MozartSpaces Tutorial

Version 2.2.1, 21.12.2012

Written by Michael Wittmann, Bernhard Efler, Tobias Dönz & Martin Planer

Table of Contents

Chapter 1: Introduction

This tutorial describes *MozartSpaces*, the Java implementation of *eXtensible Virtual Shared Memory (XVSM)*. XVSM is a space-based middleware concept where data objects are stored in a *space* that can be shared among peers. It is developed at the Space Based Computing Group, Institute of Computer Languages, Vienna University of Technology.

The space-based approach makes it easy to cooperatively work on a task. For example, you can implement a system where a peer takes one data object of the space and performs some calculation on it, while another peer waits for the result of this calculation. The two peers can be separated from each other in physical space as well as in time. This means, that the peers do not need to know each other, they "communicate" just over the space, and they do not need to be active at the same time, because the result data remains in the space after writing it. Thus, the peer that waits for the calculation result can access this entry even when the peer that calculated it is no longer present. It is also possible to improve the performance by letting several peers process data in the space concurrently.

XVSM offers the possibility to manage the data objects with coordinators. For example, one coordinator can retrieve the objects in the order they were written and another coordinator can retrieve them in random order. It offers the possibility to perform several operations within one transaction. They are either performed all in one atomic action, when the transaction is committed, or none of them, when the transaction is rolled back (because of an error). XVSM can be extended during runtime with aspects, arbitrary code that is executed when the space is accessed, for example when data objects are added or removed.

For a high-level overview of XVSM and Space-based computing in general, and how you can use it, you should read the "Application Scenarios" document. This tutorial helps you to start programming with MozartSpaces. While most of its functionality and important API is covered in the following chapters, it is not a 100 % complete documentation of all details. You should also take a look at the Javadoc API reference on the website[1]. MozartSpaces Version 2.2 requires Java 6 or newer.

[1] http://www.mozartspaces.org

In the next section of this chapter a minimal example is described. In Chapter 2 the start-up, shutdown and basic configuration of MozartSpaces is explained. Chapter 3 gives an overview to structuring the space with containers and the ways to manage and coordinate data within them. Chapter 4 introduces the transactions in MozartSpaces. After that, Chapter 5 shows you how to extend the functionality of XVSM by using aspects. The notifications (Section 5.5), which can be used to listen for new or changed data, are an example for the use of aspects. Finally, Chapter 6 mentions some advanced features like the container persistence and custom coordinators.

1.1 “Hello Space!”

You probably know the "Hello World!" example as first program in a new programming language. We have adapted it for MozartSpaces as shown below. The text “Hello Space!" is written to a shared container from which it is read and printed to the console. First a space instance and a CAPI (Core API) is created. The `Capi` class offers you all important methods to operate with MozartSpaces. The details of its parts are explained in the subsequent chapters.

You can download the source of this and some other examples on the MozartSpaces website.

```
public static void main(final String[] args) throws Exception {
    System.out.println("MozartSpaces: simple 'Hello, space!' with
        synchronous core interface");

    // create an embedded space and construct a Capi instance for it
    MzsCore core = DefaultMzsCore.newInstance();
    Capi capi = new Capi(core);

    // create a container
    ContainerReference container = capi.createContainer();

    // write an entry to the container
    capi.write(container, new Entry("Hello, space!"));
    System.out.println("Entry written");

    // read an entry from the container
    ArrayList<String> resultEntries = capi.read(container);
    System.out.println("Entry read: " + resultEntries.get(0));

    // destroy the container
    capi.destroyContainer(container, null);

    // shutdown the core
    core.shutdown(true);
}
```

Chapter 2: Startup, Shutdown and Configuration

This chapter explains the startup, shutdown and configuration of MozartSpaces.

2.1 Startup

MozartSpaces can be started in an embedded way as part of an application, or standalone. For the former approach the method `newInstance` of the class `DefaultMzsCore` can be used, like in the example above. For the latter variant the class `org.mozartspaces.core.Server` has a main method that internally creates a space. With the complete binary package that includes all dependencies a standalone instance can be started with

```
java –cp mozartspaces-dist-2.2-<buildQualifier>-all-with-
dependencies.jar org.mozartspaces.core.Server [port]
```

The parameter `port` is optional, defaults to 9876, and is described in the configuration section below. You can also start the `Server` class within an IDE or with Maven (see PCO example on the website).

The `MzsCore` is the basic interface to access a MozartSpaces instance. For easier usage the classes `Capi` (synchronous space operations) and `AsyncCapi` (asynchronous space operations) are provided. You can use the methods of these classes to access embedded and remote spaces like it is explained in the following sections and chapters of this tutorial. As already shown in the example above, you can use the following two statements to create an embedded space and a `Capi` instance for it:

```
MzsCore core = DefaultMzsCore.newInstance();
Capi capi = new Capi(core);
```

Every space instance is by default remotely accessible for other peers (spaces or space APIs) on a specific port. This can be disabled or the port can be changed via the configuration as described in Section 2.3.

2.2 Shutdown

To shut down a Space, you can use the according `Capi` method:

```
void shutdown(URI space) throws MszCoreException
```

- The URI argument specifies the space to shut down. How the URI of a space is formed is explained in the next section. Use `null` for the embedded space of

a core. You can also shut down remote peers, regardless whether they are embedded or standalone spaces.

In the current implementation by default no security mechanism is enabled that prevents the shutdown of a peer. Furthermore, persistence is disabled by default, so all space information is stored only in-memory and lost on shut down.

2.3 Configuration

MozartSpaces has an internal default configuration that can be changed using an XML configuration file or with configuration objects that are passed when the space is created (see Javadoc for the class *`DefaultMzsCore`*).

MozartSpaces tries to locate the configuration file in this way:

- It is loaded from a file named `mozartspaces.xml` or the file specified with the system property *`mozartspaces.configurationFile`*.
- The system property can be set with
  `java -Dmozartspaces.configurationFile=<configfile> ...`
  or the equivalent setting in Maven (-D argument or element `systemPropertyVariables`), Eclipse (VM arguments), ant (element `sysproperty`) etc.
- The search order for the configuration file is (only relevant if just the file name or a relative path is specified)
  - the current directory.
  - the user's home directory (system property *user.home*).
  - the classpath.

A configuration file covering the most important options *`embeddedSpace`*, *`receiverPort`* and *`spaceURI`* looks like this:

```
<mozartspacesCoreConfig>
  <embeddedSpace>true</embeddedSpace>
  <remoting>
    <transports>
        <tcpsocket scheme="xvsm">
            <receiverPort>9876</receiverPort>
        </tcpsocket>
    </transports>
  </remoting>
<spaceURI>${remoting.defaultScheme}://localhost:${remoting.transports.tcpsocket.receiverPort}</spaceURI>
<!-- example above with variables, expanded to "xvsm://localhost:9876" -->
</mozartspacesCoreConfig>
```

The *embeddedSpace* option defines whether an embedded (local) space is started up when you create a new `MszCore` instance. When you set this to false the `MzsCore` instance is acting as a client to access a remote space. This is useful when you use a dedicated server space or the spaces of other peers to store the data.

The *receiverPort* specifies the port which is opened by the space. You can set the port manually or let the runtime find a free port automatically by setting the option to 0. If you set a fixed receiver port, the configuration can only be used for one space on the same machine. In the example configuration above the port is set for the TCP socket transport which is the current default transport in MozartSpaces.

With *spaceURI* you specify the endpoint of your peer, which is used as identifier and locator. When you only want to connect spaces running on your local machine, you can leave it like it is. If you want to connect to or from spaces on other machines, you must set the space URI to a hostname or IP address that is reachable by those remote peers. By using the `${remoting.transports.tcpsocket.receiverPort}` placeholder the port is set to the configured value.

Further configuration options are explained in Chapter 6. A full configuration file with the default values is in the examples you can download from the web page. If you remove or comment out configuration tags their default values are used.

2.4 Logging

MozartSpaces uses slf4j for logging. In the standard configuration, slf4j is used with logback[2], which has its own configuration. An example configuration file for logback, `logback.xml`, is included in the examples. You need to move it to the classpath (or include it somehow in it) so that it gets used. By default, without the example configuration, logback is configured with the logging level DEBUG and a lot of debug messages are logged to the console.

[2] http://logback.qos.ch

Chapter 3: Space Structure and Data Coordination

This chapter explains the structure of a space and how the data objects (entries) are managed in it. First the containers to structure a space are explained, before the entries and coordinators are introduced.

3.1 Containers and Entries

Containers structure a space, they can be seen as “sub-spaces”, and are the place where entries are stored. An entry can be any serializable object. The figure below shows an example of a container with entries.

