Using the UML Action Language in Rational Software Architect

Mattias Mohlin
Senior Software Architect
IBM

The UML Action Language (UAL) is a formal action language supported by Rational Software Architect. It is based on the upcoming OMG standard ”Action language for foundational UML (Alf)”. However, it should be noted that the Alf standard is still in the process of being finalized. Therefore, the UAL support described in this article will likely be subject to change in the future. What is described in this article is what is supported in Rational Software Architect version 8.0.3.

We will cover the following:

• What is UAL? What are the reasons for using it when designing UML models?
• Where and how to enter UAL code in a Rational Software Architect UML model.
• A walkthrough of the different language constructs available in UAL.

Before reading this article it is a good idea to have a general understanding of Rational Software Architect and UML modeling.
UML Action Language Background

A UML action is the fundamental unit in a behavior specification, such as a state machine or activity model. You can think about actions as something that “do something” - as the atomic pieces that together form the specification of what the behavior does. Examples of actions are:
- performing an operation call
- sending a signal
- creating a new instance
- assigning a value to an attribute

Some of these actions have a graphical notation, and can be used mainly in activity diagrams. For example, the graphical notation of an action that calls an operation looks like this in Rational Software Architect:

![Graphical notation of an action that calls an operation]

The small square on the border of the action symbol is a pin. Pins are used for providing input to actions (input pins), as well as for returning output results from them (output pins). The small arrow inside the pin symbol shows whether it is an input or output pin. Actions are coordinated by flow edges that connect action symbols or pins with each other.

It is possible to use UML activities with action nodes for describing many kinds of UML behaviors. However, specifying detailed action behavior using only activity diagrams have several practical problems:
- A textual representation (syntax) of actions is much more compact than a graphical notation.
- Software developers are used to programming languages that have a textual syntax, such as Java.
- It is usually more efficient to use text editors for writing action code, than to only rely on graphical editors.

These problems are part of the reason for introducing a textual action language for UML, the Action Language for Foundational UML (Alf). This OMG standard specifies a textual syntax for most of the actions that are present in the UML standard. It also explains how to map the constructs of the Alf syntax to the concepts present in the UML model, for example how local variables are represented in an activity model using concepts such as pins and flow edges.

The Alf syntax is organized in conformance levels, with increasing expressive power and complexity:
1. **Minimum Conformance**
   - The syntax at this level is a true subset of the Java syntax. Most, but not all, UML actions can be expressed with the syntax at this level.
2. **Full Conformance**
   - The syntax at this level supports all UML actions.
3. **Extended Conformance**
   The syntax at this level supports all UML actions and also textual definitions of UML elements which otherwise only have a graphical notation, such as a class or package.

The UML Action Language (UAL) is an implementation of the Alf minimum conformance level in Rational Software Architect. It allows you to specify UML action behavior textually using a familiar Java-based syntax.
Using the UAL in Rational Software Architect

Model Templates
To allow users create UAL models out of the box, RSA provides the Executable UML model templates in the New Model Wizard under the General category.

If this template is not visible, the UML Action Language capabilities may need to be enabled under Preferences → Capabilities → Modeling

UAL can be used at three different locations in a UML model:
1. OpaqueBehavior
   The body is then a list of UAL statements, separated by semicolon.
2. **OpaqueAction**
   The body is then a list of UAL statements, separated by semicolon.

3. **OpaqueExpression**
   In this case only a UAL expression can be used, and no statements.

Let's go through a simple example where we will type some UAL code for all kinds of UML elements mentioned above. We will start by defining the behavior of a UML OpaqueAction in an activity diagram:

![OpaqueAction Diagram]

The first thing we need to do is to ensure that the tool is properly set-up for using UAL as the action language. Remember that Rational Software Architect also supports several other action languages, such as Java and C++. Go to Window - Preferences - Modeling - Active Language and select UAL in the "Active Language” drop down menu.
Next, right-click on the OpaqueAction in the activity diagram and perform the command ”Show Code View”.

This will open the Code view which is a view where you can view and edit the action code for the selected element.
Note the drop down menu in the bottom left corner of the Code view. In case the selected UML element may have more than one snippet of UAL code associated with it, this drop down is used for selecting which of these to view and edit. For an OpaqueAction there is only one possible UAL code snippet, called the “body” snippet. For other elements you may see multiple entries in this drop down menu. For example, a State may have three different UAL code snippets associated with it (the “entry”, “exit” and “do activity”).

Now, type the following UAL code in the Code view:

```
foo(5); // Call an operation
```

This UAL statement performs a call of an operation called ”foo” that takes one Integer parameter. Let's add such an operation to the class that contains the activity:

Now, let's implement this operation using some UAL code. Select the operation in the Project Explorer. Note that the Code view automatically switches focus to show the UAL code associated with the selected operation (so far none):
If the UAL code snippet to view or edit is more than a few lines, you may want to instead open a Code editor for viewing and/or editing it. To do this, press the ”Show Source in Editor” button ( ) that is available in the toolbar of the Code view. You now get an editor window where you can edit the UAL code instead:

The tooltip of the Code editor shows the fully qualified name of the element that you edit UAL for. This is useful in case you have several Code editors open at the same time for similar kinds of elements. When a Code editor is used for editing UAL code, then the Code view will not show the UAL code at the same time. Instead it then provides a hyperlink which you can use for navigating to the Code editor that corresponds to the currently selected element. This is another feature that is useful in case you have several Code editors open and want to know which editor that corresponds to the element that is selected in the Project explorer or in a diagram.

Type the following UAL code as the method for the operation “foo”:

```
x += count; // Increment x with the value of the operation parameter 'count'
```

This UAL statement increments an attribute called ”x”. Let's add such an attribute to the class that owns the operation:
If we select this attribute the Code view will allow us to enter UAL code:

This is an example where we only may enter a UAL expression, instead of UAL statements. The expression we enter becomes an OpaqueExpression that specifies the default value for the attribute. Type the following UAL expression:

```
1234 + 5678 // Some integer expression
```

Now we have entered some UAL code for all three possible kinds of UML elements:
OpaqueBehaviors, OpaqueActions and OpaqueExpressions can of course appear at also other places in a UML model, but it is beyond the scope of this article to go through all the possible elements.

The rest of this article will focus on what UAL statements and expressions that can be used - both their format (syntax) and what they mean (semantics). We will also give an introduction to the UAL libraries that are available.
UAL Reference

Here we will go through the entire UML Action Language, as implemented in Rational Software Architect. To start with, let's emphasize some important points:

1. The UAL is currently based on a non-finalized OMG standard (Alf). We can expect some changes in UAL when the Alf standard has become finalized. What we describe here is what is supported in Rational Software Architect version 8.0.3. Differences compared to the standard are highlighted in the sections below.

2. The syntax of UAL is fully compliant with Java syntax. However, the UAL syntax is a subset of the Java syntax, which means that although all UAL syntax is valid Java syntax, the opposite is not true.

3. Just because UAL code looks like Java code, it doesn't mean that it is or even behaves like Java code. UAL follows the semantics of the Alf standard, rather than the semantics of the Java language. Of course, sometimes the semantics in these languages are exactly the same, but don't assume it just because the syntaxes are the same.

4. A very important part of a language is the libraries that are available. UAL libraries are completely different from Java libraries. Also, the libraries that are specified in Alf are, for the most part, not yet supported by UAL. Instead Rational Software Architect supplies other libraries that can be used. These libraries are more suited for the kinds of models that you typically build in Rational Software Architect. However, you should remember that if your model uses these libraries is is actually using extensions that are not part of the Alf standard.

5. Models created with Rational Software Architect are used for a wide variety of purposes. Therefore, the UAL libraries that are currently available often depends on what kind of model that is developed. For example, models using the RealTime profile have access to UAL libraries that allow important parts of the run-time library to be accessed from UAL. These libraries are not relevant for models that do not use the RealTime profile.

We will start by going through the UAL constructs (statements and expressions) that are available. For each construct the following is provided:

- The name of the construct in the Alf abstract syntax and a reference to where it is described in the ALF standard. This can be used in order to get more information about it.
- The name of the construct in the Java abstract syntax, as implemented by the Eclipse JDT. This information is useful if you access UAL code programmatically through the Rational Software Architect APIs.
- A note if the construct somehow deviates from what is specified in the Alf standard.
- At least one example, showing how the construct can be used.

Then we will cover the contents of the available UAL libraries.

**UAL Statements**

UAL code written for OpaqueBehaviors and OpaqueActions should be a sequence of UAL statements, separated by semicolons. A UAL statement roughly corresponds to a UML action, although in some cases the mapping is a little more indirect (for example, one UAL statement may correspond to multiple UML actions).
Some statements may be annotated in order to execute differently at run-time. The syntax for statement annotations is based on a comment (e.g. `//@isolated`). A fixed number of predefined annotations exist, but none of these are currently supported by Rational Software Architect. However, since they are plain comments from a syntax point of view, it is still of course possible to annotate UAL statements.

**Inline Statement**

**Alf abstract syntax**: InLineStatement [ALF 9.3]

**JDT abstract syntax**: N/A (treated like a regular comment)

An inline statement can be used to place code in another programming language into a UAL code snippet. An inline statement is perceived as a comment in UAL, both syntactically and from an execution point of view. That is, the meaning of executing an inline statement is not defined, and relies on UAL being translated to another programming language in which the contents of the inline statement has a meaning.

