Java Developers Guide

rasdaman version 9.6

rasdaman Version 9.6 Java Developers Guide

Preface

Overview

This guide provides information about how to use the rasdaman database management system. The booklet explains usage of rasj, the rasdaman Java API.

Follow the instructions in this guide as you develop your application which makes use of rasdaman services. Explanations detail how, from within a Java program, to create databases, collections, and instances; how to retrieve from databases; how to manipulate and delete instances within databases; how to influence physical storage parameters; how to do transaction handling and other administrative tasks.

Audience

The information in this manual is intended for application developers.

Rasdaman Documentation Set

This manual should be read in conjunction with the complete rasdaman documentation set which this guide is part of. The documentation set in its completeness covers all important information needed to work with the rasdaman system, such as programming and query access to databases, guidance to utilities such as *raswct*, release notes, and additional information on the rasdaman wiki.

The rasdaman Documentation Set consists of the following documents:

- Installation and Administration Guide
- Query Language Guide
- C++ Developer's Guide
- Java Developer's Guide
- raswct Developer's Guide
- the rasdaman wiki, accessible at www.rasdaman.org

Table of Contents

1 Introduction

1.1 Multidimensional Data

In principle, any natural phenomenon becomes spatio-temporal array data of some specific dimensionality once it is sampled and quantised for storage and manipulation in a computer system; additionally, a variety of artificial sources such as simulators, image renderers, and data warehouse population tools generate array data. The common characteristic they all share is that a large set of large multidimensional arrays has to be maintained. We call such arrays *multidimensional discrete data* (or short: *MDD*), expressing the variety of dimensions and separating them from the conceptually different multidimensional vectorial data appearing in geo databases.

rasdaman is a domain-independent database management system (DBMS) which supports multidimensional arrays of any size and dimension and over freely definable cell types. Versatile interfaces allow rapid application deployment while a set of cutting-edge intelligent

optimization techniques in the rasdaman server ensures fast, efficient access to large data sets, particularly in networked environments.

1.2 rasdaman Overall Architecture

The rasdaman client/server DBMS has been designed using internationally approved standards wherever possible. The system follows a two-tier client/server architecture with query processing completely done in the server. Internally and invisible to the application, arrays are decomposed into smaller units which are maintained in a conventional DBMS, for our purposes called the *base DBMS*.

On the other hand, the base DBMS usually will hold alphanumeric data (such as metadata) besides the array data. rasdaman offers means to establish references between arrays and alphanumeric data in both directions.

Hence, all multidimensional data go into the same physical database as the alphanumeric data, thereby considerably easing database maintenance (consistency, backup, etc.).

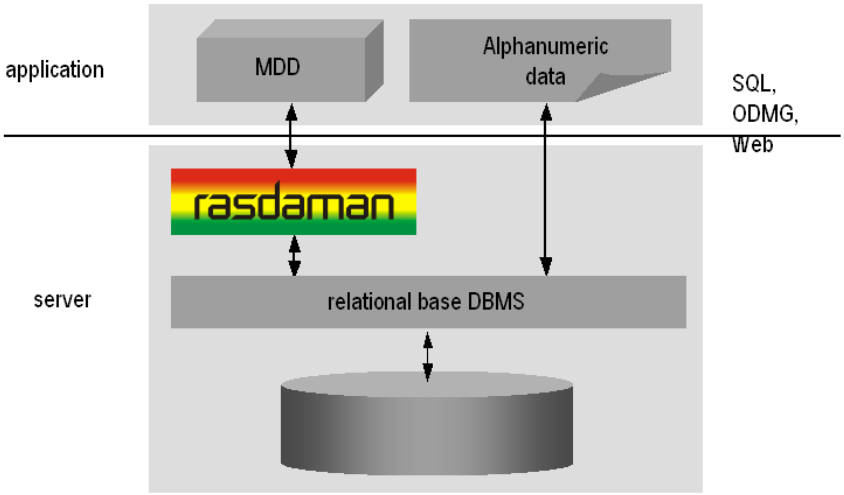

Figure 1 Embedding of rasdaman in IT infrastructure

Further information on this topic is available in the other components of the rasdaman documentation set.

1.3 Further Reading

n.n.: *rasdaman Query Language Guide*, rasdaman GmbH.

S.J. Cannan: *SQL The Standard Handbook*, McGraw-Hill Book Company, London, 1993.

R.G.G. Cattell, Douglas K. Barry: *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, California, 1999.

2 Terminology

This section gives an overview of the concepts underlying rasdaman and raster databases. For details on the operational semantics of the model the reader is strongly encouraged to study the *rasdaman Query Language Guide*.

2.1 An Intuitive Definition

An array is a set of elements which are ordered in space. The space considered here is discretized, i.e., only integer coordinates are admitted. The number of integers needed to identify a particular position in this space is called the *dimension* (sometimes also referred to as *dimensionality*). Each array element, which is referred to as *cell*, is positioned in space through its *coordinates*.

A cell can contain a single value (such as an intensity value in case of grayscale images) or a composite value (such as integer triples for the red, green, and blue component of a color image). All cells share the

same structure which is referred to as the *array cell type* or *array base type*.

Implicitly a neighborhood is defined among cells through their coordinates: incrementing or decrementing any component of a coordinate will lead to another point in space. However, not all points of this (infinite) space will actually house a cell. For each dimension, there is a *lower* and *upper bound*, and only within these limits array cells are allowed; we call this area the *spatial domain* of an array. In the end, arrays look like multidimensional rectangles with limits parallel to the coordinate axes. The database developer defines both spatial domain and cell type in the *array type definition*. Not all bounds have to be fixed during type definition time, though: It is possible to leave bounds open so that the array can dynamically grow and shrink over its lifetime.

