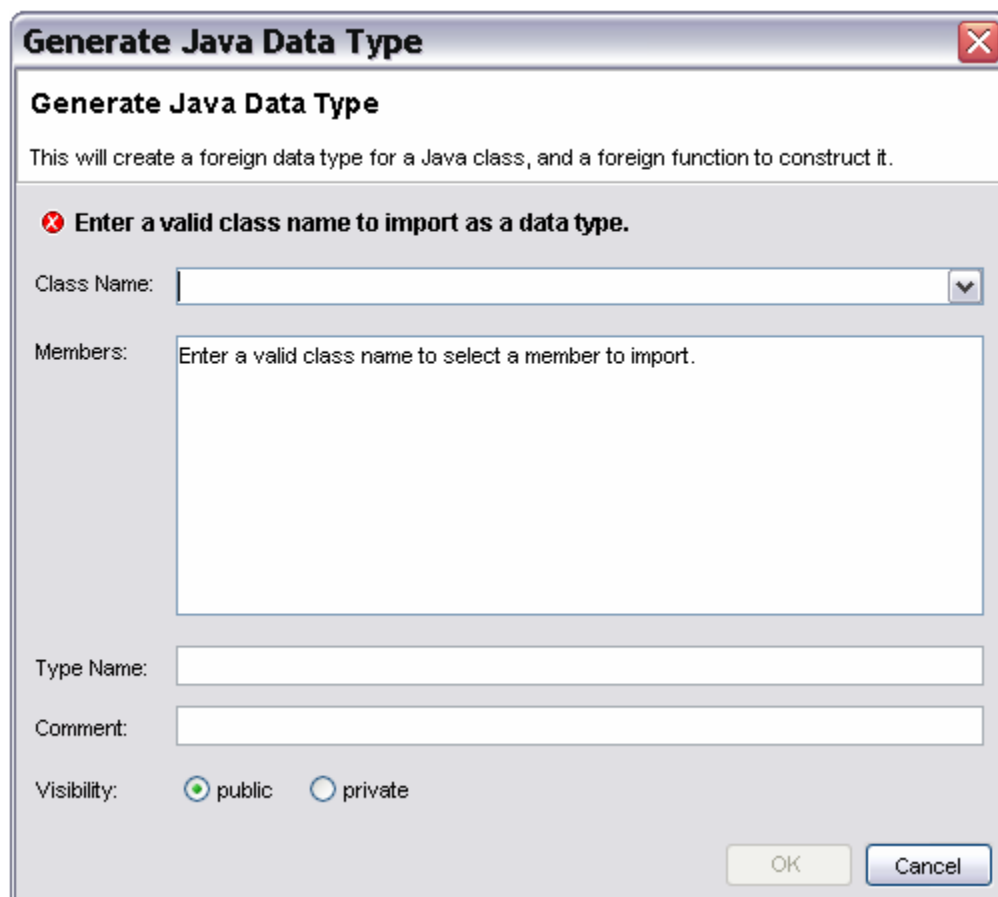
---

# Chapter 11. Generating Gems From Java Code

The Gem Cutter has the ability to create Gems by importing Java classes, methods and fields.

## 1. Creating a CAL Type From a Java Class

The Gem Cutter provides an easy way to create a data type for an existing Java class. To import a single type into the current module in the Gem Cutter, select the open the “Generate” menu and select the “Java Data Type...” option. This will display the dialog in *Figure 11.1*:



