Databoard dev

Databoard 0.5.2 Developer Manual

DataType

In Databoard all values have a type representation, DataType.java. It is the base abstract class for all concrete type classes (See table below). There is a facade class utility (DataTypes) that provides functions to most of the datatype library's features.

org.simantics.databoard.type.

Class	Description
DataType	Base class for all data types
RecordType	Record
ArrayType	Array - an ordered sequence of elements of one type.
MapType	Map - an ordered map of keys to values.
UnionType	Union
BooleanType,IntType,LongType,FloatType,DoubleType	Primitive and numeric types
StringType	String
OptionalType	Optional value
VariantType	Variant value

DataType can be acquired or created using one of the following methods:

- Construct new
- Constant
- Reflection-Read from a Class
- Read from string of the text notation.

```
    DataType type = new DoubleType();
    DataType type = DoubleType.INSTANCE;
    DataType type = DataTypes.getDataType( Double.class );

DataTypes.addDefinition("type Node = { id : String; children : Node[]
}");
    DataType type = DataTypes.getDataType("Node");
```

Java implementation does not support parametrised types.

Reflection

Data Type and Binding is read automatically from a Class.

```
DataType type = DataTypes.getDataType( Foo.class );
Binding binding = Bindings.getBinding( Foo.class );
```

Classes are RecordTypes

```
class Foo {
    public int x, y, z;
}

type Foo = { x : Integer, y : Integer, z : Integer }
```

There are three types of classes supported, and consequently three ways how objects are constructed.

Record-like classes:

```
class Foo {
    public String name;
    public Object value;
}
```

Immutable classes:

```
class Foo {
    private String name;
    private Object value;

    public Foo(String name, Object value) {
        this.name = name;
        this.value = value;
    }

    public String getName() {
        return name;
    }

    public Object getValue() {
        return value;
    }

}
```

Bean-like classes:

```
class Foo {
    private String name;
    private Object value;
```

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

        public void setValue(Object value) {
            this.value = value;
        }

        public String getName() {
            return name;
        }

        public Object getValue() {
            return value;
        }

    }
```

Static and transient fields are omited:

```
    static final long serialVersionUID = -3387516993124229943L;
    transient int hashCode;
```

Enumerations are Union Types

```
    enum Cars { Ferrari, Porche, Lamborghini, Jaguar }

    type Cars = | Ferrari {} | Porche {} | Lamborghini {} | Jaguar {}
```

Other exceptions:

- `java.lang.Object` is `Variant`.
- `java.lang.Set<T>` is `Map(T, {})`.
- `java.lang.TreeSet<T>` is `Map(T, {})`.
- `java.lang.HashSet<T>` is `Map(T, {})`. (Note HashSet binding has very low performance.)
- `java.lang.Map<K, V>` is `Map(K, V)`.
- `java.lang.TreeMap<K, V>` is `Map(K, V)`.
- `java.lang.HashMap<K, V>` is `Map(K, V)`. (Note HashMap binding has very low performance.)
- `java.lang.List<T>` is `Array(T)`.
- `java.lang.ArrayList<T>` is `Array(T)`.
- `java.lang.LinkedList<T>` is `Array(T)`.
- `void` is `{}`.
- The stacktrace of `Exception.class` is omited.

Annotations

Java Classes / Fields can be annotated with the following annotations (**org.simantics.databoard.annotations**).

UnionTypes are abstract classes or interfaces with `@Union` annotation.

```
    @Union({A.class, B.class}) interface Union1 {□□
    }

    class A implements Union1 {
        public int value;
    }

    class B implements Union1 {
        public String name;
    }
```

`@Referable` denotes that the class has recursion and is a referable record.

```
    public @Referable class Node {
        public Node[] children;
    }
```

Fields that can have `null` value have `@Optional` annotation.