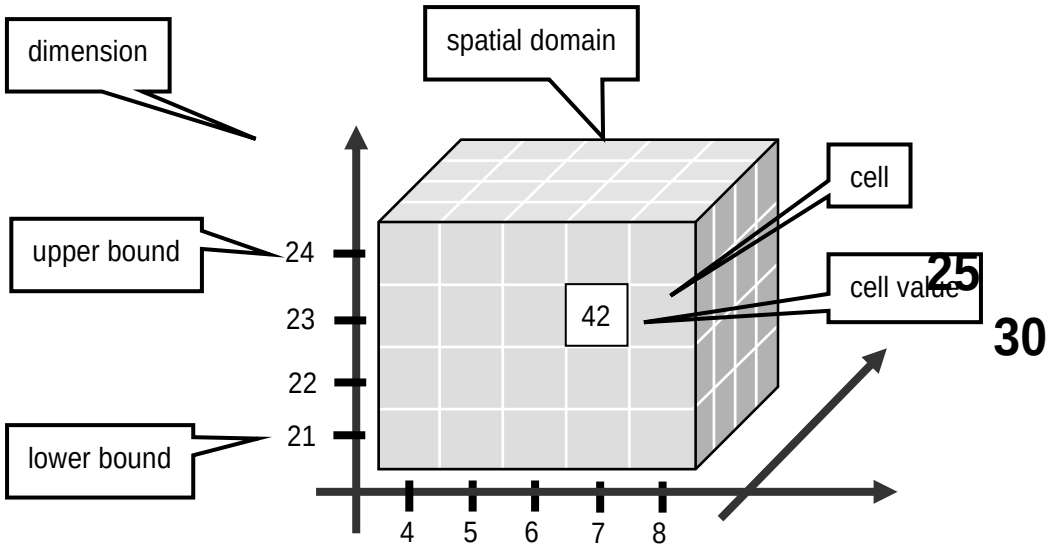

Figure 2 Constituents of an array

Synonyms for the term array are *multidimensional arrays*, *multidimensional data*, *MDD*. They are used interchangeably in the rasdaman documentation.

In rasdaman databases, arrays are grouped into collections. All elements of a collection share the same array type definition Collections form the basis for array handling, just as tables do in relational database technology.

2.2 A Technical Definition

Programmers who are familiar with the concept of arrays in programming languages maybe prefer this more technical definition:

An array is a mapping from integer coordinates, the spatial domain, to some data type, the cell type. An array's spatial domain, which is always finite, is described by a pair of lower bounds and upper bounds for each dimension, resp. Arrays, therefore, always cover a finite, axis-parallel subset of Euclidean space.

Cell types can be any of the base types and composite types defined in the ODMG standard.

In rasdaman, arrays are strictly typed wrt. spatial domain and cell type. Type checking is done at query evaluation time. Type checking can be disabled selectively for an arbitrary number of lower and upper bounds of an array, thereby allowing for arrays whose spatial domains vary over the array lifetime.

3 Application Examples

3.1 Overview

This section contains an example of using the rasdaman Java API. The intention is, for the advanced programmer, to quickly get an overview on the programming style to be observed.

The source code can be found (slightly extended) in subdirectory `examples/java` of the rasdaman distribution directory.

For details on the operational semantics of the rasdaman data model the reader is strongly encouraged to study the *rasdaman Query Language Guide*.

3.2 Application Program Example Code

```
// import all packages needed; the first two come with rasj
import rasj.*;
import org.odmg.*;
import java.util.*;

public class AvgCell
{
  public static void main(String[] args)
  {
    string server, base, coll;
    for (int i=args.length-1; i>=0; i--)
    { // evaluate command line arguments
      if (args[i].equals("-s"))
        server = args[i+1];         // set server:port
      if (args[i].equals("-d"))
        base = args[i+1];           // set database name
      if (args[i].equals("-c"))
        coll = args[i+1];           // set collection name
    }

    try // watch out for possible exceptions thrown by rasj
    { // instantiate ODMG implementation
      Implementation myApp = new RasImplementation(
                                        "http://"+ server );

      // create database object, open database
      Database myDb = myApp.newDatabase();
      myDb.open( base, Database.OPEN_READ_ONLY );

      // create transaction object, begin transaction
      Transaction myTa = myApp.newTransaction();
      myTa.begin();

      // instantiate query object, submit it to server
      OQLQuery myQu = myApp.newOQLQuery();
      myQu.create( "select mr from mr" );
      DBag resultSet = (DBag) myQu.execute();

      // process query result returned from server
      if (resultSet != null)
      { // define iterator over result set
        Iterator iter = resultSet.iterator();
        while (iter.hasNext())
        { // process next element in line
          RasGMArray result = (RasGMArray) iter.next();

          // access the array, sum up its values
          byte[] pixelfield = result.getArray();
          double sum = 0.0;
          for(int i=0; i<result.getArraySize(); i++)
            sum += pixelfield[i];
```

```
            System.out.println( "Average over " + size +
              " pixels is " +
              ( ( sum / result.getArraySize() ) + 128 ) );
          }
        }

        // all done, so commit transaction and close database
        myTa.commit();
        myDb.close();
      }
      catch (org.odmg.ODMGException e)
      { // on error, print message and try forced detaching
        System.out.println( e.getMessage() );
        if(myTa != null)
          myTa.abort();
        if(myDb != null)
          myDb.close();
      }
      System.out.println( "Done." );
    }
}
```

Note

This sample program makes use of the `mr` collection provided with the rasdaman distribution package. See the rasdaman *Installation and Administration Guide* to learn on how to create this collection as part of the demonstration database.

4 rasj

4.1 Overview

The rasj package contains the API for Java-based access to the rasdaman database system. It relies on the ODMG standard (see Section 1.3) which it implements to the extent that is necessary for raster data management.