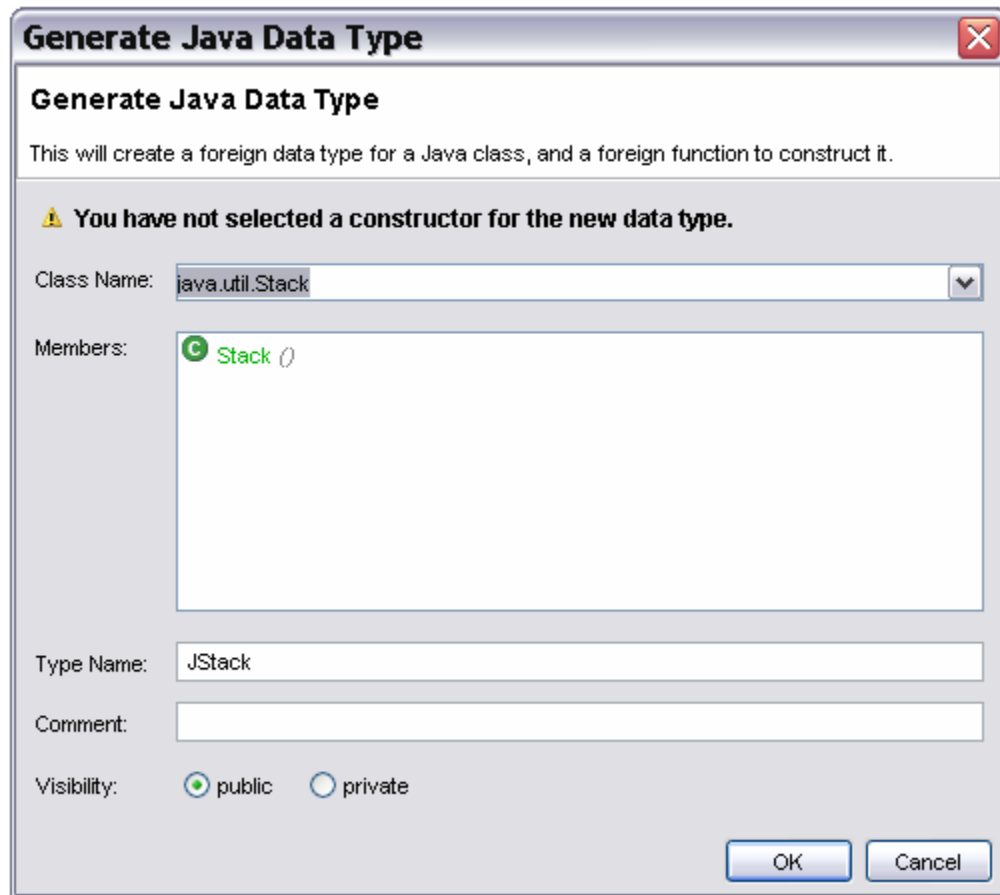
**Figure 11.1. Generate Java Data Type dialog box**

The next step is to enter the name of the class to be imported. As characters are typed, the Gem Cutter searches for Java classes with names matching the typed string. For example, suppose we want to import the `java.util.Stack` class and create a data type for it in the Gem Cutter. Typing **stack** in the **Class Name** box causes the program to display a drop-down list of the classes found, shown in *Figure 11.2*:



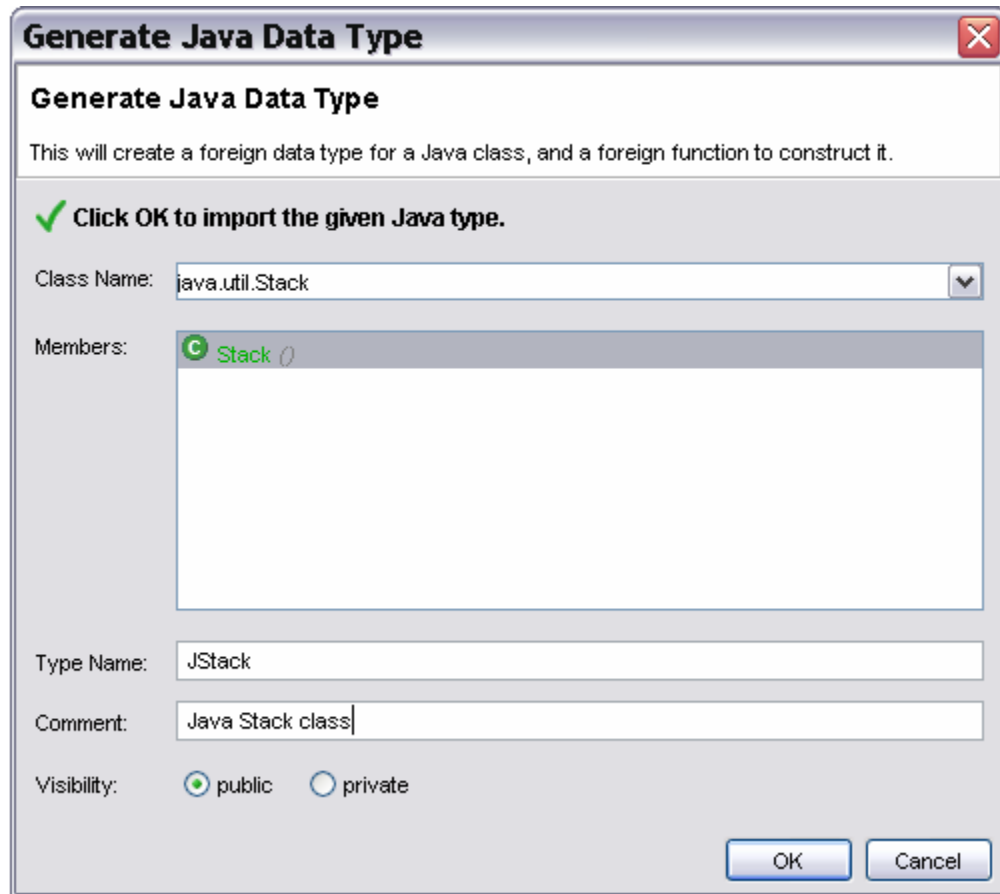
**Figure 11.2. Selecting a class to import**

