Databoard Developer Manual

Databoard 0.6.0 Developer Manual

Contents

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 = Datatypes.DOUBLE;
Datatype type = Datatypes.getDatatype( Double.class );

Datatypes.addDefinition("type Node = { id : String; children : Node[] }");
Datatype type = Datatypes.getDatatype("Node");
```

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

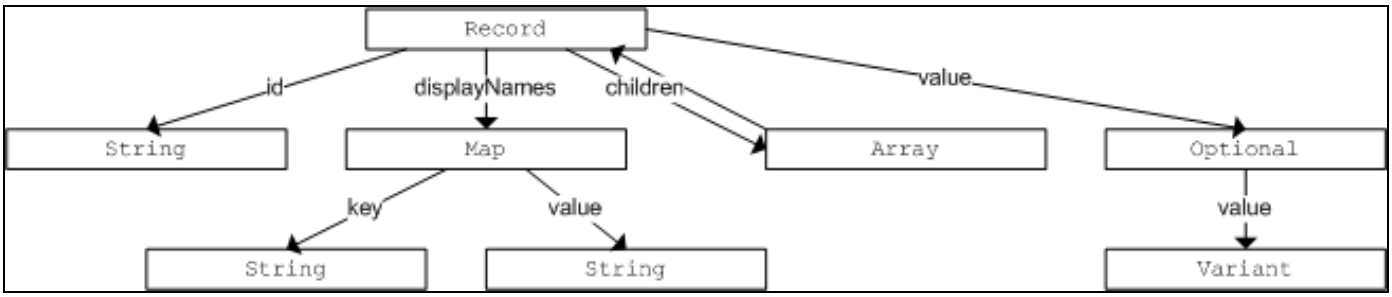
```
String type = type.toString();

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

Structure Example

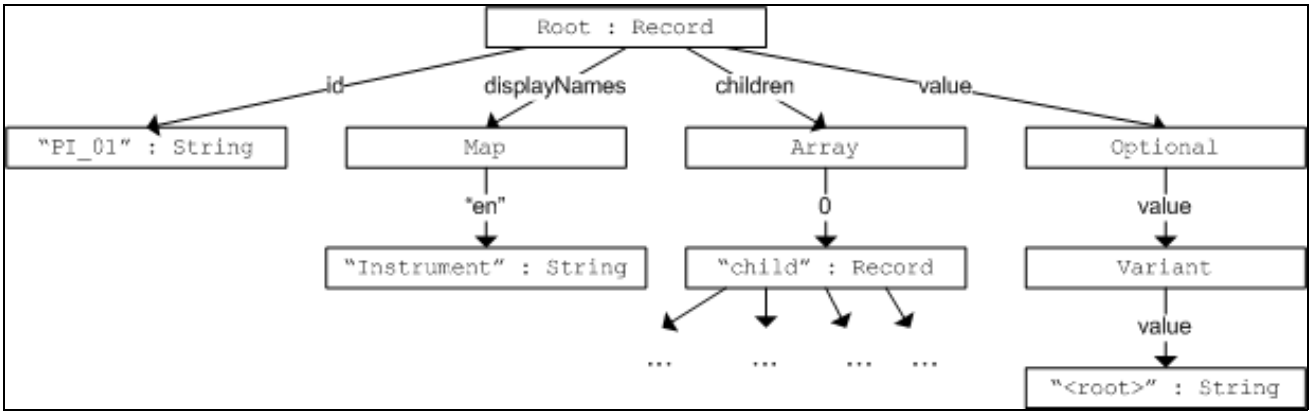
A node is a recursive type. With databoard typesystem it could be stated as

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



A couple of instances with Databoard value notation:

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



Binding

There is a type system, and when developing with java, platform neutral data values can be read from and written to objects. This is the role of a binding, a map from a Java Class to a Datatype.

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. To acquire a binding, the developer can use a utility that creates one using Java reflection functions.

```
Binding binding = Binding.getBinding( Double.class );
```

Sometimes classes cannot bound using automatic tool, for instance when using 3rd party classes which cannot be modified. The developer must then write binding self by sub-classing on of the 13 base binding classes (There is a base binding class for each Datatype).

```
Binding binding = new RecordBinding() { ... };
```

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 = Bindings.DOUBLE;
Binding binding = Binding.getBinding( Double.class );
Binding binding = Binding.getBinding( Datatypes.DOUBLE );
```

Reflection

Data Type and Binding can be read automatically from a Class by utility.

```
Datatype type = Datatypes.getDatatype( Foo.class );
Binding binding = Bindings.getBinding( Foo.class );
```

Bindings for generics classes can be created by passing arguments.

```
Binding e = Bindings.getBinding(List.class, String.class);
```

```
List<String> list = (List<String>) e.createRandom(5);

Binding binding = Bindings.getBinding( Map.class, Integer.class, Integer.class );
Map<Integer, Integer> value = (Map<Integer, Integer>) binding.createDefault();
```

Even cascading generics...