The inline statement includes the specification of the programming language in which its contents are written. This language is just specified by means of a string (the name of the programming language) and is not interpreted by UAL.

*Deviation! Inline statements are not supported by Rational Software Architect.*

```
/*@inline('C++')
   addr = & data;
*/
```

**Block**

**Alf abstract syntax**: Block [ALF 9.4]

**JDT abstract syntax**: Block

A block is a statement that groups other statements. Executing a block means to execute each contained statement sequentially. Blocks do not introduce a new scope for names. This means that although you can define a local variable within a block, it may not have the same name as any other local variable defined outside the block. Also, a local variable defined in a block can be referenced from statements that follow the block.

*Deviation! In UAL local variables defined within a block are not visible outside it. See [ALF 9.4]*

One reason for using a block is that an annotation can be attached to it. This is a means to attach common properties to a group of statements, in order to affect how to execute the statements. However, the most common use of block statements is to use it for inserting multiple statements where a single statement is expected according to the Alf grammar, for example as the else-clause of an if-statement.
Example

```java
Integer x = 5;
{ // Block start
    Integer y = 5;
    y++;
} // Block end
x = y; // x is now 11 (but gives error in RSA since y is not visible)
//@parallel
{
    // some code here
}
```

Empty Statement

**Alf abstract syntax:** EmptyStatement [ALF 9.5]

**JDT abstract syntax:** EmptyStatement

An empty statement does nothing when executing. Empty statements are mainly used at places where a statement is required according to the Alf grammar, but where you don't want to perform any particular action.

Example

```java
if (x == true)
    ; // Empty statement
else {
    // some code here
}
```

Local Name Declaration Statement

**Alf abstract syntax:** LocalNameDeclarationStatement [ALF 9.6]

**JDT abstract syntax:** VariableDeclarationStatement

A local name declaration statement defines a local variable. It is visible from its point of declaration until the end of the UAL code snippet that contains it. This is true even if it is located inside a block. See for more information.

Local names may be defined at most places in a UAL code snippet. However, the following rules must be followed:

1. It is not allowed to redefine the same name within a UAL code snippet. **Deviation! Since UAL does not make a local variable visible outside its containing block, it actually allows such a variable name to be redefined in another local name declaration outside that block.**

2. Blocks that are not guaranteed to execute at least once may not contain any local name declarations. However, certain exceptions to this rule exists, for blocks that belong to another statement, such as an if-statement. These exceptions are described in the sections for those statements below.
A local name declaration must contain an initialization expression that specifies the initial value of the local variable.

When defining a local variable using a local name declaration it is optional to specify the type of the variable. If the type is omitted it is deduced from the type of the initialization expression.

**Deviation!** UAL requires local name declarations to specify the type of the local variable explicitly.

### Example

```java
Integer x = 4;
if (obj.isValid()) {
    Boolean b; // Invalid according to Alf because the block is not guaranteed
    // to execute (however UAL allows this)
}
y = x; // Implicit declaration of an Integer variable. Not supported in UAL.
```

### Expression Statement

**Alf abstract syntax:** ExpressionStatement [ALF 9.7]

**JDT abstract syntax:** ExpressionStatement

Many expressions can be used as statements. For example, an operation call can either be located in an expression (if the called operation returns a value) or be made as a statement (if the called operation does not return a value, or the return value shall be discarded). A statement that just consists of an expression followed by a semicolon (;) is known as an expression statement.

An expression statement is executed by evaluating its expression. Any value produced during that evaluation will be discarded. However, the expression evaluation typically has side-effects which make it useful to use it as a statement without assigning the resulting value somewhere.

### Example

```java
foo(3); // Any operation return value will be discarded
new MyClass(); // Creation of object as an expression statement.
```

### If Statement

**Alf abstract syntax:** IfStatement [ALF 9.8]

**JDT abstract syntax:** IfStatement

An if-statement consists of at least one boolean expression and a block statement. If the boolean expression evaluates to true, the block is executed, otherwise not. It is also possible to add
additional boolean expressions that are evaluated if the previous one evaluated to false. After each such additional expression a block statement can be added which is executed if that particular expression evaluates to true. Finally, it is possible to have a trailing "else" block that is executed if none of the boolean expressions evaluated to true.

**Deviation!** Alf requires each statement contained in an If-statement to be a Block, while UAL also allows a single statement to be used.

Since a block statement in an if-statement is not guaranteed to execute at least once, it is not allowed to define a local variable in such a block. This is because the value of such a variable would be undefined after the if-statement, if the block did not execute. However, if the if-statement has an "else" block, and each of its contained blocks declare the same local name, then these local name declarations are allowed. In this case the type of the local variable after the if-statement is the most specific type to which the types of all the defined local variables are assignment compatible.

**Deviation!** UAL does not make local variables defined in a block visible outside the block, and hence it is always allowed in UAL to define local variables inside if-statements.

### Example

```java
if (x == 4) {
    // This code executes if x is 4
    Integer qqq = 0; // Semantic error, since this block is not guaranteed to
    // execute. However, UAL supports this.
}
if (y == 2)
    doSomething(); // UAL supports a single statement here, while Alf requires
    // a block statement (i.e. curly brackets are mandatory)
if (a == null) {
    // This code executes if a is null
    Natural x = 1;
}
else if (a.isOpen()) {
    // This code executes if a is not null, and a.isOpen() returns true
    Integer x = 2;
}
else {
    // This code executes if neither of the above expressions evaluate to true
    UnlimitedNatural x = 3;
}
w = x; // Legal according to Alf (type of w is Integer) but not supported in
       // UAL.
```

### Switch Statement

**Alf abstract syntax:** SwitchStatement [ALF 9.9]

**JDT abstract syntax:** SwitchStatement

---

Using the UML Action Language in Rational Software Architect

© Copyright IBM Corporation 2011. All rights reserved.
A switch statement evaluates an expression and compares the result with values specified for "case" blocks. If the evaluation of the expression matches one of the values specified for a "case" block, then that block is executed. It is also possible to have a "default" block which gets executed if the expression evaluation does not match any "case" block value.

A difference compared to programming languages such as C/C++ and Java is that there is no "fall-through" between "case" blocks. That is, only one "case" block will execute and it is not necessary to put a break statement at the end of a "case" block to prevent the following "case" blocks to also execute. However, there is of course no harm in having such a break statement if wanted.

The syntax of a "case" or "default" block does not include the curly brackets which normally are used for blocks. In effect it becomes optional whether or not to use curly brackets around the "case" or "default" statements.

There are no restrictions on the type of the expression used in a switch statement, nor is it required that the values are constant expressions. The evaluation of the expression will be compared for equality with each "case" block value until a matching value is found. Hence, the same values that can be compared for equality in a binary equals expression (x == y) can be used in a switch statement. When a matching value is found, the associated "case" block executes, and then the execution of the switch statement is completed.

Since a block in a switch statement is not guaranteed to execute, it is not allowed to define a local variable in such a block. This is because the value of such a variable would be undefined after the switch statement, if the block did not execute. However, if the switch statement has a "default" block, and each of its contained "case" blocks declare the same local name, then these local name declarations are allowed. In this case the type of the local variable after the switch statement is the most specific type to which the types of all the defined local variables are assignment compatible.

**Deviation!** UAL does not make local variables defined in a block visible outside the block, and hence it is always allowed in UAL to define local variables inside switch-statements.

**Example**

```java
switch (getValue()) {
    case 1: case 2: // Multiple values can be associated with each "case" block
        x--; // Not required to enclose this statement with curly brackets
    case 3:
        x++;
        break; // Legal, but redundant since there is no "fall-through" between
        // "case" blocks.
    case 4: { // Legal, but redundant, curly brackets used for this "case"
        x = 0; // block
    }
    default:
        x = 5; // Will execute if none of the other "case" blocks executes
}
```

Using the UML Action Language in Rational Software Architect

© Copyright IBM Corporation 2011. All rights reserved.
**While Statement**

**Alf abstract syntax:** WhileStatement [ALF 9.10]

**JDT abstract syntax:** WhileStatement

A while-statement consists of a boolean expression and a statement block. If the expression evaluates to true the block will be executed. This process then repeats until the expression no longer evaluates to true.

*Deviation!* *Alf requires a statement contained in a While-statement to be a Block, while UAL also allows a single statement to be used.*

Although a block statement in a while-statement is not guaranteed to execute at least once, it is in fact allowed to define a local variable in it. However, it is not allowed to refer to such a local variable outside the while-statement block.

**Example**

```java
while (!stack.isEmpty()) {
    Natural b = stack.pop(); // Will execute as long as 'stack' is not empty
    log(b); // Legal to access the local variable 'b' here...
}

Boolean x = b; // ...but not here.
```

**Do Statement**

**Alf abstract syntax:** DoStatement [ALF 9.11]

**JDT abstract syntax:** DoStatement

A do-statement consists of a boolean expression and a statement block. The block will first be executed once, and then the expression will be evaluated. If it evaluates to true the block will execute again, and this process then repeats until the expression no longer evaluates to true. A do-statement is hence similar to a while-statement, but the difference is that its statement block always executes at least once. Therefore it is allowed to define local variables within a do-statement, and these variables are also visible after the do-statement.