The overall rasj package is subdivided into two packages, `rasj` and `org.odmg`. The `org.odmg` sub-package (see Section 5) implements the general ODMG specifications while the `rasj` sub-package implements rasdaman specific features.

4.2 Class Hierarchy

The `rasj` class hierarchy has the following structure.

```
class java.lang.Object
  |
  +--class rasj.RasImplementation
  |    | (implements org.odmg.Implementation)
  |    |
  |    +--class rasj.RasODMGInterface
  |
  +--class RasPoint
  |
  +--class RasSInterval
  |
  +--class RasMInterval
  |
  +--class rasj.odmg.RasObject
  |    |  (implements rasj.RasGlobalDefs)
  |    |
  |    +--class rasj.RasGMArray
  |         | (implements rasj.RasGlobalDefs)
  |         |
  |         +--class rasj.RasMArrayByte
  |         |  (implements rasj.RasGlobalDefs)
  |         |
  |         +--class rasj.RasMArrayDouble
  |         |  (implements rasj.RasGlobalDefs)
  |         |
  |         +--class rasj.RasMArrayFloat
  |         |  (implements rasj.RasGlobalDefs)
  |         |
  |         +--class rasj.RasMArrayInteger
  |         |  (implements rasj.RasGlobalDefs)
  |         |
  |         +--class rasj.RasMArrayLong
  |         |  (implements rasj.RasGlobalDefs)
  |         |
  |         +--class rasj.RasMArrayShort
  |            (implements rasj.RasGlobalDefs)
  |
  +--class rasj.RasStorageLayout
  |
  +--class rasj.RasType
  |    | (implements rasj.RasGlobalDefs)
  |    |
  |    +--class rasj.RasBaseType
  |    |    |
  |    |    +--class rasj.RasPrimitiveType
  |    |    |
  |    |    +--class rasj.RasStructureType
  |    |
  |    +--class rasj.RasCollectionType
  |    |
  |    +--class rasj.RasMArrayType
  |    |
  |    +--class rasj.RasMIntervalType
  |    |
  |    +--class rasj.RasOIDType
  |    |
  |    +--class rasj.RasPointType
  |    |
  |    +--class rasj.RasSIntervalType
```

```
|
+--class java.lang.Throwable
     | (implements java.io.Serializable)
     |
     +--class java.lang.Exception
     |    |
     |    +--class org.odmg.ODMGException
     |    |    |
     |    |    +--class org.odmg.QueryException
     |    |         |
     |    |         +--class org.odmg.QueryInvalid-
     |    |              |                        Exception
     |    |              +--class rasj.RasQueryExecution-
     |    |                                  FailedException
     |    |
     |    +--class rasj.RasException
     |         | (implements rasj.RasGlobalExceptionDefs)
     |         |
     |         +--class rasj.RasDimensionMismatchException
     |         |  (implements rasj.RasGlobalExceptionDefs)
     |         |
     |         +--class rasj.RasIndexOutOfBoundsException
     |         |  (implements rasj.RasGlobalExceptionDefs)
     |         |
     |         +--class rasj.RasResultIsNoCellException
     |         |  (implements rasj.RasGlobalExceptionDefs)
     |         |
     |         +--class rasj.RasResultIsNoInterval-
     |         |                                  Exception
     |         |  (implements rasj.RasGlobalExceptionDefs)
     |         |
     |         +--class rasj.RasStreamInputOverflow-
     |         |                                  Exception
     |         |  (implements rasj.RasGlobalExceptionDefs)
     |         |
     |         +--class rasj.RasTypeInvalidException
     |            (implements rasj.RasGlobalExceptionDefs)
     |
     +--class java.lang.RuntimeException
          |
          +--class org.odmg.ODMGRuntimeException
          |    |
          |    +--class rasj.RasConnectionFailedException
          |
          +--class rasj.RasRuntimeException
               |
               +--class rasj.RasClientInternalException
               |
               +--class rasj.RasTypeNotSupportedException
               |
               +--class rasj.RasTypeUnknownException
```

4.3 Interface Hierarchy

The complete `rasj` interface hierarchy has the following structure.

```
interface rasj.RasGlobalDefs
interface rasj.RasGlobalExceptionDefs
```

5 ODMG

5.1 Overview

The ODMG classes implement classes defined in the ODMG standard providing functionality such as database open and close, transactions, querying, and unique identifiers, i.e., OIDs.

Don't Use `DArray` !

ODMG defines an interface `DArray` which also is part of the ODMG sub-package provided with the rasdaman distribution. These implement only 1-D arrays; most important, however, `DArray` is ***not compatible*** with rasdaman arrays. Therefore, ***do not use*** class `DArray` as a rasdaman array, but use class `RasGMArray` (and its subclasses) instead.

…But Do Use `Dbag` !

Queries return multi-sets as results. A *bag* or *multi-set* contains an arbitrary number of elements; like a set (and unlike a list), no particular sequence is defined, and like a list (and unlike a set), the same elements

can occur multiply. The query result type, therefore, is `DBag`. See also Section 8.

5.2 Class Hierarchy

The complete `org.odmg` class hierarchy has the following structure.

```
class java.lang.Exception
  |
  +--class org.odmg.ODMGException
  |    |
  |    +--class org.odmg.DatabaseNotFoundException
  |    |
  |    +--class org.odmg.DatabaseOpenException
  |    |
  |    +--class org.odmg.ObjectNameNotFoundException
  |    |
  |    +--class org.odmg.ObjectNameNotUniqueException
  |    |
  |    +--class org.odmg.QueryException
  |         |
  |         +-- class org.odmg.QueryInvalidException
  |         |
  |         +--class org.odmg.QueryParameterCountInvalid-
  |         |                                        Exception
  |         |
  |         +--class org.odmg.QueryParameterTypeInvalid-
  |                                                  Exception
  |
  +--class java.lang.RuntimeException
       |
       +--class org.odmg.ODMGRuntimeException
            |
            +--class org.odmg.ClassNotPersistenceCapable-
            |                                        Exception
            |
            +--class org.odmg.DatabaseClosedException
            |
            +--class org.odmg.DatabaseIsReadOnlyException
            |
            +--class org.odmg.LockNotGrantedException
            |
            +--class org.odmg.NotImplementedException
            |
            +--class org.odmg.ObjectDeletedException
            |
            +--class org.odmg.ObjectNotPersistentException
            |
            +--class org.odmg.TransactionAbortedException
            |
            +--class org.odmg.TransactionInProgressException
            |
            +--class org.odmg.TransactionNotIn-
                                              ProgressException
```