Selecting the desired class from the list updates the **Members** field with the available constructors for this data type. The Gem Cutter also suggests a name for the data type in the **Type Name** field, shown in *Figure 11.3*:



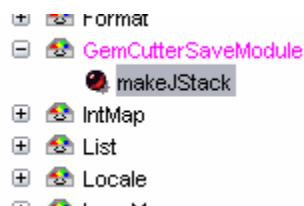
**Figure 11.3. Selecting the `java.util.Stack` class**

A constructor must be chosen to use from this class. A Gem will be generated to represent this constructor in the Gem Cutter, taking whatever input arguments the constructor requires and outputting an object of the new type. In this case, only one constructor is available, so we select it. A short comment about this data type can also be added, as shown in *Figure 11.4*:



**Figure 11.4. Selecting a constructor and adding a comment to an imported class**

Clicking “OK” will now import the data type and create a new Gem. In this case, a Gem named `makeJStack` has been created in the current module, displayed in *Figure 11.5*.



**Figure 11.5. Gem Browser with newly created `makeJStack` Gem**

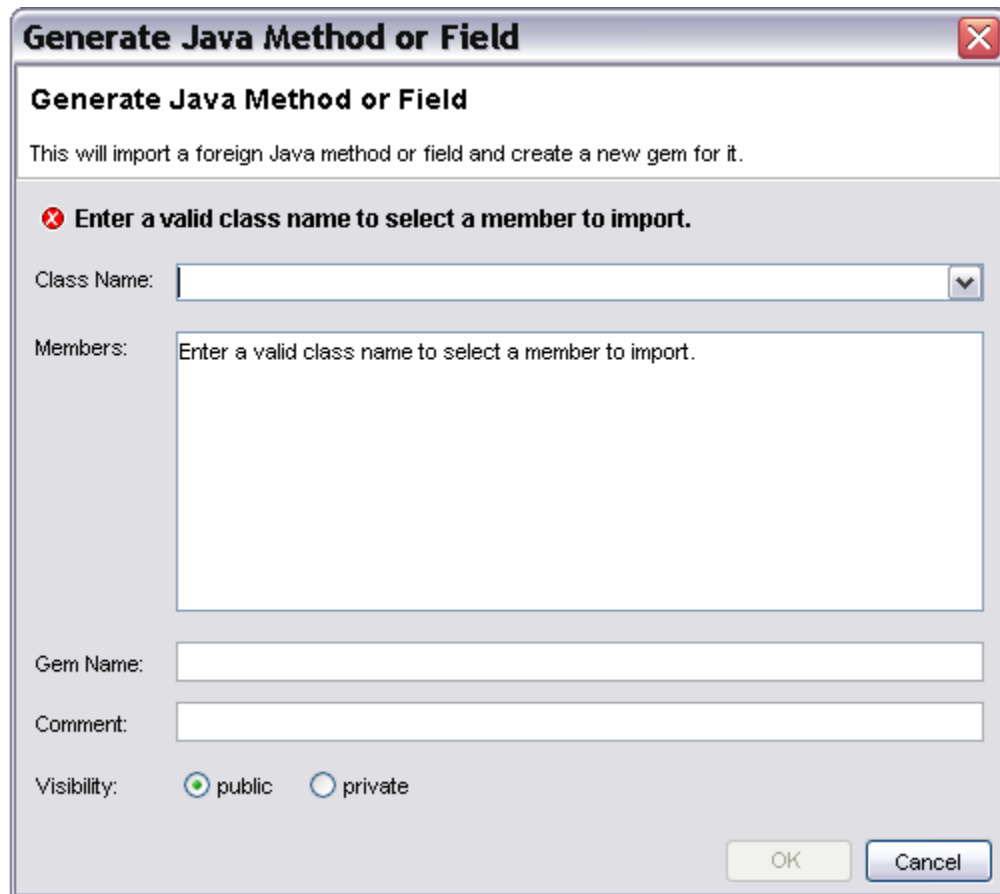
This Gem (in *Figure 11.6*) represents the default constructor for the `Stack` class, which now has the type `JStack` in the Gem Cutter.