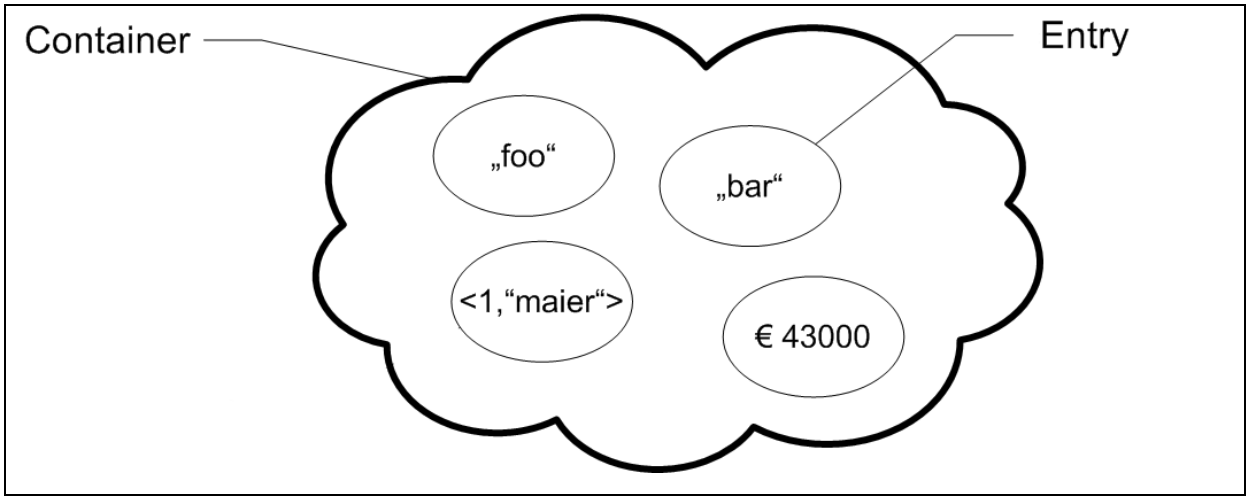
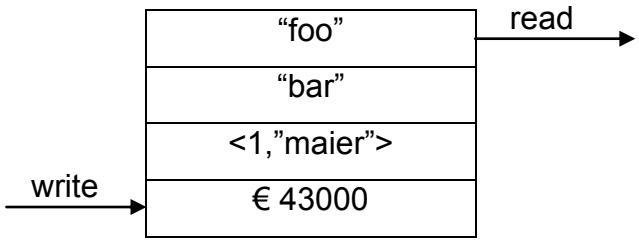


Figure 3.1: Example of a container with entries

To give you additional information about the entries, a table representation of the container is sometimes used. The arrows at the sides give the information of the order in which the entries are written or read. Such a table could look like this:



A container can either be located in a space on your local machine or on a remote machine. The following figure illustrates this.

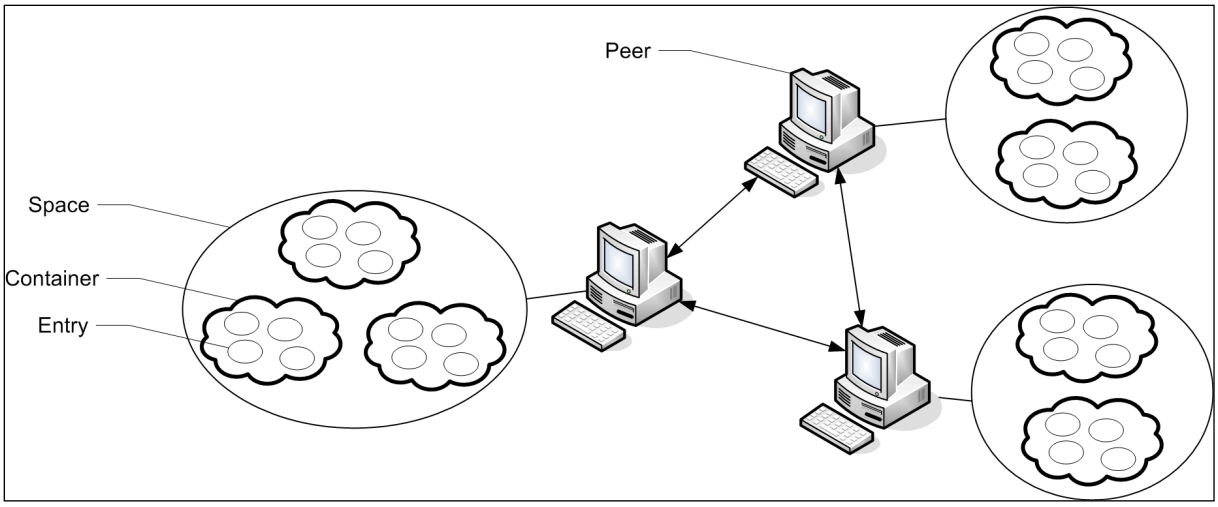


Figure 3.2: Communication of a set of peers, each with its own space with containers and entries

3.2 Container Operations

You can create a new container with the `Capi` method

```
ContainerReference createContainer(
          String name,
          URI space,
          int size,
          List<? extends Coordinator> obligatoryCoords,
          List<? extends Coordinator> optionalCoords,
          TransactionReference transaction)
    throws MzsCoreException
```

The parameters have the following meaning:

- Container's `name`: A container can have a name so that it can be looked up with the *`lookupContainer`* method (explained later), for example by another peer. Use `null` to create an unnamed container that cannot be looked up. Container names need to be unique relative to a space.
- URI of the `space`: this is the URI of the space where the container should be created. This is the configured space URI of a remote space, for example `URI.create("xvsm://mycomputer.mydomain.com:9876")`, or `null` for the embedded space.
- Container's maximum `size`: The maximum number of entries that the container can hold. If you try to write entries to a container which already holds the maximum number of entries, the write operation blocks. If you don’t want to

have such a maximum size, use `MzsConstants.Container.UNBOUNDED` as size parameter.

- List of obligatory coordinators: You can define the organization of entries in the container with so-called coordinators. They are either obligatory (this list), used for each entry that is written to the container, or optional (see below). The difference is important when entries are written to containers (see Section 3.3).
- List of optional coordinators: This is a list as for the obligatory coordinators.
- Transaction: A transaction makes a series of operations atomic. They are explained in Chapter 4. Use `null` to perform the operation with an implicit transaction that is automatically committed after the operation.

The *`createContainer`* method returns a `ContainerReference`, which can be used to access this container or data in it.

If you know that a named container already exists or a `ContainerNameNotAvailableException` is thrown on the creation, you can use the following method to get access to it with its name:

```
ContainerReference lookupContainer(
          String name,
          URI space,
          long timeoutInMilliseconds,
          TransactionReference transaction)
      throws MzsCoreException
```

The parameters of this method are similar to the ones used in *`createContainer`*. The timeout can be used to wait for some time that the container is created.

You also have the possibility to destroy a container.

```
void destroyContainer(
      ContainerReference container,
      TransactionReference transaction)
    throws MzsCoreException
```

To lock a container, use the `lockContainer` method. A container is locked until the transaction is committed or rolled back, which means that no operation outside of the transaction can access the container.

```
void lockContainer(
        ContainerReference container,
        TransactionReference transaction)
    throws MzsCoreException
```

These methods, as well as most other methods in MozartSpaces, can throw an `MzsCoreException`, which is the generalization of most exceptions in MozartSpaces.

3.3 Coordinating Entries

In XVSM, coordinators define how the entries are managed in a container. When entries are written to a container, they are registered at the coordinators. When entries in a container are selected to be read or taken (consuming read), the coordinator defines which entries are returned and in which order. For example, one coordinator can return the entries in the order they were written (FIFO) and another coordinator associates an entry with a unique name (KEY). Section 3.5 describes the pre-defined coordinators. In Chapter 6 it is explained how custom coordinators can be implemented.

On creation of a container, two lists of coordinators are specified, one of *obligatory* and one of *optional* coordinators. An obligatory coordinator has a complete view of all entries in the container, that is, every written entry is registered at it. In contrast, an optional coordinator may have only a partial view of the entries in the container.

3.3.1 Coordination Data

When an entry is written to the container, *coordination data* specifies additional properties for the entry. As already mentioned, an entry can be any object that implements the `Serializeable` interface. To be able to pass the coordination data, each entry is encapsulated in an `Entry` object together with the list of coordination data:

```
org.mozartspaces.core.Entry entry = new Entry(serializeable, coordData);
```

There is a specific coordination data class for each coordinator (type) and a coordination data instance specifies the coordinator instance with its name, which is unique for a container. Usually this coordinator name does not need to be specified and a

default name will be used, but this allows using two instances of the same coordinator for a container. For example, entries in a container can be managed by two KEY coordinators “key1” and “key2”.
While coordination data allows specifying additional properties for registering an entry at a coordinator, this is not necessary for all coordinators. For example, a FIFO coordinator has no additional properties, the entry order is determined implicitly with their registering. Therefore we distinguish between *implicit coordinators*, coordinators without additional properties, and *explicit coordinators* where properties are required. For implicit coordinators that are obligatory for a container, no coordination data needs to be specified. However, coordination data must be specified for explicit obligatory coordinators, and for all optional coordinators (explicit and implicit ones) that should be used.

3.3.2 Selectors

Selectors are used to read and take entries from a container. Analogous to coordination data for writing, there is a selector class for each coordinator. All selectors can specify the name of the coordinator and a *count*, the number of entries that should be returned. Additional coordinator-specific selector properties are possible. For a selecting operation multiple selectors can be specified in a list. They are evaluated in the order they have in this list and each coordinator returns a list of entries that match the selector, which is used as input for the next selector/coordinator pair.
The count parameter of a selector can be a concrete number or one of the constants COUNT_MAX and COUNT_ALL (in the class `MzsConstants.Selecting`). For COUNT_MAX all available entries are returned. For COUNT_ALL all entries are returned, but if at least one is locked and thus not available, an `EntryLockedException` is thrown. Entries can be locked because of open transactions (see Chapter 4). When a concrete count is used, the behaviour when entries are locked depends on the coordinator.

3.4 Entry Operations

In this section the operations to work with entries in containers are described. There is the `write` operation to add entries to a container and there are the selecting operations `read`, `test`, `take` and `delete`.