5.3 Interface Hierarchy

This is the `org.odmg` interface hierarchy:

```
interface java.util.Collection
  |
  +--interface org.odmg.DCollection
  |    |
  |    +--interface org.odmg.Darray
  |    |  (also extends java.util.List)
  |    |
  |    +--interface org.odmg.DBag
  |    |
  |    +--interface org.odmg.DList
  |    |  (also extends java.util.List)
  |    |
  |    +--interface org.odmg.DSet
  |       (also extends java.util.Set)
  |
  +--interface java.util.List
  |   |
  |   +--interface org.odmg.DArray
  |   |  (also extends org.odmg.DCollection)
  |   |
  |   +--interface org.odmg.DList
  |      (also extends org.odmg.DCollection)
  |
  +--interface java.util.Set
  |    |
  |    +--interface org.odmg.DSet
  |       (also extends org.odmg.DCollection)
  |
  +--interface org.odmg.Database
  |
  +--interface org.odmg.Implementation
  |
  +--interface java.util.Map
  |    |
  |    +--interface org.odmg.Dmap
  |
  +--interface org.odmg.OQLQuery
  |
  +--interface org.odmg.Transaction
```

5.4 How To Use

The following code piece demonstrates a typical retrieval situation: a database is opened, a transaction is started, and then a query is executed against that database.

```
Transaction myTa = null;
Database myDb = null;
OQLQuery myQu = null;
DBag resultSet = null;
RasGMArray result = null;

Implementation myApp = new RasImplementation(
                                  "http://" + server + port );
```

```
myDb = myApp.newDatabase();
myDb.open( database, Database.OPEN_READ_ONLY );

myTa = myApp.newTransaction();
myTa.begin();

myQu = myApp.newOQLQuery();
myQu.create( "select mr from mr" );
resultSet = (DBag) myQu.execute();

// ...result set processing...

myTa.commit();
myDb.close();
```

Database Login

The database name and the address of a running server manager must be indicated. Further optional parameters and their defaults are:

- login (default: "`rasguest`")
- password (default: "`rasguest`")

Multiple ODMG Implementations

It is well possible to use several implementations – for example, from different vendors – of the ODMG classes simultaneously. Like rasj, other ODMG packages will provide an `Implementation` class in their `org.odmg` package. Instantiating one `Implementation` for each package is the only prerequisite to be done. The resulting code might look like the following (incomplete) example fragment where two different implementation classes are assumed, `RasImplementation` and `Implementation2`; note that transactions for different implementations are independent from each other.

```
Transaction myTa1 = null;
Database myDb1 = null;

Transaction myTa2 = null;
Database myDb2 = null;

Implementation rasApp1 = new RasImplementation(
                         "http://" + server1 + ":" + port1 );
myDb1 = myApp1.newDatabase();
myDb1.open( rasbase, Database.OPEN_READ_ONLY );
MyTa1 = myApp1.newTransaction();
myTa1.begin();

Implementation2 myApp2 = new Implementation2(
                         "http://" + server2 + ":" + port2 );
myDb2 = myApp2.newDatabase();
myDb2.open( database2, Database.OPEN_READ_ONLY );
MyTa2 = myApp2.newTransaction();
myTa2.begin();

// ...now access both databases...
```

```
myTa1.commit();
myDb1.close();

myTa2.commit();
myDb2.close();
```

ODMG Functions Available

rasj does not implement ODMG fully (this would go beyond its purpose), rather it contains those functions necessary for rasdaman database access. When using the HTML hypertext documentation, clicking through the org.odmg package ultimately gets you to the rasdaman classes which implement the corresponding ODMG class. There, methods not available are marked as such.

Further Information

Details on how to process the query result can be found in Section 8. The example code makes use of the demonstration database whose set-up routines are part of the distribution package; find more on this topic in the rasdaman *Installation and Administration Guide*.

6 Points and Intervals

6.1 Overview

Point and interval handling is needed for indexing arrays, such as indication of array boundaries. To this end, classes `RasPoint`, `RasSInterval`, and `RasMInterval` for n-dimensional points, 1-D ("single-") intervals, and n-dimensional ("multi-") intervals resp. are provided.

Value Ranges and Consistency Constraints

All points, 1-D and n-D intervals can span negative values as well. Furthermore, intervals can have any integer value as lower bound. This is in contrast to most programming languages where usually the lower bound is fixed to 0.

However, intervals obviously need to match some consistency criteria to be valid. Foremostly, in a 1-D interval (class `RasSInterval`) as well as in

an n-D interval (class `RasMInterval`) the lower bound must not be higher than the upper bound.

Further, operations between intervals of any type must yield a valid interval again. Consider the union of two 1-D intervals `s1` and `s2`,

```
s1.unionWith( s2 )
```

Intervals `s1` and `s2` must be overlap or at least be adjacent, otherwise the resulting interval would contain a hole (mathematically speaking, it would not be simply connected). As such situations are not allowed for intervals in rasdaman, corresponding exceptions will be thrown by rasj.

If nevertheless two intervals should be merged which are apart from each other, then operation `closureWith()` can be used. It will "fill" the gap between the intervals so that a valid result interval comes out.