*Deviation!* *Alf requires a statement contained in a Do-statement to be a Block, while UAL also allows a single statement to be used.*

*Deviation!* *UAL supports local variables within a do-statement block, but they are not visible outside the do-statement.*

**Example**

```java
do {
    Boolean empty = handleQueue(q);
} while (!q.isEmpty());
```
A for-statement contains a statement block, whose execution is usually controlled by a loop variable. However, there exist several variants of for-statements that control the execution of the block in different ways:

1. **No loop variable**
   In this case the for-statement will execute the block for ever, and the iteration has to be stopped by some other means, for example by using a break-statement within the for-statement block.

2. **Loop variable with iteration condition and post-iteration statement**
   In this case the for-statement will execute the block as long as the iteration condition is true. Typically the iteration condition does some comparison of the loop variable, but this is not required and any boolean expression can be used as iteration condition. After each execution of the block the post-iteration statement is executed. Typically this statement modifies the loop variable in some way, but this is not required and any statement can be used as post-iteration statement.
   Note that both the iteration condition and the post-iteration statement are optional and can be omitted.

3. **Loop variable assigned from a sequence of values**
   In this case the loop variable iterates over the values of a sequence. The first time the block executes the loop variable has the first value of the sequence, the second time the second value and so on until there are no more values in the sequence. Then no more executions of the block will take place.

**Deviation!** The first two variants of a for-statement are not part of the Alf standard, but are supported as non-standard extensions by UAL.

**Deviation!** Alf requires a statement contained in an If-statement to be a Block, while UAL also allows a single statement to be used.

Although the block statement of a for-statement is not guaranteed to execute at least once, it is in fact allowed to define a local variable in it. However, it is not allowed to refer to such a local variable outside the for-statement block.

If a loop variable is defined in the for-statement it is only visible within the for-statement, and cannot be accessed after it.
Example

```java
for (Integer i = 0; i < 10; i++) { // For-statement with loop variable 'i'
    // Code here executes 10 times
    String s = makeString(i);
    WriteLine(s); // Legal to access 's' here...
}

String t = s; // ... but not here!
Integer x = i; // Also illegal since the loop variable is not visible here!

for (;;) { // For-statement without loop variable
    if (shouldStop())
        break;
}

Set<Integer> ints = {1,2,3};
for (Integer q : ints) // Loop variable assigned from sequence of values.
    x = q;               // UAL supports a single statement here while Alf
                         // requires a block statement.
```

Break Statement

**Alf abstract syntax:** BreakStatement [ALF 9.13]

**JDT abstract syntax:** BreakStatement

A break-statement immediately completes the execution of the enclosing statement, which should be either a switch, do, while or for-statement. In case of several nested iteration statements, the break-statement only completes the nearest one. It is not possible to break out from multiple iteration statements with one single break-statement.

Example

```java
while (true) {
    Set<Integer> ints = getColl()
    for (Integer q : ints) {
        if (q > max)
            break; // Completes the enclosing for-statement, but the outermost
                     // while-statement continues to execute.
    }
}
```

Return Statement

**Alf abstract syntax:** ReturnStatement [ALF 9.14]

**JDT abstract syntax:** ReturnStatement

A return-statement can only be used in a UAL code snippet for an OpaqueBehavior that provides an implementation for an Operation. The return-statement may specify a value in the form of an expression that is evaluated when the statement executes. In this case it is required that the operation has a return parameter, and that the type of the expression is assignment compatible with the type of that return parameter.
**Deviation!** *Alf does not support return-statements that do not return a value, but this is supported by UAL as a means to immediately complete the execution of the UAL code snippet also for the case when the operation has no return parameter.*

```java
Example
if (isDone)
    return true; // Returning a boolean value
```

**UAL Expressions**

UAL code written for OpaqueExpressions should be a UAL expression. A UAL expression is at run-time evaluated to a value or a collection of values.

Each UAL expression has a type that restricts what kind of values it can evaluate to. For example, an expression that has the type Boolean can only evaluate to either true or false, while an expression of type Natural can evaluate to any integer greater than or equal to zero. If the type of an expression is a classifier, such as a UML class or interface, then the expression evaluates to an object (or a collection of objects) at run-time. In this case we make a distinction between the static and the dynamic type of the expression. The static type is the type that can be computed statically, just by analyzing the expression and its context. The dynamic type is the type of the actual object(s) that the expression evaluates to at run-time. The static type can be the same as the dynamic type, but they can also be different. However, it is always the case that the dynamic type is assignment compatible with the static type.

A UAL expression also has a multiplicity. Most expressions have multiplicity 1, meaning that the result of their evaluation will always be exactly one value. However, some expressions have higher multiplicity, which means they can evaluate to a collection of values. The multiplicity then determines the maximum number of values that the collection may contain.

**Qualified Name**

**Alf abstract syntax:** QualifiedName [ALF 8.2]

**JDT abstract syntax:** QualifiedName

A qualified name refers to a UML named element, such as a class or attribute. The syntax is a series of names separated by a dot ('.'). The names that precede the last of these names are called the qualifier. Each name in a qualifier must refer to UML namespaces, such as packages or classes. For example, assume we have the following UML model:
For this model we may use the qualified name `P1.C1.x` as a means to refer to attribute 'x' that is defined in class 'C1' that is defined in package 'P1'.

It is not always necessary to use qualifiers when referring to named elements. Often it is enough to just use the name without qualifier. Whether a qualifier is used or not does not have any impact on how a name is evaluated at run-time. A qualifier is only a syntactic construct that is used:

1. When a non-qualified name would be ambiguous. For example, if there are many named elements with the same name visible at a certain location in the model, you need to use a qualifier to be clear about which of them you refer to.
2. When you want to emphasize the location of the named element you refer to. Even if it would be enough to use a non-qualified name there are occasions when the UAL code becomes more readable by using a qualified name, as it clearly shows the origin of the referenced element.
3. When you are referring to static named elements, such as a static operation or attribute.

The type and multiplicity of a qualified name is the same as the type and multiplicity of the named element to which it refers. A qualified name is evaluated just like a non-qualified name. See for more information.

Example

```
Integer localX = P1.C1.x; // Local variable initialized by means of a qualified name.
```

**Literal Expression**

**Alf abstract syntax:** LiteralExpression [ALF 8.3.2]

**JDT abstract syntax:** BooleanLiteral, NumberLiteral, StringLiteral

A literal expression is a constant value of one of the UML predefined types. It always evaluates to a single value (i.e. it has multiplicity 1). The type of a literal expression is derived from its syntax according to the following rules:

- The literals `true` and `false` have the predefined Boolean type.
- A positive number or zero has the predefined UnlimitedNatural type.
- A negative number has the predefined Integer type.
- A string enclosing in double quotes has the predefined String type.
Name Expression

**Alf abstract syntax:** NameExpression [ALF 8.3.3]

**JDT abstract syntax:** SimpleName

A name expression is a simple name, without a qualifier. It refers either to a UML named element, such as an operation parameter, or it refers to a local variable defined within the same UAL code snippet.

The evaluation of a name expression is the value that at run-time is held by the referenced named element or local variable. The multiplicity is determined by the named element or local variable that it refers to.

This Expression

**Alf abstract syntax:** ThisExpression [ALF 8.3.4]

**JDT abstract syntax:** ThisExpression

A this-expression evaluates to a reference to the object in whose context the UAL snippet that contains the expression executes. The static type of a this-expression is the context classifier and is determined like this:

1. If the expression is located in a method (for example in an OpaqueBehavior) of a non-static operation, then the static type is the classifier that owns the method.
2. If the expression is located in a classifier behavior (for example in a StateMachine), then the static type is the classifier that owns the classifier behavior.
3. If the expression is located in the default value of a non-static property, then the static type is the classifier that owns the property.
4. If the expression is located in another behavior (not a classifier behavior) that in turn is owned by a classifier behavior, then the static type is the classifier behavior. For example, in a StateMachine that is the classifier behavior of a class C we may have a state with an entry OpaqueBehavior. A this-expression in this OpaqueBehavior has C as its static type.
Note that in other locations it is illegal to use a this-expression. For example, in a method of a static operation usage of 'this' is illegal and will cause a semantic error.

At run-time a this-expression is evaluated to a context object according to these rules:
1. For a method of a non-static operation (except a constructor), the context object is the object on which the operation was invoked. A static operation is not invoked on an object which is the reason why it is illegal to use a this-expression in its method.
2. For a method of a constructor, or the default value of a property, the context object is the object that is about to become initialized by the constructor or default value.
3. For a classifier behavior, the context object is the instance of the classifier that owns the classifier behavior, and for which this behavior executes.
4. For a another behavior (not a classifier behavior) that in turn is owned by a classifier behavior, the context object is the instance of the classifier that owns that classifier behavior.

The multiplicity of a this-expression is 1, i.e. it always evaluates to a single object.