As for the container operations a transaction can be specified. Transactions are explained in Chapter 4 and you can ignore this parameter for now (pass `null`).
All entry operations have a blocking behaviour that is influenced by the timeout parameter. A write operation blocks when not enough space is in a bounded container or a coordinator temporarily cannot register the entry. A selecting operation blocks when not enough entries that match the selectors are available or entries are blocked by transactions. A blocked operation returns with a result when entries are removed or written such that it can be fulfilled or with an exception when the timeout is reached (or another error occurs). The timeout can be a concrete time in milliseconds or one of the following constants from `MzsConstants.RequestTimeout`:

- ZERO: The operation does not block and returns with an exception when it is not successful.
- INFINITE: The operation may block forever.
- TRY_ONCE: The operation blocks when entries are locked, but not when too little entries are available or the container is full.

Timestamps are stored at the target space when the request is processed and timeouts are detected for blocked operations. The time for transporting the request over the network is not included.

3.4.1 write

```
public void write(ContainerReference container,
            long timeoutInMilliseconds,
            TransactionReference transaction,
            Entry... entries )
    throws MzsCoreException
```

With this method you can write one or more entries to a container. It may block when the container size is limited and the container is already “too full”.

3.4.2 read

```
<R extends Serializable> ArrayList<R> read(
              ContainerReference container,
              List<? extends Selector> selectors,
              long timeoutInMilliseconds,
              TransactionReference transaction)
          throws MzsCoreException
```

This method selects and returns entries from a container without deleting them. It may block when not enough entries match the specified selectors. The following example shows writing and reading some entries with a FIFO coordinator and the corresponding selector.

```
// create new container with FIFO coordinator, size 5
ContainerReference cref = capi.createContainer(null, null, 5, null,
    new FifoCoordinator());

// fill the queue with the numbers 1 to 5
for (int i = 1; i <= 5; i++) {
    // FifoCoordinator is implicit, no coordination data necessary
    Entry entry = new Entry (i);
    capi.write(cref, entry);
}
// read one entry (default selector count of 1)
ArrayList<Integer> readEntries = capi.read(cref, FifoCoordina-
tor.newSelector(), 0, null);
// prints 1
System.out.println(readEntries.get(0));

// read two entries with 10 ms timeout
readEntries = capi.read(cref, FifoCoordinator.newSelector(2), 10, null);
// prints 1
System.out.println(readEntries.get(0));
// prints 2
System.out.println(readEntries.get(1));
```

Writing and reading entries with a FIFO coordinator.

3.4.3 test

This operation works like `read` but returns only the number of selected entries, not the entries themselves.

3.4.4 take

This method selects and returns entries from a container and deletes them. It can be seen as a “consuming read” or a `read` and `delete` in one atomic step.

3.4.5 delete

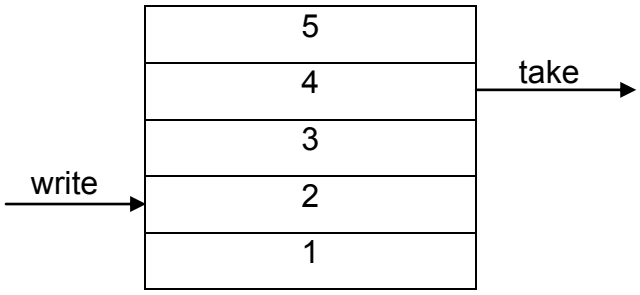
This operation works like take but returns only the number of selected and deleted entries, not the entries themselves.

3.5 Pre-defined Coordinators

This section describes the coordinators that are pre-defined in MozartSpaces. In Chapter 6 it is explained how custom coordinators can be implemented.
All pre-defined coordinators are in the package `org.mozartspaces.capi3`, have constructors to create instances that can be passed to the `createContainer` method, and static methods to create coordination data or a selector.

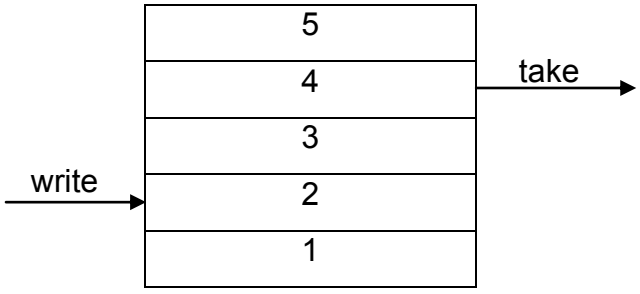
3.5.1 AnyCoordinator

This is an implicit coordinator that returns the entries in no particular order. It is used as default coordinator (or selector) for the overloaded `Capi` methods without a coordinator or selector parameter. Locked entries are ignored during selection (except for COUNT_ALL).



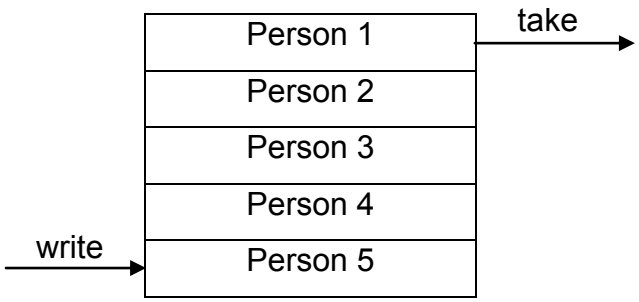
3.5.2 RandomCoordinator

This coordinator is implicit and works similar to the AnyCoordinator, but the entries are shuffled before they are selected. Locked entries are ignored during selection (except for COUNT_ALL).



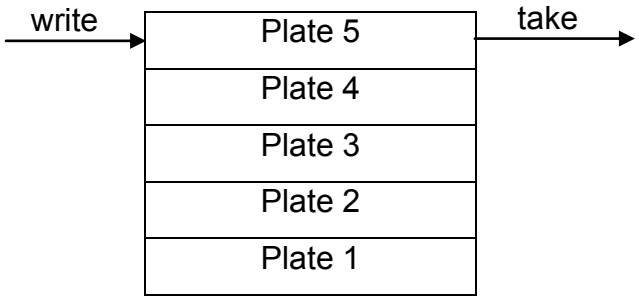
3.5.3 FifoCoordinator

This implicit coordinator guarantees first-in-first-out (FIFO) order, a queue. The entries are selected in the same order as they were registered at the coordinator.
When an entry is locked before the selection from the queue is finished, this coordinator throws an exception for a concrete count or COUNT_ALL. For COUNT_MAX the entries up to the first locked one are returned.



3.5.4 LifoCoordinator

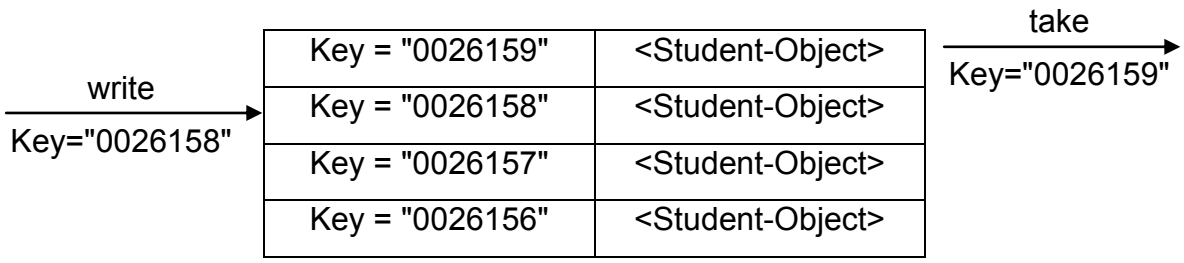
This implicit coordinator ensures last-in-first-out (LIFO) order, a stack. The entries are selected in the reverse order as they were registered at the coordinator.
The locking semantic is the same as for the FifoCoordinator. In fact, internally the FIFO and LIFO coordinators use the same implementation.



3.5.5 KeyCoordinator

This is an explicit coordinator that associates a unique key with an entry. On write this key is passed in the coordination data. To select the entry this key has to be

passed in the selector. The key must be a `String`. The key can also be part of the entry that is stored, but it also has to be specified explicitly in that case.
On write, if there is already an entry registered with the key, a `DuplicateKeyException` is thrown. Locked entries are ignored during selection (except for COUNT_ALL). Of course it does not make sense to selector more than one entry, but selecting a locked entry blocks for count 1 and returns an empty list for COUNT_MAX.



3.5.6 LabelCoordinator

This explicit coordinator associates a label with an entry. It is very similar to the key coordinator. The only difference is that a label does not need to be unique like a key. Locked entries are ignored during selection (except for COUNT_ALL).

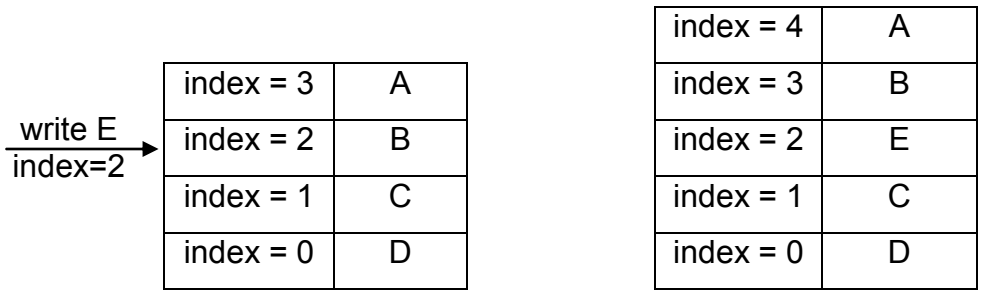
3.5.7 TypeCoordinator

This implicit coordinator stores entries according to their type. It can restrict the types of the entries that can be registered with a list of allowed types in the constructor. This is optional and no type restrictions apply when no such list (`null`) is passed. On selection, the type of the entries that should be selected can be specified. The passed types can be super-types of the entries that should match. Locked entries are ignored during selection (except for COUNT_ALL).