```
    @Optional String name;
```

String valid values are set with `@Pattern` as regular expression. ([1])

```
String @Pattern("(19|20)\\d\\d[- /.](0[1-9]|1[012])[-
/.](0[1-9]|[12][0-9]|3[01])") date;

type Date = String( Pattern = "(19|20)\d\d[- /.](0[1-9]|1[012])[-
/.](0[1-9]|[12][0-9]|3[01])" )
```

String content type is set with a `@MIMEType`. (MIME Type [2])

```
    @MIMEType("text/xml") String document;
```

Array size restricted with @Length.

```
    @Length("[0..10]") int[] array;
    @Length({"[320]", "[240]"}) int[][] image;
```

Valid numeric range is set with @Range.

```
    @Range("[0..100]") double alpha;
    @Range("[0..]" double length;
```

`Range` and `Length` notation:

- Exact Value "0"
- Exclude all "()"

- Unlimited "[..]"
- Inclusive range "[0..100]"
- Exclusive range "(0..100)"
- Inclusive lower bound and exclusive upper bound "[0..100)"

Engineering unit type is given with @Unit.

```
    @Unit("km/h") double maxVelocity;
```

Parsing

DataTypes are parsed using `DataTypes.dataTypeRepository`.

```
DataTypes.addDefinition("type Node = { id : String; children : Node[]
}");
    DataType type = DataTypes.getDataType("Node");
```

Types are printed to types and definitions with

```
    String type = type.toString();

    DataTypeRepository repo = new DataTypeRepository();
    repo.add("temp1", type);
    String typeDef = repo.toString();
```

Binding

Databoard values are platform, language and transport format neutral. In Java Environment, values are Java Objects and are handled with Bindings (Binding.java). It is a bridge between `DataType` and a Java Class, and an environment neutral access to the content of the value.

For instance, take a java.lang.Double. Its instance is the container (`private final double value;`) of the data and its Binding (DoubleBinding) is the access (`.valueOf()`, `.getDouble()`) to the data.

```
    Java Object + Binding = Databoard Value
```

Bindings have the exact same composition tree structure as its respective `DataType` - structural types have structural Bindings, and primitive types a single binding.

org.simantics.databoard.binding.

Class	**Description**
DataBinding	Base class for all data Bindings
RecordBinding	Record
ArrayBinding	Array - an ordered sequence of elements of one value.
MapBinding	Map - an *ordered* map of keys to values.
UnionBinding	Union
BooleanBinding,IntBinding,LongBinding,FloatBinding,DoubleBinding	Primitive and numeric Bindings

StringBinding	String
OptionalBinding	Optional value
VariantBinding	Variant value

Binding can be acquired or created using one of the following methods:

- Constructor
- Constant
- Reflection-Read from a Class
- Created using BindingScheme

```
Binding binding = new DoubleBinding( doubleType );
Binding binding = new RecordBinding() { ... };
Binding binding = DoubleBinding.INSTANCE;
Binding binding = Binding.getBinding( Double.class );
Binding binding = Binding.getBinding( DoubleType.INSTANCE );
```

Mapping Scheme

A *binding scheme* associates some data types with a unique binding. The mapping of types to bindings is bijective, there is one binding for each type and vice-versa.

`GenericBindingScheme` is a scheme that provides a fully implementing mutable binding for all data types. The Class mapping for each type is listed below.

Type	**Class**
`BooleanType`	`MutableBoolean.class`
`ByteType`	`MutableByte.class`
`FloatType`	`MutableFloat.class`
`DoubleType`	`MutableDouble.class`
`IntegerType`	`MutableInt.class`
`LongType`	`MutableLong.class`
`StringType`	`MutableString.class`
`UnionType`	`GenericBinding.TaggedObject.class`
`OptionType`	`ValueContainer.class`
`RecordType`	`Object[].class`
`ArrayType`	`ArrayList.class`
`MapType`	`TreeMap.class`
`VariantType`	`Variant.class`

There is are also a special case *Unbound* binding scheme. It binds to nothing `null`.

```
Binding binding = DataTypes.getUnboundBinding( DoubleType.INSTANCE )
```

Serialization

Serializer.java is a class that serializes Values into and from binary serialization format. It follows the Databoard Binary File Format.