The HTML manual lists each possible situation. It is recommended to study this for getting an understanding of all valid and invalid interval combinations.

6.2 Class Hierarchy

```
class java.lang.Object
  |
  +--class RasPoint
  |
  +--class RasSInterval
  |
  +--class RasMInterval
```

Note

Class `java.lang.Object` obviously has further subclasses, not just the one shown here.

6.3 How To Use

Here are some sample code fragments showing usage of the point and interval classes:

RasPoint

```
// (1) point instantiation using string constructor:
RasPoint p1 = new RasPoint( "[ 3, 7 ]" );
// (2) point instantiation using numerical constructor:
RasPoint p2 = new RasPoint( 5, 0 );

// get point dimension:
int d = p2.dimension();

// test if points are equal:
boolean b = p1.equals( p2 );
```

RasSInterval

```
// create a 1-D intervals (100,200) and (-150,400), resp.:
RasSInterval s1 = new RasSInterval( 100, 200 );
RasSInterval s2 = new RasSInterval( "-150:400" );
                                    // no “[” and “]” !

// get upper bound of interval:
long hiBound = s2.high();
// get lower bound of interval:
long loBound = s2.low();

// get number of cells:
long noOfCells = m1.cellCount();

// test if interval intersects with another interval
// (the return value shows the kind of intersection)
int j = s1.intersectsWith( s2 );
```

RasMInterval

```
// create new 2-D interval, set bounds to (-1,1) and (3,7):
RasMInterval m1 = new RasMInterval( "[ -1:1, 3:7 ]" );
// create a 4-D interval, leaving open array bounds for now:
RasMInterval m2 = new RasMInterval( 4 );
```

7 Multidimensional Arrays

7.1 Overview

Instances of `RasGMArray` and its subclasses represent multidimensional arrays. To handle arrays with different base types and geometries, the "implements" relation of Java is used. With this approach, greyscale images, RGB images etc. can all be treated as subclasses of the general array class `RasGMArray`.

Currently supported are types for integer arrays (e.g., grayscale images) of various cell size, as well as types for floating-point arrays with single and double precision. All of them allow arrays of any dimension and extent per dimension.

Class Hierarchy

```
class rasj.odmg.RasObject
  |  (implements rasj.RasGlobalDefs)
  |
  +--class rasj.RasGMArray
```

```
|  (implements rasj.RasGlobalDefs)
|
+--class rasj.RasMArrayByte
|  (implements rasj.RasGlobalDefs)
|
+--class rasj.RasMArrayDouble
|  (implements rasj.RasGlobalDefs)
|
+--class rasj.RasMArrayFloat
|  (implements rasj.RasGlobalDefs)
|
+--class rasj.RasMArrayInteger
|  (implements rasj.RasGlobalDefs)
|
+--class rasj.RasMArrayLong
|  (implements rasj.RasGlobalDefs)
|
+--class rasj.RasMArrayShort
   (implements rasj.RasGlobalDefs)
```

7.2 How To Use

A few code fragments will show appropriate usage of the array classes. To keep it brief and to the spot, we omit declarations and other standard steps; these can be looked up in the previous, complete coding examples.

Note: Current restriction

Queries can contain formal parameters, denoted by `$1`, `$2`, etc. (see *Query Language Guide* for details). In the current rasj implementation, only one MDD object can be bound per query (however, it is possible to bind several scalar values). This limitation will be overcome in future releases.

Example 1: compute summary data from array

The following code example retrieves all MDD objects from a sample collection and, fore each object, computes the average cell value. As a safeguard, averaging is carried out only in case of integer cells (i.e., greyscale pixels).

```
myQu = myApp.newOQLQuery();
myQu.create( "select mr from mr" );
DBag resultSet = (DBag) myQu.execute();
if (resultSet != null)
{
  Iterator iter = resultSet.iterator();
  while (iter.hasNext())
  {
    result = (RasGMArray) iter.next();
    if(result.getTypeLength() != 1)
      System.out.println(
```

```
            "skipping image because of non-int cell type" );
    else
    {
      byte[] pixelfield = result.getArray();
      double sum = 0.0;
      long size = result.getArraySize();
      for(int i=0; i<size; i++)
        sum += pixelfield[i];
      System.out.println( "Average over " + size +
                          " pixels is " +
                          ((sum/size)+128) );
    }
  }
```

Example 2: set up array object in main memory

The following code fragment instantiates a `RasGMArray` object as a 2-D greyscale image and fills it with values using the normal Java means:

```
// create 2-D MDD with cell length 1, i.e., type "byte":
RasGMArray myMDD = new RasGMArray(
                    new RasMInterval( "[1:400,1:400]"), 1 );
// byte container for array data, matching in size:
byte[] mydata = new byte[160000];

// initialize array as all-black with two grey stripes:
for(int y=0; y<400; y++)
{
  for(int x=0; x<400; x++)
  {
    if((x>99 && x<151) || (x>299 && x<351))
      mydata[y*399+x]=100;
    else
      mydata[y*399+x]=0;
  }
}

// now insert byte array into MDD object
// (sets only the pointer, no copying takes place!):
myMDD.setArray(mydata);
```

As for the last line containing the import of array data into the MDD object, please observe the following: There are specific get/set functions for the various supported array types, e.g., `getIntArray()`. While the `setArray()` and `getArray()` methods always will work, they will require data type conversion if the actual array cell type is not "byte". Therefore, it is most efficient to always use that operation which respects the actual array data type.

The following code fragment instantiates a `RasGMArray` object as a 2-D greyscale image and fills it with values using the normal Java means:

Example 3: insert new array object into database

This example generates a new greyscale image collection named `test` in the database and inserts an image into this database collection.