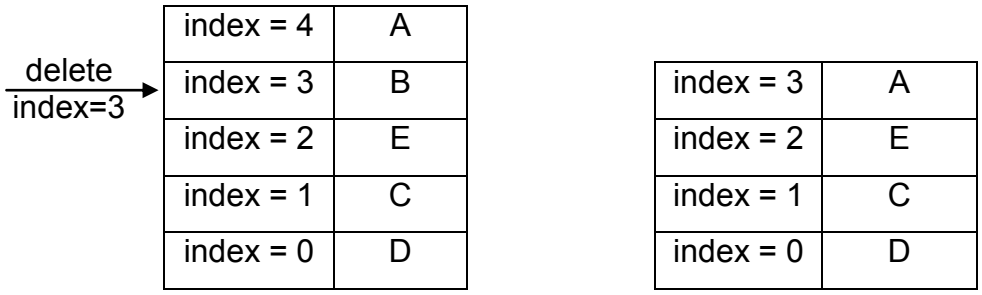
3.5.8 VectorCoordinator

This coordinator is explicit and specifies the index in a vector (or list) for an entry.
When an entry is registered at the coordinator it can be inserted at a specific index or at the end with the constant `APPEND`. If an entry is inserted, the entry at this position (if any) and all subsequent entries are shifted and their index increments. If an entry is

deleted at an index, all subsequent entries are shifted and their index decrements. This is the same behaviour as for a `Vector` or `List` in `java.util`.



Note that the entry B in the example above was stored with index 2, but has index 3 after entry E was inserted. In the example below the entry B at index 3 is deleted and the index of entry A automatically changes from 4 to 3.



Analogously to the FIFO/LIFO coordinators, when an entry is locked before the selection from the vector is finished, this coordinator throws an exception for a concrete count or COUNT_ALL. For COUNT_MAX the entries up to the first locked one are returned.

3.5.9 LindaCoordinator

The Linda coordinator is an implicit coordinator that allows to select entries with a template (template matching). Linda template matching is the selection mechanism that most tuple spaces support, for example JavaSpaces, though not always with the same syntax and semantics.

In MozartSpaces, the template is an object of the same type as the entries you want to select and its fields can contain a concrete value or a wildcard. We use `null` as wildcard and any other value as concrete value. A concrete value has to match ex-

actly (standard Java `equals`) while a wildcard matches all values for that field. Locked entries are ignored during selection (except for COUNT_ALL).
For compatibility with older MozartSpaces versions, by default the entry classes have to be annotated so that they can be used with the Linda coordinator. This can be changed with a flag when the coordinator is constructed.
When annotated entries are required, their class has to be annotated with `@Queryable` and fields that are used for matching with `@Index`. The element `autoindex` of `@Queryable` can be used to automatically index all fields of a class.

```
class Book implements Serializable {
    private final String title;
    private final List<Person> authors;
    private final Integer pages;
    // constructor omitted
}

@Queryable(autoindex = true)
class Person implements Serializable {
    private final String firstName;
    private final String lastName;
    // constructor, equals and hashCode omitted
}
```

Classes for Linda template matching examples, Person is annotated specifically

Fields with primitive types are allowed in entries and templates, but no wildcards for them are possible. Subclass matching is not supported, so an entry where the class is a subclass of the template class does not match. Template matching for fields of entries is supported, that is, a field of a template object may itself be a template. In this case the class must be annotated with `@Queryable`. Collections (instances of `java.util.Collection`) and templates inside collections can be used, but the size and iteration order of the template and entry collection must be the same, which practically restricts the use to lists.
Some examples to illustrate this:

```
// Linda Coordinator: do not require entry classes to be annotated
ContainerReference cref = capi.createContainer(null, Arrays.asList(new
LindaCoordinator(false)), null);
// 3 books
Person author11 = new Person("first", "author1");
Person author12 = new Person("first", "author1");
Person author2 = new Person("first", "author2");
Book book1 = new Book("title1", Arrays.asList(author11), 123);
Book book2 = new Book("title2", Arrays.asList(author12), 234);
Book book3 = new Book("title3", Arrays.asList(author11, author2), 345);
capi.write(cref, new Entry(book1), new Entry(book2), new Entry(book3));
```

```
// template 1: matches all books
Book tmplAll = new Book(null, null, null);
// template 2: matches book1 (all fields equal)
Book tmplSpecific = new Book("title1", Arrays.asList(author11), 123);
// template 3: matches book1 and book2
Book tmplAuthor1 = new Book(null, Arrays.asList(new Person(null, "au-
thor1")), null);
// template 4: matches book1 and book2
Book templateFirstName = new Book(null, Arrays.asList(new Per-
son("first", null)), null);
// template 5: matches book3
Book templateAuthors = new Book(null, Arrays.asList(author11, null),
null);
```

Linda template matching examples

The first two examples are simple, the most general template and an exact match. The third template matches *book2* because Person implements `equals` correctly. *book3* is not matched because it has two authors and the template has only one. Template 4 is similar but matches *book1* and *book2* with a wildcard in the author template. The fifth template matches the only book with two authors, *book3*.

3.5.10 QueryCoordinator

This implicit coordinator can be used to select entries that meet the specified criteria, that is, their fields (properties) match the selection query. Locked entries are ignored during selection (except for COUNT_ALL).
The main method for specifying the query criteria is by using the object-based query language, but the coordinator also supports an SQL-like string-based query language that can be used for issuing simple queries in a quick and succinct way.

Object-based Query Language

The starting point for building a query using the object-based query language is the `Query` object. A newly instantiated Query object represents a query without any restrictions or filters defined. Predicates that the resulting entries of this query should match (therefore reducing the result set) can be added by the following methods.

- ***cnt****(n)*: Takes *n* entries of the input.
  - This predicate will only return the next *n* available (not locked) entries and will skip locked entries. Due to this circumstance, the results may not be strictly in order.
- ***cnt****(min, max)*: Takes at least *min* and at most *max* entries from the input.

  - This predicate will at least return the next *min* and up to the next *max* available (not locked) entries and will skip locked entries. Due to this circumstance, the results may not be strictly in order.
  - Additionally the special value *Query.ALL* can be used as either the *min* and/or *max* parameter to represent the count of all stored entries in the container.
    - cnt(3, ALL) = at least 3
    - cnt(ALL, ALL) = cnt(ALL) = require all stored, matching entries

- ***reverse****()*: Reverse the order of the input sequence.
- ***sortup****(Property)*: Sort the input sequence ASCENDING.
  - The ordering of the sorting is determined by the value of the given Property.
- ***sortdown****(Property)*: Sort the input sequence DESCENDING.
  - The ordering of the sorting is determined by the value of the given Property.
- ***distinct****(Property):* Distinct the entries by the given Property
  - Only returns entries with mutually different values for the given Property. The values of the Property are evaluated by the *hashCode* and *equals* methods of the objects.
- ***filter****(Matchmaker)*: Filter entries based on the given Matchmaker.
  - An entry will only be added to the result set of the query if it evaluates to true in the given matchmaker.

Note: The order of the predicates when defining the query matters, as the evaluation order is the same as the definition order and each stage will receive the output of the previous stage as its input.

Example:

```
import org.mozartspaces.capi3.Query;

// Create a new Query object that filters, sorts and returns 3 items.
Query query = new Query.filter(…).sortup(…).cnt(3);
```

Simple query example

Properties

Properties are used to specify the path to the value that should be used by the predicate for evaluating an entry (distinct, sort, filter, etc.). This can be e.g. the entry itself or a value that is nested several layers deep in the entry.

There are two different types of Properties, a basic *Property* and the *ComparableProperty*. Both share most of the functionality for simple comparison on equality, but the *ComparableProperty* can also be used to compare values with a natural Ordering (>, ≥, <, ≤).

Properties are created by using the static methods of either the *Property* or the *ComparableProperty* class:

- ***forName****(String, …)*
  - The String arguments for this method define the path of the property. The path specifies the field/object that will be used for the query evaluation.
- ***forClass****(Class, String, …)*
  - This method has an additional Parameter *Class* and only matches on entries that are assignable to the given class type.
  - The subsequent String arguments define the path of the property like in the *forName* method.

The path of a property is defined by one or more path tokens where each token can be one of the following:

- **Field name**
  - Specifies the token by matching the name of the field in the object. If the field is annotated by the @*Index* annotation, the name is matched against the defined label.
- **#this**
  - Is used to refer to the entry object itself and can be used to evaluate its value in a query.
- *** (asterisk)**
  - Acts as a wildcard and matches against any field at its position in the path.
  - A single wildcard at the beginning of the path is ignored.

- **** (double asterisk)**
  - Acts as a “deep” wildcard and matches against any number of nested fields in the path.
- **[<int>]**
  - Is used to navigate to a specific index in a *Collection* field.
- **[<int>..<int>]**
  - Is used to select a sub list in a *List* field.