**Figure 11.6. `makeJStack` Gem**

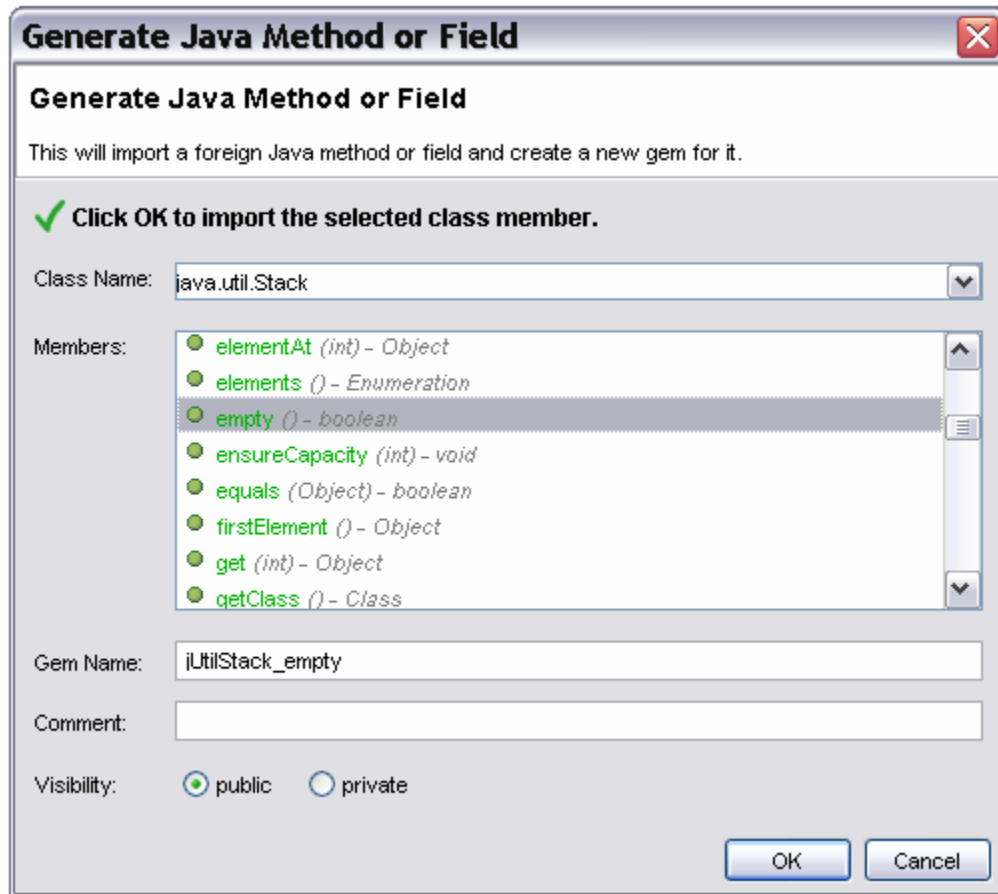
## 2. Importing Java Methods and Fields

Importing methods and fields from Java classes is similar to importing classes into the Gem Cutter as types. To continue the previous example, suppose we want to import the method `empty()` in the `java.util.Stack` class. First, select **Generate** → **Java Method or Field...** which will open the dialog box in *Figure 11.7*:



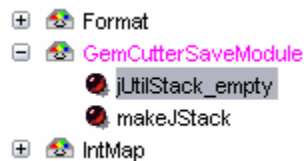
**Figure 11.7. Generate Java Method or Field dialog box**

Entering `stack` and selecting `java.util.Stack` in the **Class Name** field displays a list of the available methods and fields in the **Members** box. Scrolling down the list and clicking on the `empty()` method, we see that the Gem Cutter has created a default name for the Gem to be created, shown in *Figure 11.8*:



**Figure 11.8. Importing the `empty()` method from `java.util.Stack`**

Clicking the **OK** button will now create our Gem in the currently open module, as displayed in *Figure 11.9*.



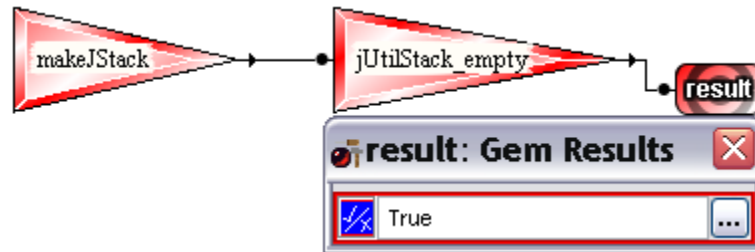
**Figure 11.9. Gem Browser with newly created `jUtilStack_empty` Gem**

The Gem looks like *Figure 11.10*:



**Figure 11.10. `jUtilStack_empty` Gem**

There are now two Gems related to the `JStack` type in the current workspace. A simple thing to try now is creating a `JStack`, and then checking that it is empty. Placing a `makeJStack` Gem and a `jUtilStack_empty` Gem on the Table Top and connecting them as shown in *Figure 11.11*, we can run the resulting function and check that the new stack is in fact empty:



**Figure 11.11.** Testing the `jUtilStack_empty` Gem

The result is that the stack is empty, as expected.

