Data Model Documentation

Table of Contents

Data Model Documentation . . . 1
- Overview of the Data Model . . . 1
- Class Hierarchy . . . 3
- Boolean Data Objects . . . 4
- Numeric Data Objects . . . 5
- String Data Objects . . . 6
- Generic Data Objects . . . 7
- CallLiteral Data Objects . . . 8
- Extending the Data Objects . . . 9

Data Model Documentation

These HTML pages are available in book form as a postscript file and as a. PDF file.

Overview of the Data Model

The Able data model provides a common set of literal and variable data objects that are used in other Able components, such as filters, rule systems, and inference engines, and can be extended by others for private needs.

Literals are immutable objects and are assigned a value at construction time. *Variables* can be assigned values at any time, either from other variables or from literals. A set of *operators* allows comparing literals and variables and assigning new values to variables.

All literals and variables are *typed*, and fall into one of four categories:

1. Boolean - values are either *true* or *false*.
2. Numeric - values are numbers, which are represented internally as Java *double*s.
3. String - values are Java *String*s.
4. Generic - values are any Java *Object*.

When assigning or comparing values, some attempt is made at converting one data type to another. For example, a numeric variable may be assigned a value from a boolean variable (or literal) since a boolean data object, whose value is either true or false, is considered to have a *numeric* value of either 1.0 or 0.0, respectively. However, not all combinations of data assignment and comparison are possible, and *AbleDataException*s are thrown when appropriate. For example, the *less than* operator has no meaning for generic data types whose "real" Java data types are not numbers or strings at the time an assignment or comparison is made.

In adition to the four data types mentioned above, there is a fifth data type, called a *call literal*. Like other literals, it is a *read-only* object, but one that wraps an Able sensor or effector. This means that every time a call literal's current value is requested, the data that is returned is actually the data obtained by calling the wrappered sensor or effector. The returned data can be any Java *Object*, but must be, of course, a type expected by the receiving application.

All literals and variables implement, at minimum, the following methods found in the **AbleRd** interface, which is the interface that must be implemented by all *"readable"* data objects:

```
  public boolean     getBooleanValue()
  public AbleLiteral getFuzzyValue()
  public Object      getGenericValue()
  public double      getNumericValue()
  public String      getStringValue()
  public AbleLiteral getValue()
```

AbleLiteral is a partial implementation of the AbleRd interface; AbleBooleanLiteral, AbleCallLiteral, AbleGenericLiteral, AbleNumericLiteral, and AbleStringLiteral are the concrete implementations.

Variables implement, at minimum, the additional following methods found in the **AbleWr** interface, which is the interface that must be implemented by all *"writable"* data objects:

```
public void setBooleanValue(boolean)
public void setFuzzyValue(AbleLiteral)
public void setGenericValue(Object)
public void setNumericValue(double)
public void setStringValue(String)
public void setValue(AbleLiteral)
```

AbleVariable is a partial implementation of the AbleWr interface; AbleBooleanVariable, AbleCategoricalVariable, AbleContinuousVariable, AbleDiscreteVariable, AbleGenericVariable, AbleNumericVariable, and AbleStringVariable are the concrete implementations.

Given the "get" and "set" methods listed above, the following coding is possible:

```
double  aDbl  = someNumericVar.getNumericValue();
boolean aBool = someNumericVar.getBooleanValue();
String  aStr  = someNumericVar.getStringValue();

anotherNumericVar.setNumericValue(aDbl);
anotherNumericVar.setNumericValue(someNumericVar.getNumericValue());
anotherNumericVar.setStringValue(aStr);

anotherNumericVar.setValue(someNumericVar.getValue());

anyTypeOfVar.setValue(anyOtherVar.getValue());
anyTypeOfVar.setValue(anyTypeOfLiteral);

if (someNumericVar.getNumericValue() <= anotherNumericVar.getNumericValue())
```