The class Property defines also a static method *withName* that takes a path and returns a builder object that allows configuring the property further with the following methods:

- **entryClass**(Class): filter entries that are assignable to the given class, like the static *forClass* method (see above)
- ***propertyClass****(Class):* Is used if the object specified by the given path should only be matched if it is assignable to the given Class.
- ***enableCache****():* Enables caching of the property (see below for details)
- ***returnSize****():* When this option is set on a property, the property will only return the size of the *Collection* on the given path.

These builder methods can be concatenated arbitrarily. Use the method *build* or *buildComparable* to create a *Property* or *ComparableProperty*, respectively.

Examples:

```
import org.mozartspaces.capi3.Property;

// Match the name of the author
Property.forName(“author”, “name”)
// OR
Property.forName(“*”, “author”, “name”)

// Match any author field
Property.forName(“author”, “*”)

// Match any name
Property.forName(“**”, “name”)

// Match the entry itself
Property.forName(“#this”)
// OR
Property.forName(Property.PATH_THIS)
// OR
Property.forName()

// Match comparable age
ComparableProperty.forName(“age”)
```

```
// Select a specific index / sub list
Property.forName("tags", "[3]") // 4th element
Property.forName("tags", "[0..3]") // 1st to 4th sub list

// Only match the name of the author on a Book
Property.forClass(Book.class, "author", "name")

// Match size/count of movies collection
Property.withName("movies").returnSize().buildComparable()

// Only match names that are Strings
Property.withName("name").propertyClass(String.class).build()
```

Property examples

Matchmakers

Matchmakers are used to match the values specified by a Property against the given criteria. Entries are only included in the result set if they evaluate to *true* in all defined matchmakers.

The following methods for creating a matchmaker are available on both the *Property* and *ComparableProperty* classes.

- ***exists****()*: Checks for the existence of the property value on an entry.
- ***equalTo****(Object)*

  equalTo*(Property)*: Checks for equality of the property value with the given object or the value specified by a Property given as an argument. Represents =.
- ***notEqualTo****(Object)*

  notEqualTo*(Property)*: Checks for inequality of the property value with the given object or the value specified by a Property given as an argument. Represents ≠.
- ***allEqualTo****(Object)*

  allEqualTo*(Property)*: Checks for equality of all elements of the property value with the given object or the value specified by a Property given as an argument.

  Represents =.
- ***allNotEqualTo****(Object)*

  allNotEqualTo*(Property)*: Checks for inequality of all elements of the property value with the given object or the value specified by a Property given as an ar-

gument.
Represents ≠.

- **elementOf***(Object, …):* Checks if the property value is an element of the supplied object set. Represents ϵ.
- ***forAll****(Matchmaker):* Checks all elements of a *Collection* specified by the property against the given matchmaker and only evaluates to true if *all* element values evaluate to true in using the matchmaker. *See note below for more information*.

In addition to these basic matchmakers the *ComparableProperty* also defines the following methods for creating matchmakers that can be used for comparing entries with a natural ordering.

- ***greaterThan****(Comparable)*
  greaterThan*(ComparableProperty)*: Checks if the property value is greater than the given object or the value specified by a Property given as an argument. Represents >.
- ***greaterThanOrEqualTo****(Comparable)*
  greaterThanOrEqualTo*(ComparableProperty)*: Checks if the property value is greater than or equal to the given object or the value specified by a Property given as an argument. Represents ≥.
- ***lessThan****(Comparable)*
  lessThan*(ComparableProperty)*: Checks if the property value is less than the given object or the value specified by a Property given as an argument. Represents <.
- ***lessThanOrEqualTo****(Comparable)*
  lessThanOrEqualTo*(ComparableProperty)*: Checks if the property value is less than or equal to the given object or the value specified by a Property given as an argument. Represents ≤.
- ***allGreaterThan****(Comparable)*
  allGreaterThan*(ComparableProperty)*: Checks if all elements of the property value are greater than the given object or the value specified by a Property given as an argument. Represents >.

- ***allGreaterThanOrEqualTo****(Comparable)*
  allGreaterThanOrEqualTo*(ComparableProperty)*: Checks if all elements of the property value are greater than or equal to the given object or the value specified by a Property given as an argument. Represents ≥.
- ***allLessThan****(Comparable)*
  allLessThan*(ComparableProperty)*: Checks if all elements of the property value are less than the given object or the value specified by a Property given as an argument. Represents <.
- ***allLessThanOrEqualTo****(Comparable)*
  allLessThanOrEqualTo*(ComparableProperty)*: Checks if all elements of the property value are less than or equal to the given object or the value specified by a Property given as an argument. Represents ≤.
- ***between****(Comparable, Comparable):* Checks if the property value is between the given comparable values (including the given lower and upper bound values).
- ***matches****(Regex):* Checks if the property string value matches the given regular expression specification. This can only be used on properties that refer to String fields.

  Note: If the base property refers to a collection of values (elements), the normal matchmakers (*equalTo*, *notEqualTo*, *greaterThan*, …) evaluate to true, if at least one of the collection elements evaluates to true if checked against the given object/property value. If this behavior is not desired, the matchmakers prefixed with *all* (*allEqualTo*, *allNotEqualTo*, …) or the *forAll* matchmaker can be used. These matchmakers only evaluate to true, if **all** of the collection elements evaluate to true if checked against the given object/property value.

Examples:

```
import org.mozartspaces.capi3.Property;

// Define a Property
Property name = Property.forName("name");
ComparableProperty age = ComparableProperty.forName("age");

// Build matchmakers
name.equalTo("Mustermann")
age.greaterThanOrEqualTo(18)
```

```
age.between(10, 30)
name.matches("^.*huber")
prices.allLessThan(100)
movies.forAll(ComparableProperty.forName("title").matches("^Matrix
\\d$"))
```

Matchmaker examples

Matchmaker connectives - These connectives can be used to combine two or more matchmakers and form complex queries.

- ***and****(Matchmaker, …)*: Logical conjunct an arbitrary number of Matchmakers. This is the representation of the logical function ∧.
- ***or****(Matchmaker, …)*: Logical disjunct an arbitrary number of Matchmakers. This is the representation of the logical function ∨.
- ***not****(Matchmaker)*: Negate a Matchmaker. Represents ¬.

Examples:

```
import org.mozartspaces.capi3.Matchmakers;

// Define a Property
Matchmaker allowed = Property.forName("allowed").equalTo(true);
Matchmaker ageCheck = ComparableProperty.forName("age")
.greaterThan(18);

// Connected matchmakers
Matchmakers.or(ageCheck, allowed)

// with statically imported matchmakers
import static org.mozartspaces.capi3.Matchmakers.or;

or(ageCheck, allowed)
```

Matchmaker examples

Arithmetic operators and *Concat* - These operators can be used to perform simple arithmetic operation on the property values. Only *Character*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double* can be used with these methods. In accordance with the Java semantics of arithmetic operations, the return type of an arithmetic operation (excluding concat) will be the greater type of both arguments (according to the type list above, ranking from low to high – e.g. Byte + Long => Long).

- ***add****(Property, Property)*
  add*(Comparable, Property)*
  add*(Property, Comparable)*: Add both values. Represents +.

- ***subtract****(Property, Property)*
  subtract *(Comparable, Property)*
  subtract *(Property, Comparable)*: Subtract the second value from the first value. Represents -.
- ***multiply****(Property, Property)*
  multiply*(Comparable, Property)*
  multiply*(Property, Comparable)*: Multiply both values. Represents *.
- ***divide****(Property, Property)*
  divide*(Comparable, Property)*
  divide*(Property, Comparable)*: Divide the first value by the second value. Represents /.
- ***concat****(Property, Property)*
  concat*(Comparable, Property)*
  concat*(Property, Comparable)*: Concatenate the string representations of both values and return the result as a string.

Examples:

```
import org.mozartspaces.capi3.Arithmetic;

add(Property.forName("salary"),
  Property.forName("taxes")).greaterThan(2000)
concat(Property.forName("name"), " 1") // -> "<Name> 1"
```

Arithmetic examples

Complete Object Based Example Query

```
import org.mozartspaces.capi3.Query;
import org.mozartspaces.capi3.Property;
import org.mozartspaces.capi3.ComparableProperty;
import org.mozartspaces.capi3.Matchmakers;
import org.mozartspaces.capi3.Arithmetic;
…
ContainerReference cref = capi.createContainer(null, null,
    MzsConstants.Container.UNBOUNDED,
           Arrays.asList(new QueryCoordinator()), null, null);

Movie movie = new Movie("The Godfather", Arrays.asList(
    "Marlon Brando", "Al Pacino"), 175);
Entry entry = new Entry(movie, QueryCoordina-
tor.newCoordinationData();
capi.write(cref, 0, null, entry);

movie = new Movie("Shrek", Arrays.asList(
    "Mike Myers", "Cameron Diaz"), 87);
entry = new Entry(movie, QueryCoordinator.newCoordinationData();
```

```
capi.write(cref, 0, null, entry);

Property title = Property.forName("title");
Query query = new Query().filter(title.equalTo("Shrek"));

// get entry using QuerySelector from the container
ArrayList<Integer> readEntry = capi.read(cref,
      Arrays.asList(QueryCoordinator.newSelector(query)),
      0, null);

System.out.println(readEntry);
```

Listing 3.4.14: QueryCoordinator

String-based Query Language (SQL)

As an alternative to the above way of defining a query, MozartSpaces now also contains a string-based query language for defining simple queries without the overhead of the object-based approach.

The language that is used for specifying queries in text form is based on the Structured Query Language (SQL) but has been modified to incorporate the functionality to navigate the object tree of the entries. Only parts of the SQL WHERE clause and the LIMIT function can be used to define a query.