## 3. Generating a Foreign Import Module

The Gem Cutter is capable of generating a new module from a collection of existing Java classes and their members.

### 3.1. Using the Java Foreign Input Factory (JFit) in the Gem Cutter

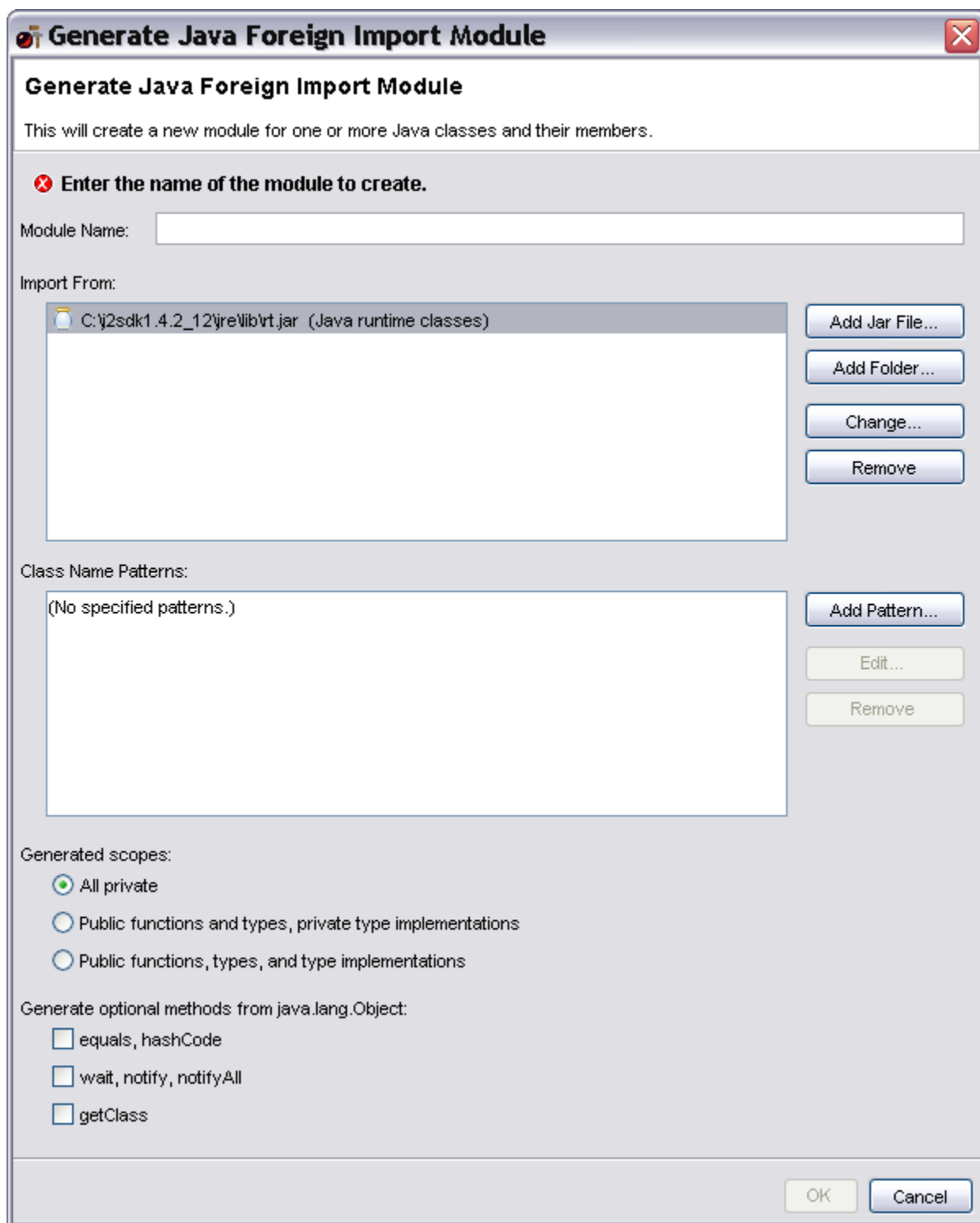
This section describes how to use the Java Foreign Import Module Factory UI to invoke JFit.

#### Invoke the Java Foreign Import Module Factory

From the Gem Cutter menu bar, select **Generate** → **Java Foreign Import Module**.

The factory dialog will be displayed, shown in *Figure 11.12*:





**Figure 11.12. Generate Java Foreign Import Module dialog box**

## Enter the Module Name

This will be the name of the generated module. For example, if “Test” is entered here, the generated module will be module `Test`, and the file will be `Test.cal`.