However, all of the assignment and comparison operations above can be better accomplished using the built-in "operators" defined in the **AbleRd** and **AbleWr** classes. AbleRd, the interface that defines "readable" data objects, defines these comparison operators, which are implemented in all AbleLiterals and AbleVariables:

```
public boolean cmpEq  (AbleRd) // ==
public boolean cmpGt  (AbleRd) // >
public boolean cmpGtEq(AbleRd) // >=
public double  cmpIs  (AbleRd) // is, fuzzy compare
public boolean cmpLt  (AbleRd) // <
public boolean cmpLtEq(AbleRd) // <=
public boolean cmpNeq (AbleRd) // !=
```

AbleWr, the interface that defines "writable" data objects, defines these assignment operators, which are implemented in all AbleVariables:

```
public void asgnEq(AbleRd)         // boolean assignment
public void asgnIs(AbleRd)         // fuzzy assignment
public void asgnIs(AbleRd, double) // fuzzy asignment with correlation
```

Given the above comparison and assignment operators, the following style of coding is possible:

```
if ( someNumericVar.cmpLtEq(anotherNumericVar) ) {
  someStringVar.asgnEq(anotherStringVar);
}
```

...and so on. In summary, these are the comparison operators implemented by all AbleLiterals and AbleVariables:

Method	Function	Usage
cmpEq	Boolean compare, equal to (==)	AbleRd.cmpEq(AbleRd)
cmpGt	Boolean compare, greater than (>)	AbleRd.cmpGt(AbleRd)
cmpGtEq	Boolean compare, greater than or equal to (>=)	AbleRd.cmpGtEq(AbleRd)
cmpIs	Fuzzy compare (is)	FsVarContinuous.cmpIs(FsSet)
cmpLt	Boolean compare, less than (<)	AbleRd.cmpLt(AbleRd)
cmpLtEq	Boolean compare, less than or equal to (<=)	AbleRd.cmpLtEq(AbleRd)
cmpNeq	Boolean compare, not equal to (!=)	AbleRd.cmpNeq(AbleRd)

And these are the assignment operators implemented by all AbleVariables:

Method	Function	Usage
asgnEq	Boolean assignment (=)	AbleVariable.asgnEq(AbleRd)
asgnIs	Fuzzy assignment (is)	FsVarContinuous.asgnEq(FsSet)

The remainder of this paper contains further details about all literals and variables in the Able data package.

Class Hierarchy

Here are the classes that make up the Able Data Model:

[IMAGE]

Remember that all literals and variables are classifed into basic types:

1. Boolean
2. Numeric
3. String
4. Generic

and lastly,

5. CallLiteral

Essentially, any data object can be compared to any other data object, and any data object and be assigned to any variable. This is accomplished through the various `get...Value()` and `set...Value()` methods. The following sections list the data objects in each category (boolean, numeric, string, and generic) and contain a chart showing what each `get...Value()` method returns. Each method's return value depends, of course, on the data object's actual current "raw" value at the time the method is invoked.

Looking at the chart for boolean data types, for example, you can see that when a boolean variable's current value is *true*, `getBooleanValue()` returns *true*, `getNumericValue()` returns *1.0*, and `getStringValue()` returns *"True"*, but when the current value is *false* `getBooleanValue()` returns *false*, `getNumericValue()` returns *0.0*, and `getStringValue()` returns *"False"*. Thus it is possible to assign a numeric variable a value from a boolean literal or variable, and so on.

When a particular combination is not possible, or when the current raw value of a particular data object doesn't lend itself to conversion to another data type, an AbleDataException is thrown. This may happen, for example, when trying to assign a numeric variable a value from a string literal or string variable. If the source string data object contains something like "123.456" or "-0.005", the assignment is possible, because the string can be parsed into a number. But if the string data object contains something like "Kilroy was here", an AbleDataException is thrown.

Boolean Data Objects

There are two boolean data objects. They are:

- AbleBooleanLiteral
- AbleBooleanVariable

Both the literal and the variable must be given an initial value of either *true* or *false* when created.