To navigate the object tree structure, a dot-syntax has been added when specifying identifiers, similar to the definition of paths for a *Property*, e.g.
"person.address.zip" is equivalent to *Property.forName("person", "address", "zip")*

The XVSM SQL dialect currently supports:

=	Equality	BETWEEN	Value between two bounds
<>	Inequality	LIKE	String matches pattern
>	Greater than	IN	Element of
>=	Greater than or equal to		
<	Less than	AND	Logical AND
<=	Less than or equal to	OR	Logical OR
		NOT	Logical NOT

LIMIT <n> can be used at the end of the query string to limit the results returned by the query to the next n available (not locked) entries. It is equivalent to cnt(0, <n>).

Number literals are interpreted according to Java semantics and must be suffixed with *F|f, D|d or L|l* to be cast to *float, double* or *long* types, respectively.
A string-based query can be created by using the *sql()* method on a Query object. The method parses the input string and returns a MozartSpaces Query as with the object-based Query methods. Therefore the two ways of building queries can be mixed and matched when wanted.

Examples:

```
// Simple queries
new Query().sql("age = 20")
new Query().sql("price <= 20.0f")
new Query().sql("name LIKE '_oo%Bar_az%'")
new Query().sql("person.address.zipcode <= 1000")

// Inter-property comparison
new Query().sql("father.address.zipcode = mother.address.zipcode")

// Connected predicates (operator precedence: NOT > AND > OR)
new Query().sql("age = 20 OR age = 30 AND NOT name = 'Foo'")
// OR (EQUIVALENT)
new Query().sql("((age = 20 OR age = 30) AND (NOT name = 'Foo'))")

// Limit results
new Query().sql("age > 20 LIMIT 5")

// Mix and match
Query query = new Query()
    .cnt(10)
    .sql("age > 18")
    .distinct(Property.forName("name", "lastname"))
```

SQL query examples

Advanced Query Functionality: Indexing and Caching

If it is conceivable that some types of entries or fields on an entry will be read much more often than written to a container, it may be reasonable to index or cache the values to improve performance when accessing those entries and/or fields.

Note: Indexing and Caching can bring performance benefits for some queries but also introduce overhead in terms of memory consumption and runtime performance. They should be used with caution and the performance improvement should be measured before using them in production code.

Indexing

When indexing a field on a class of an entry, the values of these fields will be indexed upon writing such an entry to a container. When querying for an indexed field, the comparison will query the index directly instead of iterating over all entries in the container.

To make an entry class index-able it has to be annotated with the *@Indexable* annotation. Indexes for fields can then be added with the *@QueryIndex* annotation on the fields itself. Without passing any arguments to the *@QueryIndex* annotation a *BASIC* index for the annotated field is created. This behavior can be overridden by passing values for the 2 annotation arguments *type* and *path*.

The type can be either *IndexType.BASIC* or *IndexType.EXTENDED*. A basic index will only work for simple equality checks whereas an extended index will also be able to support comparisons for elements with a natural ordering (>, >=, <, <=). It is important to note that an extended index will also introduce a higher memory consumption and overhead when writing entries to the container and should only be used when necessary.

The path argument of the annotation defines the path to the field that should be indexed. By default this is the field on which the annotation was defined. The path can be overwritten if a field that is nested deeper in the entry should be indexed that cannot be annotated directly. The starting point of the path evaluation is always the field on which the annotation was defined. The semantic of the path tokens is the same as for the Property paths (see section for *Property* above).

If multiple indexes should be defined on one field, the compound annotation *@QueryIndexes(@QueryIndex, …)* has to be used because of the limitation of Java annotations.

After inserting an annotated entry, the appropriate index will automatically be created and will then be used for all future queries that refer to an indexed field if possible.

Example:

```
@Indexable
@QueryIndex// Index the entry itself
class Person {

    // Create extended index for age
    @QueryIndex(type = IndexType.EXTENDED)
```

```
    int age;

    // Create indexes for nested properties
    @QueryIndexes(
      @QueryIndex(path = {"street"}),
      @QueryIndex(path = {"zipCode"}, type = IndexType.EXTENDED)
    )
    Address address;

    ...
}
```

Index Annotation Example

If fields of entries shall be indexed that cannot be annotated (e.g. 3rd party library classes, etc.), the index data can be passed as the coordination data when writing the entries to the container.

Since Java annotation cannot be instantiated and passed directly, the class QueryIndexData has to be used instead. This class takes 2 parameters like the annotation, indexType and path. The path semantics are the same as with Property or the @QueryIndex annotation.

Example:

```
QueryIndexData indexData =
    new QueryIndexData(IndexType.BASIC, "name");
QueryIndexData indexData2 =
    new QueryIndexData(IndexType.EXTENDED, "age");

Entry entry =
    new Entry(nonAnnotatableClassObject,
         QueryCoordinator.newCoordinationData(indexData, indexData2));

capi.write(entry, containerReference);
```

Coordination data index example

Caching

When indexing is not possible (e.g. complex comparisons, etc.) and the evaluation of a property value is expensive (e.g. highly nested values or usage of "**") it may be beneficial to use property caching instead to improve the performance of a query.

Caching will store the evaluated object of a property for a limited amount of time. If the same property is accessed again, the value will not need to be computed again and will be taken from the cache instead.

To enable caching for a property, the method *enableCache()* has to be used on a property builder. Only if this flag has been set, the cache will be used for the query (both for inserting into the cache and querying from the cache).

Example:

```
Property name = Property.withName("name").enableCache().build()
Query.distinct(name) // value of name will be cached
Query.filter(name.equalTo(…)) // cached value of name will be used

// BUT (enableCache() not set):
Query.distinct(Property.forName("name")) // cached value will NOT be
used
```

Caching Example

3.6 Coordination Examples

The source code accompanying this tutorial contains four examples that show the use of different coordinators:

- **Lottery (Random)**: The Random coordinator is used to take 6 + 1 numbers (balls) out of the numbers 1 to 45.
- **Ticket Queue (FIFO)**: A queue of persons waiting at a ticket counter is simulated with the FIFO coordinator. This example is extended in Section 5.4 with notifications.
- **Formula 1 Race (Vector)**: The positions of the cars with changes during the race demonstrate the use of the Vector coordinator.
- **Book matching (Linda)**: The complete source of the examples used in the Linda coordinator description; demonstrates most features of the template matching.

3.7 Summary

In this chapter you have learnt how containers structure a space and how they can be used to coordinate entries with the help of coordinators. Besides the explanations of the container and entry methods in the `Capi` class, the pre-defined coordinators where explained in detail.

Chapter 4: Transactions and Locking

Transactions in MozartSpaces allow combining several space operations to one atomic action, similar to transactions in relational database systems. They use pessimistic concurrency control and provide the classic ACID properties atomicity, consistency, isolation and durability (when persistence is enabled). The operations inside of a transaction are either performed all, when the transaction is committed, or none, when the transaction is rolled back.
Currently the container and entry operations in MozartSpaces support transactions. A reference to a transaction can be passed when such an operation is executed. Internally these operations are inherently transactional. If no transaction reference is passed, a transaction is created for these operations (implicit transaction) and committed or rolled back automatically immediately after the operation. The methods for managing explicit transactions are explained below.

4.1 Transaction Operations

To create an explicit transaction, use this `Capi` method:

```
TransactionReference createTransaction(
          long timeoutInMilliseconds,
          URI space)
    throws MzsCoreException
```

The timeout specifies the “time to live” of the transaction. When this time elapses, the transaction is automatically rolled back. You can pass the returned `TransactionReference` to every method where this transaction should be used. However, MozartSpaces does not support distributed transactions, that is, a transaction may be used only for operations on the space where it has been created.
A transaction ends with `commit` to make the changes permanent, usually when no error occurs for the operations that use the transaction. A transaction ends with `rollback` to discard all changes, usually when an error occurs or a result is not as desired. These two methods have the same simple signature:

```
void commitTransaction(TransactionReference transaction)
    throws MzsCoreException
```

The effect of an explicit rollback of a transaction is the same as for an automatic rollback when the timeout elapses.

After rollback or commit a transaction reference is invalid and an exception is thrown if it is used.

The typical use of the transaction operations is shown below:

```
TransactionReference tx = capi.createTransaction(1000, null);
try {
    // perform transactional methods with tx
    capi.commitTransaction(tx);
} catch (MzsCoreException ex) {
    capi.rollbackTransaction(tx);
}
```

Typical transaction operation use

Of course it is also possible to have several active transactions at the same time and use them, or perform operations with implicit transactions in between.

4.2 Locking Behaviour

MozartSpaces uses locks on entries and containers to ensure the consistency and isolation of the transactions. A variation of the classic read/write lock pattern is used. *Insert locks* are used for inserted data objects, which are invisible to other transactions. *Delete locks* are used for data objects that will be deleted on commit. Depending on the isolation level (see below), also *read locks* are used, which prevent the deletion. Insert and delete locks are exclusive, no other lock is allowed on such a locked object. Read locks are shared, that is, multiple read locks on the same object are allowed.

A container or entry operation may fail because of locks. For example, an entry that has been deleted in a still open (not committed or rolled back) transaction cannot be taken by another transaction. However, the exact behaviour of the read operation in this case depends also on the coordinator (and selector) that is used.