## Select Import Sources

In order for JFit to know the classes for which foreign imports should be generated, one or more import sources should be specified. There are two types of sources:

Jar File	The class(es) to import exist in the selected .jar file.
Folder	The class(es) to import are in .class file form, and this folder represents the parent of the root package component. For instance, if the files to import are subpackages of “com.businessobjects”, the folder here would be the parent of the “com” folder.

### Note

The classes which comprise the standard Java distribution are available in .jar form with the Java SDK. These can typically be found in the folder: `<java.home>\lib`. For instance, the `java.lang.StringBuffer` class from a recent Java distribution might be found at a path similar to: `C:\j2sdk1.4.2_07\jre\lib\rt.jar`.

## Add Import Patterns

Import patterns can optionally be provided to widen or narrow the scope of the classes of interest within the scope of the selected import sources. There are two types of import patterns:

Exclude patterns	Classes whose fully-qualified names match the given pattern will be excluded from the classes for which types and functions will be generated.
Include patterns	Classes whose fully-qualified names match the given pattern will be included in the classes for which types and functions will be generated, provided that their names do not match any exclude patterns.

In both types of patterns, wildcards may be used to match groups of classes. For instance:

<code>java.lang.StringBuffer</code>	matches the class <code>java.lang.StringBuffer</code>
<code>java.math.*</code>	matches all classes in all packages starting with <code>java.math</code> .
<code>java.lang.Illegal*Exception</code>	matches classes in the <code>java.lang</code> package starting with <code>Illegal</code> and ending with <code>Exception</code>
<code>com.businessobjects.test. classes.MyClass\$?</code>	Matches inner classes of <code>com.businessobjects.test. classes.MyClass</code> whose unqualified names are one character long

## Select Generated Scopes

This section allows the visibility of the generated types and functions to be specified. There are three options:

All private	The visibility of all generated entities is private
Public functions and types, private type implementations	The visibility of generated functions and types is public. The implementation scopes for the generated types will be private. This indicates that other modules can not “know” that the implementation of the type is as a foreign type, and thus cannot also define foreign functions which operate on the type.
Public functions, type, and type implementations	The visibility of all generated entities is public.

## Select Optional Methods to Generate from `java.lang.Object`

By default, functions are not generated for a number of methods inherited by all java classes from the class `java.lang.Object`. These methods are: `equals()`, `hashCode()`, `wait()`, `notify()`, `notifyAll()`, and `getClass()`.

This section can be used to override the default settings, and optionally generate functions for a number of these methods for all generated types.

## Generate the Module

Click “OK” to accept the user-defined inputs and generate the foreign import module.

---

# Chapter 12. Understanding Type Expressions

This section provides a basic overview of type expressions as they pertain to the CAL language in the Gem Cutter. For more information on type expressions, refer to the CAL User's Guide document.

Type expressions are fairly straightforward to decode. They describe a type in a concise textual form. For functions, Type Expressions deal with describing the 'type contract' of the function, which can be broken down into types for each input and the output. We have to be careful in our interpretation of how we assume a function will get evaluated from looking at its Type Expression.

First of all, let's consider the types of some simple values:

Type Expression	Meaning
Boolean	A boolean value
Char	A character
Double	A double precision floating point number
a	A type variable, meaning 'any type'

These type expressions are simple examples and therefore easy to understand. These types, especially the first three, should look familiar. The first example, the type variable, may be unfamiliar however. Type variables are very powerful as they allow us to describe whole groups of types where all the type variables with the same letter can be replaced with any one type expression.

The next examples of types are some built-in types with special syntax:

Type Expression	Meaning
[a]	A list of values. Every value is of type a
(a,b)	A pair of values. The first value is of type a, the second value is of type b

These will often have their type variables set to actual types as we mentioned above. For instance, a list of Boolean values will be [Boolean], and a pair of floating point values would be (Double, Double).

Types can include type class constraints. A type class is a set of types which conform to having certain functions defined over them. Gems are then defined that accept any type in a type class as input, and perform an operation which makes sense for every type that is a member of the type class. For example, the Num type class has instances Int and Double (among others) and has all the simple arithmetic functions defined. By using the Num type class we can provide a single add Gem which will accept as input any members of the Num type class, including Int and Double.

Here are some examples of type class constrained type expressions:

Type Expression	Meaning
<code>Num a =&gt; a</code>	A value which conforms to being a <code>Num</code> (number). Some examples of member types of <code>Num</code> are <code>Int</code> and <code>Double</code> , meaning that <code>a</code> can be an <code>Int</code> or a <code>Double</code> .
<code>Ord a =&gt; a</code>	A value which conforms to being an orderable type, meaning one that supports ordering functions. Some examples of member types of <code>Ord</code> are <code>Int</code> , <code>String</code> and <code>Time</code> .