Note that a new query object has to be generated for each query. It is not sufficient to just change the query string in the query object!

```
// set up query object for collection creation:
myQu.create( "create collection test GreySet" );
// set the object type name (used for server type checking):
myMDD.setObjectTypeName( "GreyImage" );
// finally, execute "create collection" statement:
myQu.execute();

// now create the insert statement:
myQu.create( "insert into test values $1" );
// let the server generate a new OID for the object to be
// inserted, and remember this OID locally:
myNewOID = myApp.getObjectId( myMDD );
// bind the MDD value which substitutes formal parameter $1:
myQu.bind( myMDD );
// …and ship the complete statement to the server:
myQu.execute();
```

7.3 rasdaman Cell Types

The set of cell base types known to rasdaman encompasses the usual numeric types. Below find the table of types known, and the necessary information to map them to Java types.

Null values, i.e., values of cells which have not been assigned a value yet, always are the numerical zero value of the corresponding type. This extends in the obvious way to composite cells.

RasDL	Length	Description
`octet`	8 bit	signed integer
`char`	8 bit	unsigned integer
`short`	16 bit	signed integer
`unsigned short`	16 bit	unsigned integer
`long`	32 bit	signed integer
`unsigned long`	32 bit	unsigned integer
`float`	32 bit	single precision floating point

`double`	64 bit	double precision floating point
`boolean`	1 bit[1]	true (nonzero value) false (zero value)

7.4 rasdaman Types vs. Java Types

Java types do not 1:1 correspond to rasdaman types. This is due to the fact that the Java type system in some aspects is different from what the ODMG Standard prescribes. Below find the most important caveats.

Long Integer

Long integer values in rasdaman always have 4 bytes, in accordance with the ODMG standard. The corresponding rasdaman types are `Ras_Long` and `Ras_ULong`.

In rasj, the array type to be used for 4-byte integers is `RasMArrayInteger` which matches with the Java `int` type occupying 4 bytes.

Mind that the Java type `long` represents 8 byte quantities. If an MDD object is passed to the database through rasj, a overflow test takes place on each integer value. An exception is thrown on overflow.

Unsigned Integers

Special care should be taken with unsigned integers, as Java does not support this. For example, for cells of type `Ras_UShort` (2 bytes) the array type `RasMArrayInteger` (4 bytes) must be used to collate values, according to the ODMG standard.

[1] memory usage is one byte per pixel

8 Storage Layout

8.1 Overview

At insertion time of an MDD object, several database-internal storage parameters can be set to affect the way the object is stored in the database. A `RasStorageLayout` object, attached to a `RasGMArray` MDD object, will guide storage of this MDD object when passed to the server through `RasOQLQuery.execute()`.

8.2 Class Hierarchy

```
class java.lang.Object
  |
  +--class rasj.odmg.RasStorageLayout
```

8.3 How To Use

The following code fragment shows how to associate a storage layout object with an MDD object; the storage layout will be evaluated at insertion time of the MDD into the database.

```
// create 2-D MDD with cell length 1, i.e., type byte:
RasGMArray myMDD =
    new RasGMArray(new RasMInterval( "[1:400,1:400]" ), 1 );

// assume that there is some byte array prepared, insert it:
myMDD.setArray( mydata );

// set image type name
// (see distribution file examples/rasdl/basictypes.dl):
myMDD.setObjectTypeName("GreyImage");

// add storage layout object:
RasStorageLayout myLayout = new RasStorageLayout();

// now you can set either TileSize or TileDomain; to this
// end, continue with Alternative 1 or 2, as described below
```

Alternative 1: set tile size

Having prepared the object as described above, now the tiling strategy can be set. Experience tells that a good size for tiles is between 128 and 256 kB, but bear in mind that the optimal size for tiles depends on the actual user behaviour as well as various system parameters.

```
// define size of tiles as 128,000 bytes:
myLayout.setTileSize( 128000 );
myMDD.setStorageLayout( myLayout );
```

Alternative 2: set domain shape

As an alternative to setting the overall tile size, the domain can be prescribed. This is more exact, as it allows to define not only size, but also the extent per dimension. For example, if it is known from the user access patterns there are ten times as much vertical slices requested than are horizontal ones, then it may be a good strategy to define tiles with a vertical:horizontal ratio of 10:1.

```
// define tiles with spatial extent [1:1000,1:100]:
myLayout.setTileDomain("[1:1000,1:100]");
myMDD.setStorageLayout( myLayout );
```

Note

rasdaman also allows to set the storage and compression format, as well as client/server transfer format. However, currently the interface controlling these parameters is only available via the C++ interface, not yet via Java. In future versions format and compression control will be available via Java, too.

9 Collections and Queries

9.1 Overview

Bag versus Set

Queries return multi-sets as results. The corresponding query result type is `DBag`.

A *bag* or *multi-set* is a collection of elements similar to sets an lists; like a set (and unlike a list), no particular sequence is defined, and like a list (and unlike a set), the same elements can occur multiply. While `{1,2,3}` is an example for a set, `[1,2,2,3]` is a bag example; `[1,2,3]` denotes the same bag as `[3,2,1]`, because sequence is irrelevant in a bag.

Let us clarify the difference with an example. A query which returns the object identifiers (OIDs) of some database objects, such as

```
select oid(a)
from a
```

never will contain duplicates, as OIDs are unique by definition On the other hand, requesting summary information on MDD objects may well lead to duplicates; for example, in a query like this:

```
select avg_cells(a)
from a
```

several objects may share the same maximum or average cell value. In the latter case, it obviously is crucial to obtain duplicates also. Therefore, the query result always is `DBag`, which forms a particular subclass of the general class `DCollection`.

Nevertheless, we will use the term result set sometimes, as it is just common database speak.

Important Hint

Use `org.odmg.DBag`, do *not* use `rasj.odmg.RasBag`!