4.2.1 Isolation Levels

MozartSpaces supports the isolation levels *repeatable read* and *read committed*. The difference between them is that read locks are used only for repeatable read. Reading delete-locked data is possible with read committed but not with repeatable read. Both isolation levels prevent reading uncommitted data (dirty read). The read locks of repeatable read also prevent the situation that two identical read operations in the same transaction return different results because entries have been deleted by another (already committed) transaction in between (non-repeatable read). However,

for both isolation levels, two identical read operations in the same operation may return different results because of inserted entries (phantom read).
In the current version, containers are never read locked, only entries.

4.2.2 Coordinator Locking Semantics

A coordinator with its selector defines the semantics of the entry operations when an entry is locked. Most selectors ignore and skip locked entries unless the selector count is COUNT_ALL (see Section 3.2.2). Selectors can also throw an exception when an entry is locked, for example to guarantee a specific entry order.
From the pre-defined coordinators the FIFO, LIFO, Key and Vector coordinators have a special locking behaviour. The FIFO and LIFO coordinators ensure the order of the entries and throw an exception if an entry is locked on selection. The Key coordinator allows that an entry is overwritten only in the same transaction. The Vector coordinator has the same rule for entry overwrite and guarantees the order of the selected entries, that is, throws an exception when a locked entry is encountered during selection.

4.3 Transaction Example

As example for transactions in MozartSpaces we take entries from two containers, process them and create a third entry which is then written to a third container. We want the two take and the write operation to be atomic. If taking the second entry, processing the two taken entries or writing the resulting entry fails, we want to abort the whole action and the state of the containers should be as before the transaction started (ignoring other operations that may be performed in the meantime). Thus we use one transaction for all three space operations and follow the typical use of transactions as shown above:

```
TransactionReference tx = capi.createTransaction(1000, null);
try {
    ArrayList<String> result1 = capi.take(c1,
       AnyCoordinator.newSelector(), RequestTimeout.ZERO, tx);
    String entry1 = CapiUtil.getSingleEntry(result1);
    ArrayList<String> result2 = capi.take(c2,
       AnyCoordinator.newSelector(), RequestTimeout.ZERO, tx);
    String entry2 = CapiUtil.getSingleEntry(result2);
    capi.write(c3, RequestTimeout.ZERO, tx,
       new Entry(entry1 + entry2));
    System.out.println("Combined entry. Committing.");
    capi.commitTransaction(tx);
} catch (Exception ex) {
```

```
    System.out.println("Combining entry failed. Rollback" + ex);
    capi.rollbackTransaction(tx);
}
```

Transaction example with several space operations on different containers

4.4 Summary

In this chapter you have seen how explicit transactions can be used to combine several space operations into one atomic action and how the locks for the pessimistic concurrency control are used to provide different isolation levels and coordinator semantics.

Chapter 5: Extension with Aspects

Aspects offer the possibility to extend the existing XVSM functionality. The idea behind XVSM aspects is to have an interface with special methods before and after each space operation, for example a `preWrite` and a `postWrite` method for a write operation. You implement these pre- and post-methods to perform some action that is executed automatically before or after the operation, when you call the corresponding method(s) of the `Capi` class.

There are two kinds of aspects – *container aspects* and *space aspects*. Container aspects are performed for operations on a specific container, while space aspects are executed for operations on all containers and for general space operations (transaction operations etc.). The hooks for the space operations where aspects are registered are called *interception points (ipoints)*. Aspects can be added and removed dynamically during runtime with special space operations.

By intercepting calls to space operations, aspects can be used to implement logging, monitoring, authentication functionality or the like.

5.1 Aspect Management Operations

Aspects can be added and removed dynamically during runtime with methods of the `Capi` class. There are different API methods to add container and space aspects, because they have different arguments (container reference or space URI). Internally they use the same code, so there is only one operation in the space (with corresponding aspect methods and ipoints).

To add a container aspect, use the following method:

```
AspectReference addContainerAspect(
          ContainerAspect aspect,
          ContainerReference container,
          ContainerIPoint... ipoints)
    throws MzsCoreException
```

The aspect object itself must implement the `ContainerAspect` interface. This interface contains the methods for the ipoints and is explained later. The `ipoints` are a set of fields from the enum `ContainerIPoint` for which the corresponding methods of the aspect should be called.

The method to add a space aspect is very similar:

```
AspectReference addSpaceAspect(
          SpaceAspect aspect,
          URI space,
          SpaceIPoint... ipoints)
    throws MzsCoreException
```

The interface `SpaceAspect` extends the interface `ContainerAspect`, as all ipoints for container aspects are also supported by space aspects. The enum `SpaceIPoint` contains the constants corresponding to the methods in `SpaceAspect`. (The constants also used for the container aspect ipoints are redefined here because enums cannot be extended in Java).

To remove a container or space aspect the following Capi method can be used:

```
void removeAspect(
          AspectReference aspect,
          InterceptionPoint... ipoints)
    throws MzsCoreException
```

You can remove an aspect from all or only some of the ipoints where it is registered.

The aspect management operations are overloaded and there are variants that take a transaction reference (and an isolation level) as argument. These arguments are ignored because currently the aspect management is not transactional.

5.2 Container Aspects

A container aspect has methods that are called for operations on a specific container that is specified when the aspect is added. The interface `ContainerAspect` has methods for the following ipoints:

- pre-read, post-read
- pre-test, post-test
- pre-take, post-take
- pre-delete, post-delete
- pre-write, post-write
- pre-destroy-container, post-destroy-container
- pre-lock-container, post-lock-container
- post-lookup-container
- pre-add-aspect, post-add-aspect
- pre-remove-aspect, post-remove-aspect

If you want to create a new container aspect, you can implement this interface or extend the `AbstractContainerAspect` class. This class has default implementations that cause the aspect to fail for all ipoints, and methods for initialization and cleanup. This abstract class is useful if you want to support just one or a few of the ipoints, which you can do by overriding the corresponding methods.

The code snippet below shows a simple aspect that implements the methods for three ipoints.

```
class LoggingAspect extends AbstractContainerAspect {
    public AspectResult preRead(ReadEntriesRequest<?> request,
        Transaction tx, SubTransaction stx, Capi3AspectPort capi3,
        int executionCount) {
            System.out.println("Reading entries with " + request);
            return AspectResult.OK;
    }

    public AspectResult postRead(ReadEntriesRequest<?> request,
        Transaction tx, SubTransaction stx, Capi3AspectPort capi3,
        int executionCount, List<Serializable> entries) {
            System.out.println("Read entries " + entries);
            return AspectResult.OK;
    }

    public AspectResult postWrite(WriteEntriesRequest request,
        Transaction tx, SubTransaction stx, Capi3AspectPort capi3,
        int executionCount) {
            System.out.println("Wrote entries " + request.getEntries());
            return AspectResult.OK;
    }
}
```

Container aspect that implements three ipoints

Let's look at the signature of the methods first. You can see a request (`ReadEntriesRequest`, `WriteEntriesRequest`), which contains the arguments of the Capi methods `read` and `write`, respectively. The next parameters are the internal transaction and sub-transaction.

In the space every transactional operation is performed in a sub-transaction inside of a transaction. The `Capi3AspectPort` is a special internal interface for a container for advanced aspect users. The last common parameter, the `executionCount`, tells how many times this request has been processed in the space (starting with 1 and including the current processing). It is incremented when the request processing starts and is bigger than one for a request that was blocked and rescheduled. Its use and the details of the sub-transaction and `Capi3AspectPort` are outside of the scope of this tutorial.

Depending on the ipoint, you also get the result of an operation as argument of the aspect method. For example, for a `read` operation you can access the entries in the aspect. Another important argument of the aspect methods is the `RequestContext`. It is “hidden” inside the request and contains a map that can be used to store key-value pairs. You can set the `RequestContext` in the Capi and use it to pass values to aspects.

You may change the `RequestContext` or entry lists (entries to write, selected entries) in the aspect methods.

Every aspect method returns an `AspectResult`. It contains a status value that can be OK, NOTOK, SKIP, DELAYABLE, or LOCKED. An aspect method should return OK for normal execution and NOTOK if an error occurs. A SKIP in a pre-operation method skips the following aspects at this ipoint and the space operation. A SKIP in a post-operation method skips the following aspects at this ipoint. DELAYABLE and LOCKED are for advanced use together with `Capi3AspectPort`. Runtime exceptions and `null` are treated as NOTOK. Section 5.4 explains the implications of the aspect status values on the execution of other aspects and the space operation itself.

5.3 Space Aspects

Space aspects are similar to container aspects. In fact, a space aspect is also a container aspect. Its methods are called for all containers and there are also ipoints for general space operations. The interface `ContainerAspect` extends `SpaceAspect` and has methods for the following additional ipoints:

- pre-create-container, post-create-container
- pre-lookup-container
- pre-create-transaction, post-create-transaction
- pre-prepare-transaction, post-prepare-transaction
- pre-commit-transaction, post-commit-transaction
- pre-rollback-transaction, post-rollback-transaction
- pre-shutdown

The `AbstractSpaceAspect` is an empty implementation that extends `AbstractContainerAspect`.

5.4 Aspect execution

Multiple aspects can be added for an ipoint. They are executed sequentially in the order they were added. For operations where container and space aspects can be registered this distinction is also relevant and the execution order is as follows:

1. Space aspects (pre-operation)
2. Container aspects (pre-operation)
3. Space operation
4. Container aspects (post-operation)
5. Space aspects (post-operation)

The result of an aspect method influences the execution of the following aspects and the space operation as follows:

- OK: the next aspect or the space operation is executed, or the result is returned
- NOTOK: the operations returns immediately with an error (exception may be passed)
- SKIP:
    - pre-operation aspect method: the following pre-operation aspects and the space operation are skipped
    - post-operation aspect method: the following aspects are skipped
- DELAYABLE, LOCKED: the operation returns immediately and is blocked (for advanced use with `Capi3AspectPort`)

Inside of aspect methods you may call `Capi` methods on the same or another space. You can create a `Capi` instance from the `MzsCore` instance passed in the initialization method. However, you should be careful with operations that block or may cause recursive aspect executions. Use timeouts to avoid (long) blocking. You can use flags/marker objects in the `RequestContext` to avoid or detect recursive aspect calls.