For convenience, there are two pre-defined, static AbleBooleanLiterals: **AbleData.True** and **AbleData.False**.

Return values from boolean data object get...Value() methods					
Boolean object's current value	**getBooleanValue()**	**getGenericValue()**	**getNumericValue()**	**getStringValue()**	**getValue()**
true	true	new Boolean(true)	1.0	"True"	new AbleBooleanLiteral(true)
false	false	new Boolean(false)	0.0	"False"	new AbleBooleanLiteral(false)

Numeric Data Objects

The numeric data objects are:

- AbleContinuousVariable

  Continuous variables are numeric variables whose current value must be between a minimum and a maximum, inclusive, both of which are specified when the variable is created. Any attempt to set a continuous variable to a value outside the specified range results in an AbleDataException.

  For historical reasons, continuous variables have an initial value of Double.NaN.

- AbleDiscreteVariable

  Discrete variables are numeric variables whose current value must be a value chosen from a finite list of numeric values. The list of acceptable values can be specified when a discrete variable is created, or it can be set for the variable at anytime. Methods are also provided to add and remove values from the list of acceptable values. However, changing a variable's acceptable value list after a variable is created and used can have ramifications for the variable's current value. Any attempt to set a discrete variable to a value not in the acceptable value list results in an AbleDataException.

  For historical reasons, discrete variables have an initial value of Double.NaN.

- AbleNumericLiteral
- AbleNumericVariable

  Both the literal and the variable must be given an initial value when created. The value can be any number as values are completely unrestricted.

Return values from numeric data object get...Value() methods

Numeric object's current value	**getBooleanValue()**	**getGenericValue()**	**getNumericValue()**	**getStringValue()**	**getValue()**
0.0	false	new Double(0.0)	0.0	"0.0"	new AbleNumericLiteral(0.0)
any **non-zero** number, n.m	true	new Double(n.m)	n.m	"n.m"	new AbleNumericLiteral(n.m)

String Data Objects

The string data objects are:

- AbleCategoricalVariable

  Categorical variables are string variables whose current value must be a value chosen from a finite list of strings. The list of acceptable strings can be specified when a categorical variable is created, or it can be set for the variable at anytime. Methods are also provided to add and remove strings from the list of acceptable strings. However, changing a variable’s acceptable value list after a variable is created and used can have ramifications for the variable’s current value. Any attempt to set a categorical variable to a string not in the acceptable string list results in an AbleDataException.

  For historical reasons, categorical variables have an initial value of AbleData.StringNull ("Able_NULL_Able").

- AbleStringLiteral
- AbleStringVariable

  Both the literal and the variable must be given an initial value when created. The value can be any string as values are completely unrestricted.

Return values from string data object get...Value() methods

String object’s current value	**getBooleanValue()**	**getGenericValue()**	**getNumericValue()**	**getStringValue()**	**getValue()**
"true" (case insensitive)	true	new String("true")	AbleDataException	"true"	new AbleStringLiteral("true")
"false" (case insensitive)	false	new String("false")	AbleDataException	"false"	new AbleStringLiteral("false")
string form of a parsable number, "n.m"	AbleDataException	new String("n.m")	(Double.valueOf("n.m")).doubleValue()	"n.m"	new AbleStringLiteral("n.m")
any other string, "foo"	AbleDataException	new String("foo")	AbleDataException	"foo"	new AbleStringLiteral("foo")

Behavior of string data object set...Value() methods					
Source object's current value	**setBooleanValue()**	**setGenericValue()**	**setNumericValue()**	**setStringValue()**	**setValue()**
boolean true	setStringValue(true)	N/A			
boolean false	setStringValue(false)	N/A			
Boolean, b	N/A	setBooleanValue(b.booleanValue())	N/A		
Number, n	N/A	setNumericValue(n.doubleValue())	N/A		
String, s	N/A	setStringValue(s)	N/A		
AbleLiteral, a	N/A	setValue(a)	N/A		
Object, o	N/A	AbleDataException	N/A		
any double, n.m	N/A		setStringValue(Double.toString(n.m))	N/A	
any String, "foo"	N/A			"foo" For categorical variables, "foo" must be in the list of acceptable strings.	N/A
any AbleLiteral, a	N/A				calls set...Value(a.get...Value()) depending on type of literal

Generic Data Objects

The generic data objects are:

- AbleGenericLiteral
- AbleGenericVariable

  Both the literal and the variable must be given an initial value when created. The value can be any Java Object as values are completely unrestricted.