User defined types (and types for which there are no special syntactic treatments) are just described in type expressions by their type names. Here are some examples:

Type Expression	Meaning
<code>Maybe a</code>	A <code>Maybe</code> type (from the standard <code>Prelude</code> ).
<code>Ordering</code>	An enumeration of ordering relationships.

Finally, functions are described by their type expressions. Here are some examples:

Type Expression	Meaning
<code>Int -&gt; Int -&gt; Int</code>	A function which takes an integer, then another integer and produces an integer
<code>[Char] -&gt; Int -&gt; Char</code>	A function which takes a list of characters, then an integer and produces a character
<code>Ord a =&gt; a -&gt; a -&gt; a</code>	A function which takes two orderable values and returns an orderable value
<code>(a -&gt; Boolean) -&gt; [a] -&gt; [a]</code>	A function which takes a function (which takes any type and returns a boolean value) and a list of any type, and which returns a list of any type. All the 'any types' must be the same when this type is used.
<code>(a -&gt; b) -&gt; [a] -&gt; [b]</code>	A function which takes another function (which takes one 'any type' and returns another 'any type') and a list of any type, and that returns a list of any type. Notice that in this case, whatever type the first argument of the provided function is must be the same type as the elements of the list in the second argument. Also, whatever type the second argument of the provided function is must be the same type as the elements of the returned list.
<code>(Ord a, Eq b) =&gt; (a,b) -&gt; [a] -&gt; b</code>	A function which takes a pair of values, the first of which is a member of the type class <code>Ord</code> , and the second is a member of the type class <code>Eq</code> . The function also takes a list of values whose type

Type Expression	Meaning
	is the same as for the first value in the pair. The function returns a value typed the same as the second value in the pair.

---

# Glossary

## Terms

Arity	The number of arguments that this function will consume in order to fully evaluate.
CAL	This is the native language used by Business Objects Gems. The language is a functional language. This means that every entity in the language is a function. Functions can be passed as arguments, and returned by other functions.
Code Gem	A green Gem that allows the user to input CAL code to implement the functionality of the Gem. See Also <i>Gem</i> , <i>CAL</i> .
Collector	A black Gem that receives a value and gives it a local name. This value can then be output to other Gems using Emitters. See Also <i>Emitter</i> , <i>Gem</i> .
Constructor	See <i>Data Constructor</i> .
Data Constructor	<p>A Data Constructor is an identifier which is used in CAL to build values of a specific type. Constructors behave like functions, usually taking some simpler typed values and creating an instance of a more abstract type (although there are many constructors taking no 'arguments'). To differentiate them from functions, constructors have capitalised names.</p> <p>For example, to create a <code>Maybe</code> type representing a <code>Double</code> value that exists, the <code>Just</code> data constructor can be used in the following manner:</p> <pre>Just 5.4</pre> <p>Data constructors are defined in data definitions in the CAL language. See Also <i>CAL</i>, <i>Function</i>.</p>
Emitter	A white Gem used to output the value represented by a local variable. A Collector must be placed on the Table Top before a corresponding Emitter can be created and used. See Also <i>Collector</i> , <i>Gem</i> .
Function	A mathematical concept which maps a set of values onto another set. In standard programming languages functions are usually considered to be routines that perform some computation and return a value. CAL, being a functional language, takes a more pure approach to functions and is able to manipulate functions as data. In fact, conceptually, everything in the CAL language is a function and has the same treatment. This makes for a simple language, yet one with very powerful features for transforming data and describing analysis.