```
Binding e = Bindings.getBinding(List.class, List.class, String.class);
List<List<String>> listList = (List<List<String>>) e.createRandom(5);
```

Classes are RecordTypes

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

Is a binding to the following Datatype

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

There are three types of classes supported, and therefore three ways how objects are constructed. If you create binding for your class with Bindings#getBinding(clazz), the class must adhere one of these format. You may have to add annotations such as @Recursive, @Optional, @Arguments.

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 }
```

is interpreted as union type

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

If you cannot modify the class, you have to create binding for it by subclassing base binding classes, eg. RecordBinding.

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)

```
@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;
```

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.

`DefaultBindingScheme` is a scheme that converts any datatype to a binding. It prefers java.lang.X primitives. The Class mapping for each type is listed below.

Type	Class
BooleanType	Boolean.class
ByteType	Byte.class
FloatType	Float.class
DoubleType	eDouble.class
IntegerType	Integer.class
LongType	Long.class
StringType	String.class
UnionType	TaggedObject.class
OptionType	ValueContainer.class
RecordType	Object[].class
ArrayType	ArrayList.class
Array(Byte)	byte[].class
MapType	TreeMap.class
VariantType	Variant.class

`MutableBindingScheme` 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	TaggedObject.class
OptionType	ValueContainer.class
RecordType	Object[].class
ArrayType	ArrayList.class
MapType	TreeMap.class
VariantType	Variant.class

Serialization

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

```
Binding binding = Bindings.DOUBLE;
Serializer serializer = Bindings.getSerializer( binding );
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 Specification.
- `Binding#createDefault()` creates a valid instance of the Datatype.
- `Binding#createRandom(int)` creates a valid instance with random values. Useful for unit tests.
- `Binding#clone(Object)` creates a new instance with same content.
- `Binding#readFrom(Object, Binding, Binding)` copies contents from another object of same type.

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`.

Adapter can be created automatically or implemented self.

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

Example:

```
Adapter adapter = Bindings.getAdapter(Bindings.MUTABLE_DOUBLE, Bindings.DOUBLE);
java.lang.Double double = adapter.adapt( new MutableDouble(5.0) );
```

There is also convenience.

```
java.lang.Double double = Bindings.adapt( new MutableDouble(5.0), Bindings.MUTABLE_DOUBLE, Bindings.DOUBLE );
```

The argument given to `Adapter#adapt(Object)` may be re-used in the result unless the adapter is a cloning adapter which guarantees a clone. Note, even wih cloning adapters immutable classes, (eg java.lang.Integer) are never cloned.

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

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) );

CarIm americanCarInfo = si2imAdapter.adapt( europeanCarInfo );
```

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 cannot be 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 {
    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

Say, you have several gigabytes of data in a file. The whole object doesn't need to be serialized at once. You can read and write the value partially using Accessor interface. The actual container can be a file, memory byte[]/ByteBuffer or a Java Object. The content is structured as tree using Databoard's type system. All but referable records are supported (=no recursion in accessors).

org.simantics.databoard.accessor interfaces.

Class	Description
Accessor	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 and Files are facade classes that contains utilities for instantiating and handling Accessors.

`Binary Accessor` is an access to a value in binary format (`byte[]` or `ByteBuffer`).

Example: Binary accessor

```
Datatype type = Datatypes.getDatatype( Rectangle2D.Double.class );
Binding binding = Bindings.getBinding( Rectangle2D.Double.class );
Serializer s = Binding.getSerializer( binding );

// Serialize rectangle
Rectangle2D rect = new Rectangle2D.Double(0,0, 10, 10);
byte[] data = s.serialize(rect);

// Open accessor to byte data and modify first field in the byte data
RecordAccessor ra = Accessors.getAccessor(data, type);
ra.setFieldValue(0, Bindings.DOUBLE, 5.0);

// Deserialize values from the byte data back to the rectangle object
s.deserialize(data, rect);
System.out.println(rect.getX());
```

Example: File accessor, create

```
RecordType type = Datatypes.getDatatype( Rectangle2D.Double.class );
// Create a new file and initialize it with rectangle type, and open file accessor
FileRecordAccessor fa = Accessors.createFile( file, type );

// Write the first field (x)
fa.setFieldValue(0, Bindings.DOUBLE, 5.0);
fa.close();
```

Example: File accessor, open

```
// Open an accessor to an existing binary file
FileVariantAccessor fa = Accessors.openAccessor(file);
RecordAccessor ra = fa.getContentAccessor();

// Read the first field (x)
Double x = (Double) ra.getFieldValue(0, Bindings.DOUBLE);
fa.close();
```

Example: Java Accessor

```
Binding binding = Bindings.getBinding(Rectangle2D.Double.class);
Rectangle2D rect = new Rectangle2D.Double(0,0, 10, 10);

// Open accessor to rectangle
RecordAccessor ra = Accessors.getAccessor(binding, rect);

// Set rectangle's first field (x) to 5.0
ra.setFieldValue(0, Bindings.DOUBLE, 5.0);
System.out.println( rect.getX() );
```

Accessor Reference

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.

```
ChildReference ref = ChildReference.compile(
  new NameReference("node"),
  new ComponentReference()
);
Accessor child = accessor.getComponent( ref );

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

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

ChildReference ref = ChildReference.create("node/v");
Accessor child = accessor.getComponent( 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.

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.
- Files has Read, Write and accessor functions.
- 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.

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.

```
public interface MyInterface { String helloWorld(String msg); }

[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 );
```