Example

```
{{Class1}}
```

Initializing the attribute 'self' to the 'Class1' instance

**Parenthesized Expression**

**Alf abstract syntax:** ParenthesizedExpression [ALF 8.3.5]

**JDT abstract syntax:** ParenthesizedExpression

A parenthesized expression is simply another expression within parentheses. The main reason for using parentheses around an expression is to control the order of its evaluation with regards to other expressions around it. For example, the expression $5 \times 6 + 7$ evaluates to 37, because the multiplicity operator (*) has a higher precedence than the addition operator (+). By using parentheses we can change the expression to perform the addition before the multiplication: $5 \times (6 + 7)$. Now the expression evaluates to 65.

Another reason for using parentheses around expressions could be to make the code more readable, as it becomes more obvious which part of an expression that will be evaluated first. Complex expressions that do not use parentheses can be hard to understand since the reader must remember the precedence for each operator that it contains.

The type and multiplicity of a parenthesized expression is the same as for its contained expression.

Example

```
((MyClass) c).foo(); // Parentheses required since member access (.)
```

Using the UML Action Language in Rational Software Architect
© Copyright IBM Corporation 2011. All rights reserved.
// has higher precedence than the cast operator.

if ( (account != null) && account.exists()) { // Redundant parentheses since
    // operator && has higher
    // precedence than operator !=
}

**Property Access Expression**

**Alf abstract syntax:** PropertyAccessExpression [ALF 8.3.6]

**JDT abstract syntax:** FieldAccess

A property access expression is used for reading or writing the value of an object property. It consists of an expression and a name. The type of the expression must be a class that has a property with the specified name. The referenced property must be visible in the context of the class. For example, it may not be private if the property access expression does not occur within the same class where the property is defined.

At run-time a property access expression is evaluated by first evaluating its expression to an object. The evaluation of the property access expression is then the value that is held by the referenced property in that object.

The type and multiplicity of a property access expression is the type and multiplicity of the referenced property.

If the expression of a property access expression has a multiplicity greater than 1, then the property access expression performs an implicit iteration over all objects to which its expression evaluates. For each such object the value held by the referenced property is read or written. In case it is read, the result of the property access expression is a collection with the extracted values.

**Deviation!** *UAL does not support a property access expression that operates on a collection of objects. That is, it is required that the expression of a UAL property access expression has multiplicity 1.*

**Example**

Car car = new Car("white");
car.color = "red"; // Writing the value of property "color" on object "car".

Integer x = car.license; // Reading the value of property "license" on object
    // "car".

this.owner = owner; // If a property has the same name as a parameter, a
// property access expression with a "this" expression
// can be used to disambiguate the names.

Set<Car> cars = {new Car("red"), new Car("blue")};
List<String> colors = cars.color; // Valid in Alf but not supported by UAL.
String c1 = colors[0]; // "red"
String c2 = colors[1]; // "blue"

Invocation Expression

Alf abstract syntax: InvocationExpression [ALF 8.3.7]
JDT abstract syntax: MethodInvocation, ConstructorInvocation, SuperConstructorInvocation

An invocation expression is used for invoking operations and sending signals. It consists of an optional target expression and a list of actual arguments. If present, the target expression evaluates to the object on which the invocation takes place. Otherwise an implicit this-expression is used as the target expression, unless a static operation is invoked in which case there is no target object to use. The list of actual arguments contains expressions that are evaluated to values that are passed in the invocation. The syntax is a comma-separated list of expressions. The first expression corresponds to the first formal parameter of the invoked operation or sent signal, the second expression corresponds to the second formal parameter and so on. If a formal parameter is specified as optional (i.e. it has a multiplicity that includes 0), then it is allowed to omit the corresponding actual argument.

**Deviation!** UAL does not support the invocation of operations or sending of signals with optional parameters. All formal parameters will be treated as mandatory even if they are specified to be optional.

Note that the Alf standard does not specify the order in which the actual argument expressions are evaluated. This is because they can be evaluated concurrently. Therefore it really should be avoided to use actual argument expressions with side effects, to avoid a dependency on a certain order of evaluating these expressions.

**Example**

```
C x = new C();
x.op1(2, true); // Invoking operation C.op1(Integer, Boolean) on object 'x'
x.op1(getInt(), getBool()); // The order in which the operations getInt() and getBool() executes is undefined so they should not have depending side effects.
x.sig(""); // Sending signal sig(String) on object 'c'
C.sf(); // Invoking static operation C.sf(). No target object specified!
```

Using the UML Action Language in Rational Software Architect
© Copyright IBM Corporation 2011. All rights reserved.
If a formal parameter has direction out or inout, then it is required that the corresponding actual argument expression can be assigned to. That is, the expression passed for such a parameter must be legal as the left hand side of an assignment.

**Deviation!** UAL does not properly support out or inout parameters. It allows any expression to be passed as the argument for such a parameter.

In case an operation with a return parameter is invoked, the type and multiplicity of the invocation expression is the same as for that return parameter. Otherwise the invocation expression has no type or multiplicity, and may not appear as the right hand side in an assignment.

A class may have many operations with the same names, but with different signatures. The signature of an operation consists of its name and the types of its formal parameters (except the return parameter, if any). Two operations with the same names but with different signatures are said to be overloaded, and which operation that will be invoked by an invocation expression is determined by the static types of the actual argument expressions.

```
Example

D

foo(p1: UnlimitedNatural)
foo(p1: Integer)

D.foo(4); // Invoking operation D.foo(UnlimitedNatural)
D.foo(-4); // Invoking operation D.foo(Integer)
D.foo((Integer) 4); // Invoking operation D.foo(Integer)
```

A special case of invocation expression is one that uses the keyword `super` as the target expression. This can only be used when invoking operations, not when sending signals. Also, it is required that the class that is the context class of the invocation expression has a super class. The type of the `super` target expression is that super class. Hence this construct can be used in order to invoke an operation defined in the super class. This syntax is only needed when omitting the target expression would lead to an unintended recursive call of the redefined context operation, instead of invoking the base operation in the super class.

Other special cases are the following syntax variations:

- `this(<arguments>)`
- `super(<arguments>)`

These may only appear in expression statements belonging to a constructor operation implementation. The meaning of the former is to invoke a constructor operation in the context class. The meaning of the latter is to invoke a constructor operation in the super class of the context class. If used, it is required that there is exactly one single such super class.
When an invocation expression refers to a signal reception, the signal that is referenced by the signal reception will be sent to the target object. Note that it is the name of the signal reception that is used in the invocation expression, but it is normally recommended to use the same name for the signal reception as for the corresponding signal. Since a signal may not have a return parameter, an invocation expression that refers to a signal reception never has a type or multiplicity, and may not appear as the right hand side in an assignment.

If an invocation expression refers to a destructor operation (i.e. an operation stereotyped by `<destroy>`) then the destructor is invoked, and after that the target object will be destroyed.

If the target expression of an invocation expression has a multiplicity greater than 1, then the invocation expression performs an implicit iteration over all objects to which its target expression evaluates. For each such object the specified operation will be invoked, or signal sent. In case an operation with a return parameter is invoked, the result of the invocation expression is a collection with the values returned by the operation invocations.

**Deviation!** UAL does not support an invocation expression that operates on a collection of objects. That is, it is required that the target expression of a UAL invocation expression has multiplicity 1.

**Instance Creation Expression**

**Alf abstract syntax:** InstanceCreationExpression [ALF 8.3.12]

**JDT abstract syntax:** ClassInstanceCreation

An instance creation expression creates a new instance (i.e. object) of a class or a non-primitive datatype. It consists of the keyword `new` followed by a reference to the class or data type to instantiate. It may also have a list of actual arguments that will be provided as actual arguments for the constructor that is invoked on the class or data type after the instantiation. The rules for deciding which constructor to invoke are the same as for which operation that gets invoked by an invocation expression. This means that constructors may be overloaded just like any other operation, and that the types of the actual arguments in the instance creation expression may influence which constructor that gets invoked.

The type of an instance creation expression is the referenced class or data type that is instantiated. Its multiplicity is always 1, i.e. an instance creation expression always creates exactly one instance. The run-time evaluation of an instance creation expression is the created instance.

If the instance creation expression has no actual arguments, then the referenced class or data type either must have a constructor without any parameters, or no constructor at all. In the latter case a default implicit constructor without parameters will be used.

If the constructor that is used by the instance creation expression is private, then it is required that the instance creation expression is located in the context of the instantiated class. If the constructor is protected, then the expression must either be located in the instantiated class or in
one of its subclasses. For a public constructor there are no restrictions on the location of the instance creation expression.

**Example**

```java
D obj = new D(4); // Constructor D(UnlimitedNatural) will be invoked
D obj2 = new D(-4); // Constructor D(Integer) will be invoked
new D(); // Illegal since there is no parameterless constructor in D
new D("x"); // Illegal unless this expression occurs in the context of D,
            // since the constructor D(String) is private.
```

**Sequence Construction Expression**

**Alf abstract syntax:** SequenceConstructionExpression [ALF 8.3.15]

**JDT abstract syntax:** ArrayInitializer

A sequence construction expression constructs a sequence of values. It may only appear as the initializer of a local UAL variable, or a UML property. It is a syntax error to use it as a general expression at other locations.