	See Also <i>CAL</i> .
Gem	<p>The representation of a CAL function. A Gem can represent a specific slice of data (possibly with a dynamic meaning), or some business logic without association to specific data. Gems can transform, filter or abstract data in any way. Gems are combinable in a completely 'type safe' manner. This means that it is not possible to produce abstractions that have invalid or ambiguous meanings in the way that they consume or produce data.</p> <p>See Also <i>CAL</i>.</p>
Gem Browser	<p>The window allowing the user to browse for Gems in the current workspace. The Gem Browser allows Gems to be arranged in many different ways, and the user can also use it to search for Gems by name.</p> <p>See Also <i>Gem</i>.</p>
Gem Cutter	<p>The graphical design tool for Gems.</p> <p>See Also <i>Gem</i>.</p>
IntelliCut	<p>A feature of the Gem Cutter designed to help the user find Gems and connections that are appropriate to make, based on the current state of the Table Top. Possible connections will be indicated to the user with dotted lines, and a list of connectible Gems will be displayed. The IntelliCut feature can be enabled or disabled via the <b>View</b> → <b>Preferences</b> menu option.</p>
Local Function	<p>A named 'inner' function definition which is restricted in scope to the definition of a single function. Local function definitions are useful to break the definition of a complex function into smaller pieces. Local functions are defined by <code>let</code> expressions in CAL.</p> <p>See Also <i>CAL</i>.</p>
Local Variable	<p>A variable definition which is restricted in scope to the definition of a single function. Local variables are useful to break the definition of a complex function into smaller pieces. Local variables are defined by <code>let</code> expressions in CAL, whereby a value is assigned to the variable identifier. Once assigned, local variables are immutable. Local variables are the only kind of variable in CAL, which has no concept of global, shared, or mutable variables.</p> <p>See Also <i>CAL</i>.</p>
Module	<p>The organising principle for the CAL language. Similar to the concept of a package in other languages, a module is a collection of data types and functions that are designed for a particular purpose (or for which it makes sense to collect them together). The module also acts a boundary for scope and visibility, allowing items to be declared as private and thus only visible within the module itself. This greatly helps to avoid namespace pollution and in helping to establish an overt intent for the usage of items within the module. Modules can be imported into one another which permits the public members of the imported module to be visible within the importing module.</p> <p>See Also <i>CAL</i>.</p>



Operator	<p>An operator is a function which can be used inline in CAL syntax. Functions are normally applied with prefix fixity (i.e. the function name comes first, followed by its arguments). However, this is inconvenient in cases where the functions are usually written with different fixity, the most common case of which are the inline numeric operators (e.g. <code>+-/*</code>).</p> <p>See Also <i>Function</i>, <i>CAL</i>.</p>
Parametric	<p>An expression that is based on the value of a variable. Referring to type expressions, this implies that the type is generalised and can be made more specific by setting the value of the variable.</p> <p>For example:</p> <pre>data List a = Nil   Cons a (List a);</pre> <p>This means that a <code>List</code> of 'a' is either the empty list (<code>Nil</code>) or a list construction consisting of an 'a' and a <code>List</code> of 'a's</p> <p>See Also <i>Type Variable</i>.</p>
Partial Evaluation	<p>CAL is a functional language which has the property of being able to partially evaluate functions and return other functions in the case where not all the input arguments are provided (i.e. applied arguments &lt; arity).</p> <p>See Also <i>Arity</i>, <i>Function</i>, <i>CAL</i>.</p>
Polymorphism	<p>Able to deal with multiple types. All functions in CAL (and hence Gems) are implicitly polymorphic. For example, the type expression of the function <code>length</code> is:</p> <pre>length :: [a] -&gt; Int;</pre> <p>This tells us that <code>length</code> takes a list of 'anything' and returns a <code>Double</code> (double precision floating point number). This implies that <code>length</code> will work on any list type.</p> <p>See Also <i>Function</i>, <i>CAL</i>, <i>Type Expression</i>.</p>
Prelude	<p>The fundamental standard library for CAL which provides data type and function primitives for writing CAL programs. The Prelude is included and compiled before other user CAL code and is the basis on which other code is commonly built. Other CAL libraries use the standard facilities offered by Prelude.</p> <p>See Also <i>CAL</i>.</p>
Record Creation Gem	<p>A magenta triangular Gem used to create a record value.</p> <p>See Also <i>Gem</i>.</p>
Record Field Selection Gem	<p>A purple triangular Gem used to select a field from a record.</p> <p>See Also <i>Gem</i>.</p>
Type	<p>A description of the kind of values that are legal for a variable. A type represents a set of values that, taken together, make up the type. For example, the <code>Integer</code> type represents the set of all integers. We can test whether types are compatible by checking if one set is a pure subset of the other.</p>

	See Also <i>Type Variable</i> .
Type Expression	<p>An expression formalism for describing types. For example:</p> <p>[Double] A list of double precision floating point numbers</p> <p>(Season-&gt;Temp) A function taking a Season and returning a Temp</p> <p>See Also <i>Type</i>.</p>
Type Unification	See <i>Unification</i> .
Type Variable	<p>A variable used in a type expression. These are normally identified by single letter names (a,b,c...). They mean that any type can be used to substitute for their location, but when this happens, the same type must be used for all occurrences of the given variable in the single type expression.</p> <p>See Also <i>Type</i>, <i>Type Expression</i>.</p>
Unification	<p>The act of merging two or more entities into a single entity which can represent all cases of the original set.</p> <p>Unification is the process by which two types are converted into one where the new type is compatible with both the original types. Incompatible types cannot be unified and this results in a typing error.</p>
Value Gem	<p>A blue Gem that defines a constant value.</p> <p>See Also <i>Gem</i>.</p>