```
Binding binding = new DoubleBinding();
Serializer serializer = binding.serializer();
byte[] data = serializer.serialize( new Double( 100.0 ) );

Double value = (Double) serializer.deserialize( data );
```

Files can be partially accessed using BinaryAccessor, see Accessors. This is useful when handling larger than memory files.

Validation

Value can be *well-formed* or *valid*. The domain of valid values are defined with restrictions in data types, and `@Length`, `@Range`, `@Pattern` and `@MIMEType` Annotations in Classes

Validation mechanism in Binding asserts that the instance is a valid value of the respective Data Type.

```
try {
    Binding.assertInstaceIsValid( object );
} catch( BindingException e ) {
    // In-valid object
}
```

Other Notes

- `Binding` is a `Comparator`, all data values are comparable, the order is defined in Databoard_Specification.
- `Binding#createDefault()` and `Binding#createRandom(int)` creates a valid instance of the DataType.
- `Binding#clone(Object)` creates a new instance with same content.

Parsing & Printing

Data values are printed and parsed of the Text notation with the following `Binding` methods:

```
String text = binding.printValue( value, true );

Object value = binding.parseValue( text );
```

And also to value definitions *`name : type = value`*

```
StringBuilder sb = new StringBuilder();
DataValueRepository repo = new DataValueRepository();
repo.add( "temp", value );
binding.printValue( value, sb, repo, true );
String text = sb.toString();
```

```
    Object value = binding.parseValueDefinition( text );
```

Adapter

There can be different Java Class Bindings for a single data type. For example, `Double` type can be have bindings `DoubleJavaBinding` and `MutableDoubleBinding` to two respective classes `java.lang.Double` and `MutableDouble`. Instance of one binding can be adapted to instance of another with an `Adapter`.

```
java.lang.Double double = Bindings.adapt( new MutableDouble(5.0),
mutableDoubleBinding, doubleJavaBinding );
java.lang.Double double = adapter.adapt( new MutableDouble(5.0),
mutableDoubleBinding, doubleJavaBinding );
```

Adapter can be created automatically or implemented self.

```
    Adapter adapter = new Adapter() { ... };
Adapter adapter = Bindings.getAdapter( domainBinding, rangeBinding );
Adapter adapter = Bindings.adapterCache.getAdapter(domain, range,
false, false);
```

The instance given as the argument to `Adapter#adapt(Object)` may be re-used completely, partially or cloned in the adapted result object. A clone can be guaranteed, if clone argument is `true` when adapter is created. Note, immutable classes, eg. java.lang.Integer, are never cloned, never reinstantiated.

```
Adapter cloner = Bindings.adapterCache.getAdapter(domain, range, false,
true);

Rectangle2D rect2 = Bindings.clone( rect1, rectBinding, rectBinding );
```

Type Conversion

In some cases different types may be are type-conversion compatible. An instance of one type is convertible to instance of another.

Engineering Units of same quantity are convertible.

```
class CarSI {
    String modelName;
    @Unit("km/h") double maxVelocity;
    @Unit("kg") double mass;
    @Unit("cm") double length;
    @Unit("kW") double power;
}

class CarIm {
    String modelName;
    @Unit("mph") float maxVelocity;
    @Unit("lbs") float mass;
    @Unit("ft") float length;
    @Unit("hp(M)") float power;
}

Adapter si2imAdapter = Bindings.getTypeAdapter(
    Bindings.getBinding(CarSI.class),
    Bindings.getBinding(CarIm.class) );
```

Primitive Types. Note, primitive adapter throws an exception at runtime if values are not adaptable.

```
Adapter adapter = getTypeAdapter( integerBinding, doubleBinding );
Double double = adapter.adapt( new Integer( 5 ) );
```

Records are matched by field names.

```
class Foo {
    int x, y, z;
}
class Bar {
    int z, y, x;
}
Adapter adapter = getTypeAdapter( fooBinding, barBinding );
```

Subtype to supertype: Note, this conversion is not symmetric, supertypes cannot be converted to subtypes.