The type of a sequence construction expression is Collection<T> where the element type T is deduced from the types of the sequence elements. It is deduced as the most specific type to which all sequence elements are assignment compatible. At run-time a sequence construction expression evaluates to a collection object. It's multiplicity is therefore 1 (since there is one collection object), although the collection of course can contain any number of elements. However, since any collection object can be implicitly converted to a sequence of its elements, a sequence construction expression can also be used as the initializer of a UML property or UAL variable with multiplicity > 1.

**Deviation!** **UAL does not support the Alf collection types, such as Set<T> or List<T>**.

**Example**

```java
Integer a[] = {1,2,3}; // Initializing the collection variable 'a' to
                      // a collection of 3 integers. 'a' has multiplicity 3.

Set<Integer> b = {1,2,3}; // Initialization of variable 'b' typed by a
                          // collection type. Not supported by UAL.

a = {4,5,6}; // Error! A sequence construction expression can only be used
             // as an initializer expression.
```

Using the UML Action Language in Rational Software Architect

© Copyright IBM Corporation 2011. All rights reserved.
Sequence Access Expression

**Alf abstract syntax:** SequenceAccessExpression [ALF 8.3.16]

**JDT abstract syntax:** ArrayAccess

A sequence access expression is used for accessing one particular element in a sequence or collection object. It consists of an expression and an index. The expression must evaluate to either a sequence or a collection object. The index specifies which of the elements in the sequence or collection that shall be accessed. Indexes are zero-based, i.e. the first element has index 0, the second has index 1 and so on.

*Deviation!* The Alf standard uses indexes that start from 1.

A sequence access expression can be used both for reading from a sequence or collection, and for writing to it (i.e. to replace one of its elements with another).

If a sequence access expression accesses an element from a collection, then it is required that the collection is ordered.

```
Example

Integer a[] = {1,2,3};
List<Integer> lst = getList();

Integer b = a[0]; // 'b' becomes 1
a[2] = 9; // 'a' is now {1,2,9}
Integer c = lst[0]; // Legal, but not supported by UAL.

Set<Integer> set = {1,5,9};
Integer x = set[0]; // Error! A set is an unordered collection!
```

Increment and Decrement Expression

**Alf abstract syntax:** IncrementOrDecrementExpression [ALF 8.4]

**JDT abstract syntax:** PrefixExpression or PostfixExpression with operator '++' or '--'

An increment or decrement expression increments or decrements an integer value by one. There are two syntactic forms of both these expressions:

- Prefix form (++x or --x)
- Postfix form (x++ or x--)

Both forms have the same effect, but they evaluate to different values at run-time. The evaluation of the prefix form is the evaluation of the operand expression before it is incremented or decremented. The evaluation of the postfix form is the evaluation of the operand expression after it has been incremented or decremented.

The operand expression must be of a type that is assignment compatible with Integer. The type of an increment or decrement expression is Integer.

```
Example

Using the UML Action Language in Rational Software Architect
© Copyright IBM Corporation 2011. All rights reserved.
```
Natural n = 5;
Integer i = n++; // 'n' becomes '6' and then 'i' becomes 6
Integer j = ++n; // 'n' becomes '7' and then 'j' becomes 6

Unary Expression

**Alf abstract syntax:** UnaryExpression [ALF 8.5]
**JDT abstract syntax:** PrefixExpression with operator '!', '~', '+' or '-'. CastExpression.

A unary expression consists of a single operand expression and an operator.

The table below lists the available unary expressions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation (!)</td>
<td>Boolean</td>
<td>The evaluation of this expression is the logical negation of the evaluation of the operand expression, which must be a boolean expression.</td>
</tr>
<tr>
<td>Bitwise negation (~)</td>
<td>Bitstring</td>
<td>This expression expects an operand expression of Bitstring type (or an Integer which can be converted to a Bitstring). Each bit in the bit string will be negated.</td>
</tr>
<tr>
<td>Plus (+)</td>
<td>Integer</td>
<td>The evaluation of this expression is the same as the evaluation of its operand expression, which should be a numeric type (such as UnlimitedNatural or Integer).</td>
</tr>
<tr>
<td>Minus (-)</td>
<td>Integer</td>
<td>The evaluation of this expression is the numeric negation of the operand expression, which which should be a numeric type (such as UnlimitedNatural or Integer).</td>
</tr>
</tbody>
</table>

In addition there is the cast operator, which also is a unary expression although it does not specify an operator, but a type name instead. It is used for changing the static type of an expression and for filtering values that do not conform to a specified type from a sequence.

**Deviation!** UAL does not support use of the cast operator for filtering values from a collection.

If the dynamic type of the cast operand expression is not compatible with the type specified by the cast expression, then the run-time evaluation of the cast expression is null.

**Deviation!** UAL does not support evaluating "filing" asts to null. Instead a run-time error will appear, either when doing the cast or when using the resulting value. The usage of the cast expression is governed by a preference under the Modeling → UML Action Language preference page. The default value of this preference is Error which means that an error is reported when the cast expression is used in UAL code. The value of this preference can be set at Warning to treat the usage of cast expression as a warning or to Allow to allow the usage of the cast expression.

Using the UML Action Language in Rational Software Architect

© Copyright IBM Corporation 2011. All rights reserved.
Example
Boolean isOpen = !door.isClosed(); // Boolean negation
Integer i = ~255; // Bitstring negation
Natural n = 4;
Integer j = +(n); // Converts from Natural to Integer without using cast
Integer k = -(n); // k becomes -4

YourClass obj = getClass();
MyClass c = (MyClass) obj; // Cast to change static type of 'obj'.
if (c == null)
  // 'obj' was not a MyClass
  MyClass c = (MyClass) obj; // Cast to change static type of 'obj'.

Collection<Integer> coll = {1, -5, 6};
Collection<Natural> conv = (Natural) coll; // 'coll' becomes {1, 6}. Not
  // supported by UAL.

Binary Expression

**Alf abstract syntax:** BinaryExpression [ALF 8.6]

**JDT abstract syntax:** InfixExpression with operator "*, '/', '%', '+', '-', '<', '>', '>=', '==', '!=', '&', '|', '^', '&&' or '||'.

A binary expression consists of a two operand expressions and an operator.

The table below lists the available binary expressions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication (*)</td>
<td>Integer</td>
<td>The evaluation of this expression is the result of multiplying the evaluations of the operand expressions with each other. The operand expressions must be type compatible with Integer.</td>
</tr>
<tr>
<td>Division (/)</td>
<td>Integer</td>
<td>The evaluation of this expression is the result of dividing the evaluations of the operand expressions with each other. The operand expressions must be type compatible with Integer.</td>
</tr>
<tr>
<td>Remainder (%)</td>
<td>Integer</td>
<td>The evaluation of this expression is the result of dividing the evaluations of the operand expressions with each other and the obtaining the remainder of the integer division. The operand expressions must be type compatible with Integer.</td>
</tr>
<tr>
<td>Addition or StringConcatenation (+)</td>
<td>Integer or String</td>
<td>When the operand expressions have a type compatible with Integer, then the evaluation of this expression is the result of adding the evaluations of the operand expressions with each other. If the operand expressions have String type, then the evaluation of this expression is the operand string concatenated together into a new string.</td>
</tr>
<tr>
<td>Subtraction (-)</td>
<td>Integer</td>
<td>The evaluation of this expression is the result of subtracting the evaluation of the first operand expression with the evaluation of the second</td>
</tr>
<tr>
<td>Operator</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>Left shift (&lt;&lt;)</td>
<td>Bitstring</td>
<td>The first operand expression should be a Bitstring (or an Integer which can be converted to a Bitstring). The second operand expression should be a Natural, which specifies how many steps to switch the bits of the first operand to the left. Right bits will be filled with 0's.</td>
</tr>
<tr>
<td>Signed right shift (&gt;&gt;&gt;)</td>
<td>Bitstring</td>
<td>The first operand expression should be a Bitstring (or an Integer which can be converted to a Bitstring). The second operand expression should be a Natural, which specifies how many steps to switch the bits of the first operand to the right. Left bits will be preserved with their previous values.</td>
</tr>
<tr>
<td>Unsigned right shift (&gt;&gt;&gt;&gt;)</td>
<td>Bitstring</td>
<td>The first operand expression should be a Bitstring (or an Integer which can be converted to a Bitstring). The second operand expression should be a Natural, which specifies how many steps to switch the bits of the first operand to the right. Left bits will be filled with 0's.</td>
</tr>
<tr>
<td>Less than (&lt;)</td>
<td>Boolean</td>
<td>Evaluates to true if the left operand expression is less than the right operand expression. Both operand expressions should be type compatible with Integer.</td>
</tr>
<tr>
<td>Greater than (&gt;)</td>
<td>Boolean</td>
<td>Evaluates to true if the left operand expression is greater than the right operand expression. Both operand expressions should be type compatible with Integer.</td>
</tr>
<tr>
<td>Less than or equal (&lt;=)</td>
<td>Boolean</td>
<td>Evaluates to true if the left operand expression is less than or equal to the right operand expression. Both operand expressions should be type compatible with Integer.</td>
</tr>
<tr>
<td>Greater than or equal (&gt;=)</td>
<td>Boolean</td>
<td>Evaluates to true if the left operand expression is greater than or equal to the right operand expression. Both operand expressions should be type compatible with Integer.</td>
</tr>
<tr>
<td>Equal (==)</td>
<td>Boolean</td>
<td>Evaluates to true if the operand expressions evaluate to the same value.</td>
</tr>
<tr>
<td>Not equal (!=)</td>
<td>Boolean</td>
<td>Evaluates to true if the operand expressions do not evaluate to the same value.</td>
</tr>
</tbody>
</table>
| Logical AND or bitwise AND (&) | Boolean or Bitstring | If the operand expressions are Booleans this expression evaluates to true if both operand expressions evaluate to true, otherwise false. If the operand expressions are Bitstrings (or Integers which can be converted to Bitstrings) then the
Logical OR or bitwise OR (||) | Boolean or Bitstring | If the operand expressions are Booleans this expression evaluates to true if either of the operand expressions evaluates to true, otherwise false. If the operand expressions are Bitstrings (or Integers which can be converted to Bitstrings) then the expression performs a bitwise OR operation between bits.