9.2 Class Hierarchy

```
class java.util.AbstractCollection
  |  (implements java.util.Collection)
  |
  +--class rasj.odmg.RasCollection
     |  (implements org.odmg.DCollection)
     |
     +--class rasj.odmg.RasBag
     |  (implements org.odmg.DBag)
     |
     +--class rasj.odmg.RasSet
        (implements org.odmg.DSet)
```

9.3 How To Use

The following code piece demonstrates how to use object sets in the typical case of querying the database and piecewise processing the result set:

```
OQLQuery myQu = myApp.newOQLQuery();
myQu.create( "select mr from mr" );
DBag resultSet = (DBag) myQu.execute();
if (resultSet != null)
{
  Iterator iter = resultSet.iterator();
  while ( iter.hasNext() )
  {
    RasGMArray result = (RasGMArray) iter.next();
    // ...here now process result...
```

```
    }
}
```

Synchronous query execution

When a query is sent to the rasdaman server it will be executed in completeness – a running query cannot be aborted[2]. Care should be taken therefore not to start queries requiring resources beyond the capability of the server hardware and software environment, as the rasdaman service may be blocked for an indefinite time period.

9.4 Query Result Type

Database collections satisfy some criterion of homogeneity; this common property is expressed through the underlying type definition. Likewise, a collection returned as a query result has such an underlying common type definition. However, as queries dynamically describe and instantiate structures, this may not always adhere to some type existing in the database – sometimes the structure is new, so a type structure has to be generated "on the fly". While such a type does not have a name, its structure is well defined through the query itself.

This dynamic typing is predefined in the ODMG standard to which rasj adheres, so further information can be obtained there. See Section 1.3 for more information on ODMG, and the rasj HTML documentation of class `OQLQuery` for details on query return objects.

To access cells from arrays in query result bags, accessor functions are provided, such as `getObject()`, `getInteger()`. These functions are supervised by the type checking mechanism, hence using a function on an in appropriate type will cause an exception of type `ClassCast-Exception`.

Generally speaking, it is up to the application to know the result type structure of the query it has sent to the server.

[2] This has nothing to do with transactions – after each completion of a query, the embracing transaction can be aborted indeed.

10 OIDs

10.1 Overview

The class `RasOID` manages object identifiers (OIDs) for persistent MDD and collections.

10.2 Class Hierarchy

```
java.lang.Object
  |
  +--rasj.odmg.RasOID
```

Note

Class `java.lang.Object` obviously has further subclasses, not just the one shown here.

10.3 How To Use

The following code fragment prints the OID for each object in a query result set.

```
myQu = myApp.newOQLQuery();
myQu.create( "select mr from mr" );
DBag resultSet = (DBag) myQu.execute();
if (resultSet != null)
{
  Iterator iter = resultSet.iterator();
  while ( iter.hasNext() )
  {
    RasGMArray result = (RasGMArray) iter.next();
    System.out.println( "<"
                        + result.getOID().getSystemName()
                        + "|"
                        + result.getOID().getBaseName()
                        + "|"
                        + result.getOID().getLocalOID()
                        + " >" );
    // last statement is equivalent to:
    // System.out.println( getObjectId( result ) );
  }
}
```

Get a fresh OID

The following code fragment prints the OID for each object in a query result set.

11 Type Management

11.1 Overview

rasdaman allows to define new types during runtime of the system. This is in contrast to programming languages where type structures are fixed at compilation time. rasdaman, therefore, offers separate mechanisms to maintain database types; these are provided through the `RasType` class and its subclasses. For each structure relevant in dealing with persistent (i.e., database stored) entities, a corresponding type class is provided.

Note

Right now, rasj does not allow to create and manipulate persistent types in the database; methods provided mainly serve to inquire the result type of a query for a maximum of code flexibility. The `rasdl` utility has to be used for that. In a future release, the APIs will also allow to create and manipulate persistent types in the database.

11.2 Class Hierarchy

```
class rasj.RasType
     |  (implements rasj.RasGlobalDefs)
     |
     +--class rasj.RasBaseType
     |    |
     |    +--class rasj.RasPrimitiveType
     |    |
     |    +--class rasj.RasStructureType
     |
     +--class rasj.RasCollectionType
     |
     +--class rasj.RasMArrayType
     |
     +--class rasj.RasMIntervalType
     |
     +--class rasj.RasOIDType
     |
     +--class rasj.RasPointType
     |
     +--class rasj.RasSIntervalType
```

11.3 How To Use

The following code piece demonstrates how the type structure given by some `RasType` object can be evaluated and printed in a user-friendly form.

```
// instantiate a sample MDD type object:
RasType rType = RasType.getAnyType( "marray <char, 1>" );

// Now let's forget again that we know rType, let's analyse.
// Check if the type object is some MDD type:
if (rType.getClass().getName().equals("rasj.RasMArrayType"))
{ // yes, it is an MDD; is it structured or simple?
  if (rType.isStructType())
  { // yes, structured:
    System.out.println( "Structured base type is: " +
                        rType.getBaseType() );
  }
  else
  { // no, atomic:
    System.out.println( "Atomic base type is: " +
                        rType.getBaseType() );
  }
}
else
{ // no, not an MDD at all.
  System.out.println(
                "type object doesn't describe an MArray." );
}
```

12 Exceptions

12.1 Overview

Exceptions serve to handle deviations from the desired flow of operation. Several exceptions can be thrown by rasj classes; as a general rule, all exceptions are subclassed from the general Java exception class `java.lang.Exception`. Exceptions are further grouped into four main classes

- `org.odmg.Exception`
- `java.lang.RuntimeException`
- `rasj.RasException`
- `rasj.RasRuntimeException`.

See the HTML documentation for details on the exception class hierarchy.

12.2 Class Hierarchy (pruned)

```
Class java.lang.Exception
  |
  +--class org.odmg.Exception
  |
  +--class java.lang.RuntimeException
  |
  +--class rasj.RasException
  |
  +--class rasj.RasRuntimeException
```

Note

All classes have further subclasses See Sections 4.2 and 5.2 for more information.