5.5 Notifications with Aspects

Notifications in XVSM allow you to get called when the state of the space changes. Currently only a simple variant that allows observing a container for changes is implemented. As central part it uses an aspect that writes special notification entries in the post-methods of the entry operations to a notification container. The notification container is on the same space as the observed container (server-side). The entries

in this container are taken on the peer where the notification is created (client-side) and a listener is called. To use the notifications a small additional API is provided. Internally only the existing space operations are used.

The main class of the notification API is `NotificationManager`. It can be used to create notifications and destroy all of its notifications when you want to stop your application. The signature of the simple method to create a notification is

```
Notification createNotification(
          ContainerReference container,
          NotificationLister listener,
          Operation... operations)
    throws MzsCoreException
```

The observed container is specified together with a listener and the container operations that should be observed. `NotificationListener` is an interface with only one method:

```
void entryOperationFinished(
          Notification source,
          Operation operation,
          List<? extends Serializable> entries)
    throws MzsCoreException
```

The `Operation` is an enum with fields for the entry operations: READ, TEST, TAKE, DELETE, and WRITE. The entries are a list of the read, tested, taken, deleted or written objects, either the application objects without coordination data or, for write, the `Entry` objects with coordination data.

The method to create a notification is overloaded, but passing a transaction reference or context is for advanced use outside of the scope of this tutorial.

There is a notification container for each notification and it is created with a FIFO coordinator. The notification entries are written with the transaction (and isolation level) of the entry operation that is intercepted. There are no further guarantees about the order of the notifications.

5.6 Notification Example TicketQueue

This example extends the very simple `TicketQueue` example from Section 3.6 that only demonstrates the FIFO coordinator. Now we have two different applications, `ShopAssistant` and `Customer`. The `ShopAssistant` has to be started first. It starts a space that listens for incoming connections on a specific port, creates a container

with the FIFO and Type coordinator that represents the sales desk and creates a notification for write operations on it.

The `Customer` looks up the sales desk container, registers a notification for write operations, and writes a payment object to the container. This notifies the `ShopAssistant` who takes the payment entry and issues a ticket for the customer. The correlation between the payment and the ticket is done with an ID. The ticket is written to the sales desk container, which triggers the notification listener in the `Customer`. The `Customer` takes the ticket and exits.

If you look at the code of the example (available on the website), you see several checks for the entry type and the ID of a customer that ensure that the correct ticket is taken from the sales desk container (`Customer`) and only payments are processed (`ShopAssistant`). This is because of the simple notification that only allows to filter on the container operation type and the possible race conditions when multiple customers concurrenctly access the sales desk container. To demonstrate the latter, the `ShopAssistant` can be called with an argument that specifies the number of customers that are started in separate threads directly after the start of the application. The output of the program is not always the same, it depends on the thread scheduling, but the chance is high that there are several tickets in the sales desk container at the same time and that a customer takes the wrong one (and gives it back afterwards). Of course, instead of taking a ticket and then checking the customer ID, the selection could be more specific (for example with a key, label or the use of Linda templates or the Query coordinator).

5.7 Summary

Aspects can be used to extend MozartSpaces. The existing space operations can be dynamically enhanced or replaced with own code in aspect methods at defined interception points. Aspects can be added to the whole space or a specific container.

Notifications allow you to get called when the contents of a container changes and are internally implemented with aspects.

Chapter 6: Advanced Features

This chapter explains the integrated persistence, how custom coordinators can be added, and other advanced configuration settings of MozartSpaces.

6.1 Persistence

Persistence for containers and their entries was added in Version 2.2. It uses the *Berkeley DB (BDB) Java Edition* library to store the data as key-value pairs on disk. No installation of BDB is necessary. In the MozartSpaces configuration a persistence profile for the whole space can be set. By default the profile `in-memory` is used, which does not persist the containers. For BDB three different profiles are available. They differ in the configuration of the BDB `SyncPolicy` used for durability of the database log when a transaction is committed:

- `berkeleydb-lazy`: the database log is asynchronously written to disk. Data loss can occur on application or system crash.
- `berkeleydb-transactional`: the database log is synchronously written to disk. Data loss can occur on system crash.
- `berkeleydb-transactional-sync`: the database log is synchronously written and flushed to disk

There is a trade-off between the performance and the durability guarantees of the different profiles. The *lazy* profile is the fastest but also least reliable, while the *transactional-sync* profile is much slower but guarantees durability. The *transactional* profile depends on the operating system and the file system to write the data from the file system buffers to disk after the transaction commit.
For all BDB profiles the transactions are mapped 1:1 from the MozartSpaces transactions to the BDB transactions. The isolation is still done by MozartSpaces in-memory, so the isolation level READ_UNCOMMITTED is used for BDB.
The location of the BDB database files can be configured and should be different for every MozartSpaces instance (`MzsCore` with embedded space). Furthermore, the size of the BDB internal cache can be configured.
Independent of BDB, the serializer to convert the objects to byte arrays for persistent storage and the size of an optional cache for serialized objects can be configured.

When persistence is enabled for the space you can create containers that are explicitely not persistent by setting the flag `forceInMemory` (`createContainer` method in `Capi`). Apart from that, the use of the API is independent from the persistence configuration. However, application programs may require small changes, for example when containers are restored from disk on start-up of the space and creating a container is not necessary or not possible (duplicate name).

6.2 Custom Coordinators

In addition to the pre-defined coordinators, own coordinators can be implemented and added with the configuration. The configuration is necessary to add a translator from the API classes to the implementation classes. The API classes for the coordinator and the selector are serialized and transported, while the implementation classes contain the internal references and collections. The API classes need to implement the interface `Coordinator` and `Selector` (package `org.mozartspaces.capi3`). The implementation classes need to implement the interface `NativeCoordinator` and `NativeSelector`. The translator classes implement the interfaces `NativeCoordinatorTranslator` and `NativeSelectorTranslator`.
The implementation of a custom coordinator is quite complex. Internally many cases (entry locked or not authorized, selector is the first in a selector chain or has a predecessor etc.) need to be distinguished. So try to start with an existing coordinator, copy it, and change the semantics as required.

6.3 Advanced Configuration

In addition to the settings explained in Section 2.3 (embedded space, receiver port, space URI) and the persistence and custom coordinator configuration described above, MozartSpaces allows further configuration. Some settings enable experimental features and in general the comments in the example configuration file should be read and followed.

6.3.1 Core Processor

The *number of threads for the core processor's* (XP) thread pool can be set. The XP processes the requests for the space operations, that is, every space operation is performed with one of these threads. This thread pool can also be deactivated. Then

the thread that provides the request (from the API or transport handler) is reused. This avoids the overhead for the thread context switch, but may affect the concurrency of the transport handler and does not work with asynchronous operations (`AsyncCapi`) on the embedded space.

6.3.2 Remoting Configuration

MozartSpaces supports multiple transport handlers. The scheme of the space URI is used to select the transport handler, and the default scheme is used for space URIs without a scheme part. Usually a single transport handler is enough. Currently there is only one well-tested transport handler, the TCP socket transport. It uses the global setting `bindHost`, to bind to all or a specific network interface. The `receiverPort` is used for the listener (or server) socket. The number of threads limits the maximal number of possible collections as two threads are used for each connection, one for the incoming connection and one for the outgoing connection. The `serializer` is used to serialize and deserialize the transferred messages.

6.3.3 Serializer

A serializer serializes objects to a byte array and deserializes objects from a byte array. It implements the interface `org.mozartspaces.core.util.Serializer`. Several serializers are pre-defined and can be referenced by their name/ID. Custom serializers can be added to the configuration with their class name.
Multiple serializers can be configured to be active at the same time. They can be referenced at the following locations of the configuration:

- TCP transport
- Persistence
- Entry Copier

Currently the following serializers are pre-defined:

- javabuiltin: uses the default binary serialization of Java
- jaxb: uses JAXB with an XML schema; the schema is incomplete, because the structure of the entries is not specified, and XStream[3] is used to serialize the

[3] http://xstream.codehaus.org/

objects in the “gaps”; some functionality is not supported by this serializer (customer coordinators, query coordinator, error handling etc.)

- kryo[4]: a fast binary serialization provided by a small library
- xstream-json: JSON serialization with XStream and its Jettison driver
- xstream-xml: schemaless XML serialization with XStream

6.3.4 Entry Copier

By default entries are not copied when they are written to or selected from the space. They are implicitly copied by serialization and deserialization when they are transported over the network. For operations on the embedded space an entry copier can be configured. It copies the entries, and optionally also the request context, so that there are no references from the entry objects in the application to the entry objects in a space container. Entries can be copied by a serializer or by the clone method of the interface `org.mozartspaces.util.MzsCloneable`.

6.3.5 Security

MozartSpaces has an integrated attribute-based authorization that uses the attributes, for example roles or user IDs, from an external identity provider that performs the authentication. This functionality is currently experimental and out of scope of this tutorial.

[4] http://code.google.com/p/kryo/