Logical XOR or bitwise XOR (^) | Boolean or Bitstring | If the operand expressions are Booleans this expression evaluates to true if the operand expressions evaluate to different values, otherwise false. If the operand expressions are Bitstrings (or Integers which can be converted to Bitstrings) then the expression performs a bitwise XOR operation between bits.

Conditional AND (&&) | Boolean | This expression evaluates to true if both operand expressions evaluate to true, otherwise false. If the first operand expression evaluates to false, then the second operand expression is never evaluated.

Conditional OR (||) | Boolean | This expression evaluates to true if either of the operand expressions evaluate to true, otherwise false. If the first operand expression evaluates to true, then the second operand expression is never evaluated.

The "equal" and "not equal" expressions are straight forward for operand expressions that are primitive type values, such as integers or booleans. However, if the operand expressions are references to objects, it is the references (not the objects themselves) that are compared for equality. Note that a UML String is not a class, but a primitive type, so two strings can be compared with the "equal" operator.

**Example**

```
Natural n = (Natural) 5 * 6; // Multiplication yields an Integer!
Integer res = 7 / 3; // 'res' becomes 2 (because this is integer division)
Integer rem = 7 % 3; // 'res' becomes 1 (remainder of integer division)
String hw = "hello " + "world!"; // String concatenation
Integer bits = 255 << 2; // 'bits' becomes 1020 (1111111100)
Boolean b = is1() & is2(); // Both operations always invoked
b = is1() && is2(); // is2() not invoked if is1() returns true
if (hw == "hello world!") {} // Strings can be compared with '=='
```

**Classification Expression**

**Alf abstract syntax:** ClassificationExpression [ALF 8.6.5]

**JDT abstract syntax:** InstanceofExpression
A classification expression checks if an operand expression has a certain dynamic type. The referenced type must be a classifier, such as a class or interface. If the operand evaluates to an object whose type either is or inherits/implements the specified type, the classification expression evaluates to true, otherwise false.

If it can be statically determined that a classification expression never can return true, then the expression is treated as semantically incorrect.

**Deviation! Usage of classification expression in UAL is governed by a preference under the Modeling->UML Action Language preference page. The default value of this preference is Error which means that the usage of the instanceof expression is reported as an error in UAL code. The preference value can be set to Warning to treat the use of the instanceof expression as a warning or to Allow to allow the usage of instanceof expression**

```
 Handler h = new Handler();
 Boolean b1 = h instanceof Handler; // true since h is a Handler object
 Boolean b2 = h instanceof AbstractHandler; // true since Handler inherits
 // AbstractHandler
 Boolean b3 = h instanceof IHandler; // true since Handler implements
 // IHandler (indirectly)
 Boolean b4 = h instanceof CommandHandler; // Error! Handler and
 // CommandHandler are unrelated
 // classes.
```

**Conditional Test Expression**

**Alf abstract syntax:** ConditionalTestExpression [ALF 8.7]

**JDT abstract syntax:** ConditionalExpression
A conditional test expression has a Boolean expression and two other operand expressions. If the Boolean expression evaluates to true, then the conditional test expression evaluates to its first operand. Otherwise it evaluates to its second operand.

The type of a conditional test expression is the most specific type to which the types of both operand expressions is assignment compatible.

Example

<table>
<thead>
<tr>
<th>String msg = isError() ? &quot;Error!&quot; : &quot;OK!&quot;;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer i = isOK() ? 1 : -1; // Type of conditional test expression here is</td>
</tr>
<tr>
<td>// Integer since one operand has type</td>
</tr>
<tr>
<td>// UnlimitedNatural and the other Integer.</td>
</tr>
</tbody>
</table>

Assignment Expression

**Alf abstract syntax:** AssignmentExpression [ALF 8.8]

**JDT abstract syntax:** Assignment

An assignment expression consists of two operand expressions - a left hand expression and a right hand expression. The left hand expression must be possible to assign a value to. For example, it may not be a constant expression. The type of an assignment expression is the same as the type of its left hand operand expression.

At run-time the evaluation of an assignment expression is done by first evaluating its right hand operand expression. The value that results from that evaluation is then assigned to the evaluation of its left hand operand. The assignment expression itself has the same evaluation as its right hand side.

If the left hand expression evaluates to a formal parameter reference, then that formal parameter must have direction out or inout. It is not allowed to reassign a formal in parameter.

**Deviation! UAL allows formal in parameters to be redefined.**

For an assignment expression to be legal it is required that the type of the right hand expression is assignment compatible with the type of the left hand expression. A type TR is assignment compatible with TL if one of the following conditions are fulfilled:

1. TR and TL is the same type
2. TR inherits or implements TL, directly or indirectly
3. TR is UnlimitedNatural and TL is Integer
4. TL is the any type
5. TR is UnlimitedNatural or Integer and TL is BitString
6. TR is a collection class and TL is a sequence

In addition to the simple assignment expression on the form \( x = y \), there is also a compound assignment \( x \ op= y \), where \( op \) is any arithmetic or logical operator mentioned in . A compound
assignment $x \ op = y$ is functionally equivalent to a simple assignment $x = x \ op y$. However, if $x$ contains a sequence access expression, it is only evaluated once.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer a = 0;</td>
</tr>
<tr>
<td>a = 1+2; // Right hand side evaluates to 3 and is then assigned to 'a'.</td>
</tr>
<tr>
<td>Integer b = (a = 5); // An assignment expression evaluates to its right hand</td>
</tr>
<tr>
<td>// operand evaluation. 'a' and 'b' here both becomes 5.</td>
</tr>
<tr>
<td>b += 4; // Compound assignment equivalent to $b = b + 4$;</td>
</tr>
</tbody>
</table>

**UAL Libraries**

Now that we have gone through all available UAL statements and expressions we are ready to look into what libraries that can be used from UAL. The Alf standard includes some standard libraries, but these are not yet supported by Rational Software Architect. Instead, there are other UAL libraries available. Some of these, such as the UALRTServices, are only useful when developing particular kinds of models, such as models targeting the RealTime profile. Others are more general and can be used in any kind of model.

Some wizards that create new UML models automatically include package imports for commonly used UAL libraries. If your model does not already include the wanted package imports you have to manually add them. To do this, right-click on the top-level package in the model and select *Add UML - Relationship - Package Import*:

All UAL libraries have names starting with the "UAL" prefix:
If a model that uses UAL libraries will be transformed to source code, such as C++ or Java, you must remember to specify in the transformation configuration which UAL libraries that are used. This is done by means of something called **marking models**. Each UAL library has its own set of marking models, one for each supported target language. A marking model for a particular library and target language defines how usage of that library should be translated to the particular target language. To set-up marking models for the UAL libraries that your model uses, go to the Main tab of the transformation configuration editor, and select the ones your model needs. Here is an example what may be shown in the Main tab of a transformation configuration that transforms a UML model to C++ code targeting the RealTime C++ services library:
Select the marking models that correspond to the libraries that are used by your model. Then press the "Add to Marking Models Tab". The selected marking models will appear in the Marking Models tab. For example, if the marking model for the UALRTServices library is selected, the following entry will appear in the Marking Models tab:

```
pathmap://RT_UAL2CPP_MARKINGS/UALRTServicesCppMarkings.emx
```

It should be noted that the use of marking models is not limited to UAL libraries. In fact, all kinds of libraries, including user-defined, may use marking models to specify how to translate their usage to a target language. In the picture above you can for example see that there is a marking model for the UMLPrimitiveTypes library. If your model uses any UML primitive type, such as String or Boolean, you must remember to add this marking model to the transformation configuration used for transforming your model to target code.

The rest of this chapter describes the available UAL libraries, and provides some examples of how to use their contents from UAL code snippets.