12.3 Handling Exceptions in the Client

Catching an exception can be done, for example, as shown below. Obviously there are several ways doing this – however, a few rules should be obeyed:

- Granularity of exception catching depends on the overall program structure and purpose. For example, for data insertion one may want to build not just one large transaction, but several smaller units which, in case of failure, can be rerun with less time expenditure.
- Don't forget to clean up program state during exception recovery – think of closing (aborting? committing?) transactions, closing the database, etc.

Sample exception handling code

The following code piece demonstrates simple exception handling. The whole database access code is wrapped into a try statement. In case of an exception, the corresponding catch statement attempts to abort the transaction (if any is open) and to close the database. If in the course of these actions another exception occurs (for example, because the communication line has broken down), an error message is generated and the program terminates.

```
try
{
  Implementation myApp = new RasImplementation(
                                "http://" + server + port );
  myDb = myApp.newDatabase();
  myDb.open(base, Database.OPEN_READ_ONLY);
  myTa = myApp.newTransaction();
  myTa.begin();
  // here do some work with the database
  myTa.commit();
  myDb.close();
```

```
}
catch ( java.lang.Exception e )       // catch any error
{
  System.out.println( e.getMessage() );
  try
  {
    if(myTa != null)
      myTa.abort();
    if(myDb != null)
      myDb.close();
  }
  catch ( org.odmg.ODMGException exp ) // catch an abort
                                       // or close error
  {
    System.err.println( "Cannot commit/close: "
                        + exp.getMessage());
  }
}
```

12.4 Exceptions in the Class `rasj.RasException`

The following exceptions are rasj specific:

RasDimensionMismatchException

The dimensions of the two operand objects do not match.

RasIndexOutOfBoundsException

The specified index is not within the bounds of the array indexed.

RasResultIsNoCellException

The operation result is no cell, but an array cell is expected at this position. This happens, e.g., if the cast operator for casting to the base type of class `RasGMarray` is invoked on an object which is not 'zero-dimensional'.

RasResultIsNoIntervalException

The result is no interval, but an interval is expected at this position.

RasStreamInputOverflowException

An initialization overflow occured. This happens, e.g., if the stream input operator is invoked more often than the object has dimensions.

RasTypeInvalidException

Access method does not fit base type.

12.5 Exceptions in the Class `org.odmg.QueryInvalidException`

RasQueryExecutionFailedException

This exception extends `ODMGQueryInvalidException` by offering direct access to the rasdaman error number and the line, column and token in the query string that produced the error.

12.6 Exceptions in the Class `org.odmg.ODMGRuntimeException`

RasConnectionFailedException

This exception is raised when the connection to the server fails.

12.7 Exceptions in the Class `rasj.RasRuntimeException`

RasClientInternalException

This runtime exception indicates an internal error on client side which cannot be solved by the user. In case of such an event, please send a report to your dealer containing the complete error message and a precise description of the actions that lead to this exception.

RasTypeNotSupportedException

This exception is raised when the base type of a query result is not supported by the current version of the rasj package.

RasTypeUnknownException

This exception is raised when the base type of a query result is unknown on client-side..

13 Compilation and Execution of Client Programs

13.1 Compiling Code Using rasj

Environment Variables

The `CLASSPATH` variable – which is used by the Java compiler to locate packages used – must be extended with the path for the rasj directory of the rasdaman distribution. This can be done, e.g., with the following command:

```
export CLASSPATH=$RMANHOME/jlib/rasj.jar;$CLASSPATH
```

Alternatively, the `-classpath` option of `javac` can be used to explicitly make known the package locations to the Java compiler.

Further, the JDK class directory must be contained in `CLASSPATH`, and the JDK binaries directory must be contained in the `PATH` variable.

Java sources making use of the rasj package are compiled and run as usual. For example, some source file `Lookup.java` containing class `Lookup` would be compiled as

```
javac Lookup.java
```

Running it as an application would be done through this command line statement:

```
java Lookup
```

Sample Programs

Several sample Java programs are provided as part of the rasdaman distribution; they are located in the `examples/java` directory of the distribution.

Applets and Applications

rasj allows to build applications written in Java which can be applets as well as applications.

Notes

Remember the uppercase/lowercase distinction of Java!

For all classes with package definitions – such as `rasj.RasGMArray` – the package name must be prefixed.

13.2 Java Version Compatibility Statement

rasj has been successfully tested with JDK versions up to 1.6.

13.3 HTTP communication

rasj internally uses HTTP to communicate with the rasdaman server. By selecting individual URLs and ports in the database open statement (see Section 5), safe database access across firewalls is possible.

13.4 Copyright Note

rasj contains code for password encoding based on MD5.

Provision of this code is done in accordance with the GNU *Library General Public License* (see `www.gnu.org`).

13.5 Legal Note

Note that under some legislations usage and/or distribution of cryptography code may be prohibited by law. If you have obtained the above-mentioned library in or from a region under such a legislation, whatever

you do with it is fully under your own responsibility. Please inform rasdaman GmbH about the source where you have it obtained from so that we can take action against any violator.

14 HTML Documentation

The implementation is described in extensive documentation integrated with the source code from which a set of HTML files. This documentation can be used with any Web browser. The entry point for the complete documentation pages, including the rasj part, is `doc/index.html` in the rasdaman distribution directory (see *Installation and Administration Guide*, Section 3).

ODMG Class Availability

Note that the `org.odmg` package is taken verbatim from the ODMG standard. rasdaman interface classes are derived as implementations of the standard classes. However, only those classes have been implemented which are necessary for rasdaman. If in doubt, the `Implementation` section should be consulted where unavailable items are marked (due to copyright restrictions, the ODMG text must remain unchanged).