```
class Node {
    String id;
}
class ValueNode extends Node {
    Object value;
```

```
    }
    Adapter adapter = getTypeAdapter( valueNodeBinding, nodeBinding );
```

Non-existing fields to Optional fields

```
    class Node {
        String id;
    }
    class NominalNode extends Node {
        String id;
        @Optional String name;
    }
    Adapter adapter = getTypeAdapter( nodeBinding, nominalNodeBinding );
```

Enumerations

```
enum Cars { Audio, BMW, Mercedes, Honda, Mazda, Toyota, Ford,
Mitsubishi, Nissan, GM }
    enum JapaneseCars { Honda, Mazda, Toyota, Nissan, Mitsubishi }

    Binding carsBinding = Bindings.getBinding( Cars.class );
Binding japaneseCarsBinding = Bindings.getBinding( JapaneseCars.class
);
Adapter adapter = Bindings.adapterCache.getAdapter(japaneseCarsBinding,
carsBinding, true, false);
```

Accessors

Accessor is an interface for partial reading, writing and monitoring of a arbitrary data container. It is a model abstraction. Whether a history, a real-time simulation experiment, the data model is presented in terms of Databoard type system.

org.simantics.databoard.accessor interfaces.

Class	**Description**
DataAccessor	Base class for all data Accessors
RecordAccessor	Record
ArrayAccessor	Array - an ordered sequence of elements of one value.
MapAccessor	Map - an *ordered* map of keys to values.
UnionAccessor	Union
BooleanAccessor,IntAccessor,LongAccessor,FloatAccessor,DoubleAccessor	Primitive and numeric Accessors
StringAccessor	String
OptionalAccessor	Optional value
VariantAccessor	Variant value

Accessors can be opened to a sub-nodes with AccessorReference or by calling getAccessor. AccessorReference is a string of instances, either accessor type specific of

LabelReferences.

```
AccessorReference ref = AccessorReference.compile(
    new FieldNameReference("node"),
    new VariantValueReference()
);
Accessor child = accessor.getAccessor( ref );

AccessorReference ref = AccessorReference.compile(
    new LabelReference("node"),
    new LabelReference("v")
);
Accessor child = accessor.getAccessor( ref );

AccessorReference ref = AccessorReference.create("n-node/v");
Accessor child = accessor.getAccessor( ref );

AccessorReference ref = AccessorReference.create("node/v");
Accessor child = accessor.getAccessor( ref );

VariantAccessor va = recordAccessor.getFieldAccessor("node");
Accessor child = va.getValueAccessor();
```

Listening mechanism

Accessor offers a monitoring mechanism for the data model. There is an `InterestSet` that is a description of a sub-tree that is to be monitored of the data model. `Events` are objects that spawned on changes to the data model. Each event object is annotated with reference path that is in relation to the node where the listener was placed.

Implementations

Accessors is a facade class that contains utilities for instantiating and handling Accessors.

`Binary Accessor` is an Accessor implementation that stores values in a Databoard binary format. The back-end can be bound to file or memory (`byte[]` or `ByteBuffer`).

To create a new file based store and acquire an accessor to it, use:

```
    RecordType type = ...
    FileRecordAccessor file = Accessors.createFile( file, type );
```

Open an accessor to a binary file, use:

```
    FileVariantAccessor fa = Accessors.openFile( file );
    FileRecordAccessor ra = fa.getValueAccessor();
```

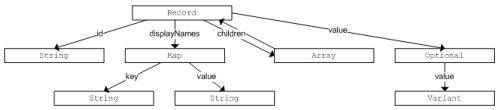
`Java Accessor` is an Accessor implementation that Java Instances as Databoard Data Model. The implementation requires a Binding. To access a Java Object as Accessor, use:

```
    RecordAccessor ra = Accessors.getAccessor( binding, instance );
```

Example

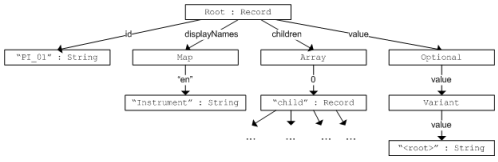
There is a data system with the following structure. The structure is presented with Databoard type notation (the text and tree representations below).