**UALContainers**

This library contains a number of classes and interfaces that represent containers (a.k.a. collections) of different kinds. You can use these to create various kinds of data structures.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitSet</td>
<td>Class</td>
<td>A representation of a Bitset</td>
</tr>
<tr>
<td>FIFOQueue</td>
<td>Class</td>
<td>A concrete representation of a First In First Out</td>
</tr>
<tr>
<td>Class</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LinkedList</td>
<td>Class</td>
<td>A concrete representation of a singly linked list</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>Class</td>
<td>A concrete representation of a priority queue</td>
</tr>
<tr>
<td>Stack</td>
<td>Class</td>
<td>A concrete representation of a Last in First out stack</td>
</tr>
<tr>
<td>TreeSet</td>
<td>Class</td>
<td>A concrete representation of a sorted set which is an ordered collection without duplicates</td>
</tr>
<tr>
<td>Vector</td>
<td>Class</td>
<td>A concrete representation of a vector</td>
</tr>
<tr>
<td>Iterator</td>
<td>Interface</td>
<td>An interface that provides iteration capabilities</td>
</tr>
<tr>
<td>List</td>
<td>Interface</td>
<td>An abstract representation of a list which is an ordered collection which allows duplicates</td>
</tr>
<tr>
<td>Map</td>
<td>Interface</td>
<td>An abstract representation of a key value mapping</td>
</tr>
<tr>
<td>Queue</td>
<td>Interface</td>
<td>An abstract representation of a queue</td>
</tr>
<tr>
<td>Set</td>
<td>Interface</td>
<td>An abstract representation of a Set which is an unordered collection without duplicates</td>
</tr>
</tbody>
</table>

Example

```java
List<Integer> list = new LinkedList<Integer>();
list.add(1);
list.add(5);
Integer i = list.size(); // 'i' becomes 2
```

**UALCore**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Class</td>
<td>A representation of java.lang.Class which is intended as the in-image representation of a class.</td>
</tr>
<tr>
<td>ReflectiveObject</td>
<td>Class</td>
<td>This class is intended to provide reflection like capabilities on a UAL class. Classes inheriting from ReflectiveObject will be able to provide the runtime class information using the getClass() operation. Reflection is mostly used while developing capsule applications where the type descriptor of a particular type needs to be accessed during the run time</td>
</tr>
</tbody>
</table>

Example
A class DataObject that generalizes the ReflectiveObject would have a virtual function getClassData() generated in the source which will return the type descriptor of the class. The class ReflectiveObject translates to RTDataObject in the generated C++ code

```cpp
class DataObject : public RTDataObject
{
public:
    virtual const RObject_class * getClassData( void );
    int intValue;
    DialPlanData dialPlanData;
    virtual ~DataObject( void );
    DataObject( const DataObject & rtg_arg );
    DataObject & operator = ( const DataObject & rtg_arg );
    DataObject( int intVal );
    DataObject( void );
    DataObject( DialPlanData dpData );
};
```

The implementation of this function will return the type descriptor as generated by the RTC++ transformation

```cpp
const RObject_class * DataObject::getClassData( void )
{
    return &RTType_DataObject;
}
```

When sending a signal that requires a data parameter, a call to this function is generated where a type descriptor is expected

```cpp
new RTOutSignal(( & ( outPort ) ), signal, data, data->getClassData()) -> send();
```

Note that though this concept is mainly used in capsule applications, the concept is not limited to Real time applications and hence it is included as a part of the UAL Core library. This library is not complete and is expected to change in the future. It is not part of the standard ALF specification

**UAExceptions**

This library defines several exceptions that are commonly seen in Java programming. Note that exception throwing and catching is not part of the Alf standard. This library is intended to provide model representation of several commonly seen Java exceptions

This library is not complete and is expected to change in the future. Currently, there are no uses of these exception anywhere in the UAL implementation
### Exception Class

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassCastException</td>
<td>Class</td>
<td>Indicates error during casting one type to an incompatible type</td>
</tr>
<tr>
<td>Exception</td>
<td>Class</td>
<td>Super class of all exception</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>Class</td>
<td>Indicates that a method has been passed an inappropriate argument</td>
</tr>
<tr>
<td>IndexOutOfBoundsException</td>
<td>Class</td>
<td>Indicates that an index into a collection or an array is out of range</td>
</tr>
<tr>
<td>IOException</td>
<td>Class</td>
<td>Indicates an error during some Input/Output operation</td>
</tr>
<tr>
<td>RuntimeException</td>
<td>Class</td>
<td>An exception that can be thrown by the Java runtime library during execution of an application.</td>
</tr>
</tbody>
</table>

### UALImpl

This library is supposed to be internal to the UAL implementation and is not supposed to be used directly by the user. The classes in this library are internally used by the implementation to provide support for the Object and String types to make these types similar to the Java types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Class</td>
<td>This class represents java.lang.Object</td>
</tr>
<tr>
<td>StringImpl</td>
<td>Class</td>
<td>This class represents java.lang.String</td>
</tr>
</tbody>
</table>

### UALPrimitiveDatatypes

This library contains some primitive data types that can be used as an alternative, or as a complement, to the predefined UML data types. Note that some of these types are equivalent to corresponding UML predefined types. If you only need to use these types, it is recommended to instead use the corresponding UML predefined types, to make your model more compliant with the UML and Alf standards.

This library is expected to be deprecated in the near future and a new primitive types library is expected to be introduced in accordance with the ALF specification.

**Deviation!** *The primitive data types Natural and BitString that are specified in the Alf standard are not yet supported by UAL.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Primitive Type</td>
<td>Equivalent and compatible with the UML predefined type Boolean.</td>
</tr>
<tr>
<td>byte</td>
<td>Primitive Type</td>
<td>TBD</td>
</tr>
<tr>
<td>char</td>
<td>Primitive Type</td>
<td>TBD</td>
</tr>
<tr>
<td>double</td>
<td>Primitive Type</td>
<td>TBD</td>
</tr>
</tbody>
</table>
## UALRTServices

This library provides UAL access to the RealTime services library, which is a run-time framework for real-time applications. This library is therefore only used by models created with Rational Software Architect RealTime Edition. The RealTime services library exists for different target languages, such as Java and C++. UALRTServices is the UAL-version of this library, and makes concepts such as messaging, timing, concurrency etc. available at the UAL level.

It is beyond the scope of this document to provide a fully detailed description of the RealTime services library. However, since the names of types and operations are the same as are used in the target language versions of the RealTime services library, you can refer to the documentation for these libraries for more information.

### Example

```java
byte b = 5;
float f = 7 / b; // UML does not have a primitive floating point type, so we
// can use the UALPrimitiveTypes.float type to work with
// non-integral values in UAL.
```

### Name | Type | Comment
--- | --- | ---
**Application** | **Class** | This singleton class represents the entire real-time application. It inherits the Controller class. In addition it provides the following operations:

- `getArgCount`
  - Returns the number of arguments that was present on the command line when starting the application.
- `getArgString`
  - Returns the command-line argument with a specified index (zero-based).

**Capsule** | **Class** | This class acts as the implicit base class for all capsule classes. It provides a number of operations that can be used from UAL snippets which have a capsule as its context:

- `rtDeferMessage`
  - Defer the processing of the current message.
- `rtGetController`
  - Returns the controller for the physical thread on which the capsule executes.
- `rtGetMsgData`
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CapsuleId</td>
<td>A representation of the ID of a capsule instance.</td>
</tr>
<tr>
<td>CapsuleRole</td>
<td>A representation of a UML part which may contain capsule instances. The main usage for this class is to specify the role (i.e. part) where to incarnate a particular capsule. The following operations are provided:</td>
</tr>
<tr>
<td></td>
<td>- getDefaultSize</td>
</tr>
<tr>
<td></td>
<td>TBD</td>
</tr>
<tr>
<td></td>
<td>- size</td>
</tr>
<tr>
<td></td>
<td>TBD</td>
</tr>
<tr>
<td>Controller</td>
<td>An instance of this class is responsible for executing a number of capsule instances in the same physical thread. It provides the following operations:</td>
</tr>
<tr>
<td></td>
<td>- abort</td>
</tr>
<tr>
<td></td>
<td>Terminates the controller. All capsule instances managed by this controller will be destroyed, and messages waiting in the message queue will be deleted. If the controller is run by the main thread, then the entire application quits.</td>
</tr>
<tr>
<td></td>
<td>- getError</td>
</tr>
<tr>
<td></td>
<td>Returns an Error object describing the latest error that has occurred in the controller.</td>
</tr>
<tr>
<td></td>
<td>- perror</td>
</tr>
<tr>
<td></td>
<td>Prints to stderr the latest error that has occurred in the controller, prefixed with an optional user-defined message.</td>
</tr>
<tr>
<td></td>
<td>- strerror</td>
</tr>
<tr>
<td></td>
<td>Returns an error message for the latest error that has occurred in the controller.</td>
</tr>
<tr>
<td>External</td>
<td>The External class (or rather External::Base)</td>
</tr>
<tr>
<td>Frame</td>
<td>Class</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
</tbody>
</table>
|       |       | - **destroy**  
|       |       | Destroys either one specific capsule instance, or all capsule instances located in a particular capsule role. |
|       |       | - **getOwner**  
|       |       | TBD |
|       |       | - **incarnate**  
|       |       | Creates a new capsule instance and puts it into a specified capsule role. This API should be used instead of creating the capsule instance by means of a `new` expression. |
|       |       | - **incarnationAt**  
|       |       | Returns the capsule instance at a specified index in a capsule role. |
|       |       | - **plugin**  
|       |       | Puts an already incarnated capsule instance into a specified capsule role at a specified index. A capsule instance can at most be part of one capsule role. |
|       |       | - **unplug**  
|       |       | Removes a capsule instance from its current capsule role. |
| Log   | Class | The Log class (or rather Log::Base) provides access |
to the system log. It is accessed by declaring a port using the Log protocol. The following operations are available:

- **close**
  Closes the log. All further output to the log will be disabled.