Return values from generic data object get...Value() methods					
Generic object's current value	**getBooleanValue()**	**getGenericValue()**	**getNumericValue()**	**getStringValue()**	**getValue()**
Boolean, b	b.booleanValue()	b	1.0 if true; 0.0, if false	"True" if true; "False" if false	new AbleGenericLiteral(b)
Number, n	false if n.doubleValue()==0.0; true otherwise	n	n.doubleValue()	Double.toString(n.doubleValue())	new AbleGenericLiteral(n)

String, s	true if s.equalsIgnoreCase("true");
	false if s.equalsIgnoreCase("false");
	otherwise AbleDataException
		s	(Double.valueOf(s)).doubleValue()
			or
			AbleDataException
				s	new AbleGenericLiteral(s)

AbleLiteral, a	a.getBooleanValue()	a	a.getNumericValue()	a.getStringValue()	new AbleGenericLiteral(a)
any other Object, o	AbleDataException	o	AbleDataException	o.toString()	new AbleGenericLiteral(o)

CallLiteral Data Objects

The call literal objects are:

- AbleCallLiteral

A call literal is a wrapper for an Able sensor or effector. This means that every time a call literal's current value is requested, the data that is returned is actually the data obtained by immediately and synchronously calling the wrappered sensor or effector. Call literals more or less expect that the data returned by the wrappered sensor or effector is a Boolean, Number, or String, but in reality, any Java *Object* is allowed. The returned data must be, of course, a type expected by the receiving application, which will most likely use the `getGenericValue()` method to obtain the call literal's value.

Call literals are unique in another respect: they can be used only on the right-hand side of a compare or assignment operation. Attempts to use them on the left-hand side result in an AbleDataException. For example, given two string literals, the literals may be used in either of the following ways and obtain identical results:

```
boolean test1 = stringLitA.cmpEq(stringLitB);
boolean test2 = stringLitB.cmpEq(stringLitA);
```

But when using a call literal in the expression, the following happens:

```
boolean test1 = stringLitA.cmpEq(callLitA);   // Perfectly OK
boolean test2 = callLitA.cmpEq(stringLitA);   // AbleDataException !
```

The reason for this anomaly is simply performance: it is just too inefficient to test for and convert data types when a call literal, which can represent any data type, is on the left-hand side of an expression. You might reason, then, and correctly, too, that the data type on the left-hand side of an expression sets the tone for any conversions and compares that need to be done.

A `get...Value()` chart is not given here, because the code dealing with return values from sensors and effectors is too complex to summarize neatly in a table.

Extending the Data Objects

While it is not possible to add completely new data types to the Able data model, it is possible to extend any of the existing data objects with additional or modified behavior. If you want to create a *ComplexNumberVariable*, for example, you might start by extending the AbleNumericVariable class, the AbleContinuousVariable class, or the AbleDiscreteVariable class, depending on whether you want your complex numbers completely unrestricted, limited within a certain range, or limited to one-of-n complex values, respectively. If you choose to extend AbleContinuousVariable so that you can limit your complex numbers to a range, you probably need to override the setDiscourseLo(), setDiscourseHi(), and withinUniversOfDiscourse() methods, among others, to get the desired behavior. You also need to override or perhaps even add new "operator" methods appropriate for your new ComplexNumber data object.

Last modified: Thu Aug 24 10:09:33 CDT 2000