```
  type Node = {
         id : String;
         displayNames : LocalizedTexts;
         children : Node[];
         value : Optional(Variant);
       }
```



An instance of Node resides in the data system in proprietary format. It is accessible with an Databoard accessor. The tree and the type structure follows the given type representation (text and tree representations below).

```
  root : Node = {
id = “PI_01”
displayNames = map{ “en” = “Instrument “
}
            children =
             [
{id=”Child”,
displayNames = map{ “en” = “Child”} },
                value = 5 : Integer
              }
             ]
value = “<root>”
: String
          }
```



Utilities

- DataTypes is a facade class that has functions for handling DataTypes.
- Bindings is a facade class that has functions for handling Bindings.
- Accessors is a facade class that has functions for handling Accessors.
- Units is a facade class that has functions for handling Engineering Units.
- Methods has Methods, Interfaces and RPC utility functions.
- RandomAccessBinary is a interface for byte handling operations. In addition to basic primitive reading & writing, there are methods for grow, shrink, insert and remove.
  - BinaryFile and BinaryMemory are corresponding file and memory implementations.

- Blob is an implementation that represents a sub-region of a RandomAccessBinary.

File

There is a DataType based List<?> implementation FileList.java. FileList is a random access List. It supports both variable and fixed length data types. The files that contain variable length data types are read through and indexed when the file is opened. There is no header in the file format, the header must be known before hand to the user of the class.

```
FileList<String> file = new FileList<String>(
"Example.tmp", String.class );
    file.add("Hello");
    file.close();
```

Interface Types

There are interfaces, method types and method type definitions. Interface type describes a software interface. It is a collection of methods type definitions. Method type is an unnamed function with the following properties : Response Type, Request Type and ErrorType; where Response Type is any Data Type, Request Type is a Record and Error Type is an Union. Method type definition is nominal method description.

The respective Java classes are:

- Interface.java
- MethodTypeDefinition.java
- MethodType.java

In java InterfaceType description can be created with one of the following methods:

- Implementing InterfaceType
- Reading an Java Interface Class using reflection

```
    Interface it = new Interface( ... methodDefinitions );
    Interface it = getInterface( MyInterface.class );
```

MethodInterface.java is a binding of an Java Instance and an Interface Type. It decouples the method invocation from the object.

MethodInterface can be created with the following methods:

- Implementation
- Reflection

```
    MethodInterface mi   = new MethodInterface() {...}
MethodInterface mi = DataTypes.bindInterface( MyInterface.class,
myObject );
```

Utilities `DataTypes.createProxy()` and `DataTypes.bindInterface()` adapt between MethodInterface and Java Instance.

```
MethodInterface mi = DataTypes.bindInterface( MyInterface.class,
myObject );
MyInterface myObject = DataTypes.createProxy( MyInterface.class, mi );
```

Remote Procedure Call

Utilities Server.java and Client.java put MethodInterface over TCP Socket.

```
    Server myServer      = new Server(8192, mi);
    MethodInterface mi   = new Client("localhost", 8192);
```

MethodInterface with Server and Client together forms a Remote Procedure Call (RPC) mechanism.

```
    [Server]
MethodInterface mi = Methods.bindInterface( MyInterface.class, myObject
);
    Server myServer      = new Server(8192, mi);

    [Client]
    MethodInterface mi   = new Client("localhost", 8192);
MyInterface myObject = Methods.createProxy( MyInterface.class, mi );
```

External links

[1] http://en.wikipedia.org/wiki/Regular_expression
[2] http://en.wikipedia.org/wiki/Mime_type

Source: https://www.simantics.org/wiki/index.php?title=Databoard_dev

Principal Authors: Toni Kalajainen