- **commit**
  Ensures that all data written to the log has been flushed.

- **cr**
  Begins a new line in the log.

- **crtab**
  Begins a new line in the log, and starts it with a number of tab characters.

- **log**
  Writes a piece of data to the log, followed by a carriage return so that the next log entry appears on a new line.

- **open**
  Opens the log, making it enabled for writing.

- **show**
  Writes a piece of data to the log. No carriage return is appended, which means that the next log entry will appear on the same line.

- **space**
  Writes a space character to the log.

- **tab**
  Writes a tab character to the log.

---

**RTString**

**Class**

This string type is an alternative to using the UML predefined type String. It provides more operations than are available on the UML String type:

- **charAt**
  Returns the character at the specified index in the string.

- **compareTo**
  Compares the string with another string for equality.

- **compareToIgnoreCase**
  Compares the string with another string for equality, ignoring differences in case.

- **concat**
  Concatenates the string with another string.

- **indexOf**
<table>
<thead>
<tr>
<th>RTTcpSocket</th>
<th>Class</th>
<th>This is a utility class that provides services related to TCP based communication with other applications. It provides the following operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- <strong>acceptFrom</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts a connection request on a server socket.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>close</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Closes the underlying TCP socket.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>create</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Creates the underlying TCP socket.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>getState</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the current state of the socket. The socket state is described by the RTTcpSocketState enumeration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>hasData</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For a server socket this operation tells whether a new client request has come or not. For a client socket this operation tells</td>
</tr>
<tr>
<td></td>
<td></td>
<td>whether there are any characters to read from the socket.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>listen</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Makes a server socket start listening for client requests on a particular port.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>read</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reads characters from the socket.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>write</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Writes characters to the socket.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTThread</th>
<th>Class</th>
<th>TBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTTimeSpec</td>
<td>Class</td>
<td>A representation of timer values. This class is used when accessing the Timing API. It provides the following operations:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>getClock</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is a static operation that initializes an RTTimeSpec object with the current time. RTTimeSpec also provides a constructor that allows</td>
</tr>
</tbody>
</table>

| | |

Searches the string for the first occurrence of a particular character or substring.

- **lastIndexOf**
  Searches the string for the last occurrence of a particular character or substring.
- **length**
  Returns the number of characters in the string.
- **toCharArray**
  Converts the string to an array of characters.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing</td>
<td>Class</td>
<td>The Timing class (or rather Timing::Base) provides services related to timers. It is accessed by declaring a port using the Timer protocol. The following operations are available:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cancelTimer</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>informAt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>informEvery</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>informIn</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>timeout</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>timeouts</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A timer is identified by an instance of the class <code>Timing::Request</code>.</td>
</tr>
<tr>
<td>Error</td>
<td>Enumeration</td>
<td>This enumeration contains literals for the different kinds of errors that may occur when using the RealTime services library. The 'ok' literal is used to indicate a successful result of an operation (i.e. no error occurred).</td>
</tr>
<tr>
<td>Priority</td>
<td>Enumeration</td>
<td>This enumeration contains literals that specify different levels of priority for messages. It is used when sending messages and setting timers.</td>
</tr>
<tr>
<td>RTTcpSocketState</td>
<td>Enumeration</td>
<td>This enumeration contains literals that specify the state of a TCP socket. It is used by the RTTcpSocket API.</td>
</tr>
</tbody>
</table>
| ProtocolRole | Interface | Represents the type of a port, and provides information about the set of messages that can be
sent or received by the port. The operations provided by this interface are available on all ports:

- **bindingNotification**
  Controls whether there should be notifications sent (rtBound and rtUnbound) when the port gets bound or unbound.

- **bindingNotificationRequested**
  Returns true if binding notifications are set-up to occur for the port.

- **createOutSignal**
  Creates a signal instance that can be sent out of the port.

- **deregisterSAP**
  Deregister a service accessor port.

- **deregisterSPP**
  Deregister a service provider port.

- **getRegisteredName**
  Returns the registered service name for a port.

- **isBoundAt**
  Checks if a particular port instance is currently bound to another port.

- **isRegistered**
  Checks if an unwired port is currently registered or not.

- **purge**
  Returns the number of messages that have been deleted from the defer queue for the port.

- **purgeAt**
  Returns the number of messages that have been deleted from the defer queue for a particular port instance.

- **recall**
  Recalls the first message that has been deferred for any of the port instances held by the port. The recalled message is placed at the end of the main message queue.

- **recallAll**
  Recalls all messages that have been deferred for any of the port instances held by the port. The recalled messages are placed at the end of the main message queue.

- **recallAllAt**
  Recalls all messages that have been deferred
for a particular port instance. The recalled messages are placed at the end of the main message queue.

- **recallAllFront**
  Recalls all messages that have been deferred for a particular port instance. The recalled messages are placed at the front of the main message queue.

- **recallAt**
  Recalls the first message that has been deferred for a particular port instance. The recalled message is placed at the end of the main message queue.

- **recallFront**
  Recalls the first message that has been deferred for any of the port instances held by the port. The recalled message is placed at the front of the main message queue.

- **registerSAP**
  Register a service accessor port.

- **registerSPP**
  Register a service provider port.

- **resize**
  TBD

- **rtGetDescriptor**
  TBD

- **size**
  Returns the number of port instances held by a port.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>External Protocol</th>
<th>Used when declaring External ports. See the class External above.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>Frame Protocol</td>
<td>Used when declaring Frame ports. See the class Frame above.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Log Protocol</td>
<td>Used when declaring Log ports. See the class Log above.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Timing Protocol</td>
<td>Used when declaring Timing ports. See the class Timing above.</td>
</tr>
</tbody>
</table>

**Example**

```java
// Get the Controller from the context of a capsule, and terminate it!
Controller ctrl = rtGetController();
ctrl.abort();

// Create a new capsule instance (incarnation) and put it in a particular
// capsule role. Use the controller to check for errors.
frame.incarnate(role);
if (ctrl.getError() != Error.ok) {
```

Using the UML Action Language in Rational Software Architect

© Copyright IBM Corporation 2011. All rights reserved.
// Some error occurred during incarnation
}
// Destroy all capsule instances in a particular capsule role
frame.destroy(role);
// Print a text message to the system log.
log.log("Hello!");
// Enable an external port (allowing an external thread to raise an event
// on it)
externalPort.enable();
// Set a timer that will expire in 4 seconds, and then raise a timeout event
// on a 'timer' port.
RTTimespec delta = new RTTimespec(4,0);
Timing.Request tid = timer.informIn(delta);
delta.destroy();

UALServices
This library provides miscellaneous utilities that can be useful when developing applications
using UAL. The elements in this model library represent various utilities that applications can
expect from standard platforms like console output, socket connections etc.
This library is not complete and is expected to change in the future. This library is not in
accordance with the ALF specification and is a non-standard extension specific to the RSA
implementation of UAL

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>Class</td>
<td>A class that represents a condition to provide mutual exclusion for concurrent access</td>
</tr>
<tr>
<td>FileInputStream</td>
<td>Class</td>
<td>A concrete representation of an input stream that would allow users to read from a file on disk</td>
</tr>
<tr>
<td>FileOutputStream</td>
<td>Class</td>
<td>A concrete representation of an output stream that would allow users to write to a file on disk</td>
</tr>
<tr>
<td>InetAddress</td>
<td>Class</td>
<td>A representation of an internet address</td>
</tr>
<tr>
<td>InputStream</td>
<td>Class</td>
<td>An abstract representation of an input stream that would allow the users to read from a stream of bytes</td>
</tr>
<tr>
<td>InterruptedException</td>
<td>Class</td>
<td>An exception that indicates that a thread was interrupted during its execution</td>
</tr>
<tr>
<td>Lock</td>
<td>Class</td>
<td>A representation of a Lock to allow mutual exclusion for concurrent access</td>
</tr>
<tr>
<td>OutputStream</td>
<td>Class</td>
<td>An abstract representation of an output stream that would allow users to write to a stream of bytes</td>
</tr>
<tr>
<td>PrintStream</td>
<td>Class</td>
<td>A concrete representation of an output stream that would allow the users to write to a stream of characters like the console</td>
</tr>
<tr>
<td>ServerSocket</td>
<td>Class</td>
<td>A concrete representation of a connection oriented socket that would listen on a port and accept</td>
</tr>
<tr>
<td>Class/Interface</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>Socket</td>
<td>A concrete representation of a client socket that would connect to a server and exchange data</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>A concrete representation of a thread of execution</td>
<td></td>
</tr>
<tr>
<td>UnknownHostException</td>
<td>An exception that indicates failure to connect to a server socket</td>
<td></td>
</tr>
<tr>
<td>Runnable</td>
<td>An interface that represents an instance of an object which can be run by a thread</td>
<td></td>
</tr>
</tbody>
</table>