

XVSM Tutorial

Version 2.1

Written by Michael Wittmann & Bernhard Efler

Table of Contents

Chapter 1: Introduction	3
1.1 "Hello World!" – “Hello Space!”	5
Chapter 2: MozartSpaces Startup and Configuration	7
2.1 Startup	7
2.2 Configuration.....	7
Chapter 3: The Data structures	10
3.1 The Container, an introduction	10
3.2 Entry.....	14
3.3 Coordinating the contained things.....	15
3.4 Selectors and CoordinationData	25
3.4.1 VectorSelector, VectorCoordinationData.....	26
3.4.2 KeySelector, KeyCoordinationData	27
3.4.3 LabelSelector, LabelCoordinationData	27
3.4.4 LindaSelector, LindaCoordinationData	27
3.4.5 QuerySelector, QueryCoordinationData	27
3.5 Methods to access data	28
3.5.1 read	28
3.5.2 delete.....	29
3.5.3 take.....	30
3.5.4 write.....	30
3.6.1 Exercise: The Ticket Queue (FIFOCoordinator)	30
3.6.2 Exercise: Formula 1 Race (VectorCoordinator).....	31
Summary:.....	31
Chapter 4: Transactions	33
4.1 Exercise (Transactions):.....	35
Summary:.....	36
Chapter 5: A completely new aspect: Aspects in XVSM.....	37
5.1 Container Aspects.....	37
5.2 Space Aspects	40
Summary:.....	41
Chapter 6: Don't miss a thing: Notifications	42
6.1 Exercise (TicketQueue with Notifications).....	43
Summary:.....	44
Chapter 7: Bibliography	45

Chapter 1: Introduction

XVSM is standing for eXtensible Virtual Shared Memory, which is middleware technology to store data objects in a space that can be shared with other peers. Data objects are written to and read from that common data storage.

The space-based approach has some nice advantages. Many implementations of such an approach offer the possibility to distribute data over multiple computers, which can help to be more fault-tolerant, when one participating computer fails or leaves the space. In this case, the data is not necessarily lost as it may be stored on another computer. Moreover, when multiple computers can access the same data storage, they can cooperatively work on this data. For example, you can implement a system where a space participant that takes one entry of the space and performs some action on it, while another participant waits for the result of this calculation. The two participants can be separated from each other in physical space as well as in time. This means, that each participant can run on its own computer and not on the same one, and they don't need to run at the same time, because the calculation result remains in the data space after writing it. Thus, the space participant that waits for the calculation result can access this entry even when the participant that calculated it is no longer present. It is also possible to let multiple computers work on the same space to cooperatively and concurrently work on the set of data to improve the performance to handle the stored data objects.

XVSM in the current version offers the possibility to manage the data entries and the order and number of entries that are retrieved. For example, you can define a Coordinator to retrieve the entries in the opposite order they were written and another Coordinator to retrieve them in random order.

It also offers the possibility to perform actions within one transaction, this is to perform it as one atomic action that either executes successfully and thus the results are written to the space or when an error occurs, the whole transaction is rolled back and all changes are discarded. This reduces the danger that the data objects in the space are inconsistent after an error.

You also can register listeners on actions that are performed either on the data or on the space itself. If an action is performed that you are listening on, the corresponding

method to react on this event is called. This can be used to react for example when a data entry is added or removed. Such a technique can be used to enhance the functionality of the system, even during runtime, as these handlers can be registered and unregistered even while the system is running.

As a precondition to understand how to use XVSM and the terminology, it is recommended that you first read the Application scenarios document, which gives an overview about various programming techniques in Space-Based computing and in XVSM in particular. The tutorial helps you to start programming using MozartSpaces, which is an open source implementation of XVSM in Java. It is not intended that this tutorial is a complete documentation of the API, it only gives you advice how to program with XVSM. You can find the complete API reference at [1].

In the next sections, the start-up and configuration of MozartSpaces is explained and an introduction to the Container and the ready-to-use data structures that can be used within the container is given. Then, this document tries to show you how to select specific data in the space. After that, it shows you how to extend the functionality of XVSM by using Aspects. As a usage of Aspects, the usage of notifications is explained, which are used to tell the listening program if new data is available, data has changed or data has been deleted.

Please note that XVSM is prepared for various additional functions, including the distribution of the data over multiple peers, automatic data persistency, and many more. This tutorial uses the most recent Java implementation of XVSM, which is MozartSpaces version 2.0, provided by the Space Based Computing Group [2] at the Vienna University of Technology. In this version, these functions are not yet implemented. Thus the tutorial focuses on the functions that are working in this version, so you can try out all examples and exercises that are given in this tutorial.

Further please note that MozartSpaces only works with Java 6.0 or newer.

1.1 "Hello World!" – "Hello Space!"

The "Hello World!" example is nearly mandatory for each new programming language or paradigm. Thus also here, an adaptation of this example is described. The details of it are explained in the subsequent chapters.

The purpose of this example is to show the basics of the new paradigm and to have a fast overview of its usage. In the original example, the text "Hello World!" shall be printed on the screen. In the XVSM example, the text "Hello Space!" is used and as a further variation this text shall be written to a shared container from which it is read and printed on the screen:

```
public static void main(final String[] args)
{
    try
    {
        System.out.println();
        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);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

The `Capi` class offers you all methods to operate with MozartSpaces. For example, you use it to create or write to a container, or to access the Entries that are stored in a container. The details about containers and Entries are explained in Chapter 2.

When you have instantiated the `Capi` (Core Application Programming Interface) class, you can create a container. Then you write an Entry containing the String

"Hello, space!" to that container. Afterwards, this Entry is read again from the container and written to System.out.

Chapter 2: MozartSpaces Startup and Configuration

2.1 Startup

A standalone instance of MozartSpaces can be started using the complete binary distribution packages. class.

After downloading the file `mozartspaces-dist-2.0-<buildQualifier>-all-with-dependencies.jar` from the web page, MozartSpaces can be started using

```
java -jar MozartSpaces-dist-2.0-<buildQualifier>-all-with-dependencies.jar [port]
```

The parameter `port` is optional, defaults to 9876 and is described in the configuration section below. If this command is executed, MozartSpaces is started with the class `org.mozartspaces.core.Server`, which is set as the main class in the JAR file.

Furthermore, it is possible to use a specific configuration file by setting the according system property:

```
java -cp MozartSpaces-dist-2.0-<buildQualifier>-all-with-dependencies.jar -Dmozartspaces.configurationFile=<configfile> org.mozartspaces.core.Server
```

2.2 Configuration

MozartSpaces can be configured using an XML configuration file. MozartSpaces tries to locate the 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> ..."`
- The search order for the configuration file is (only relevant if only the file name or a relative path is specified):
 - current directory
 - user's home directory (`user.home`)
 - classpath

The most important options for this tutorial are `spaceURI`, `embeddedSpace` and `receiverPort`.

The receiver Port 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 by one space on the same machine.

With the `spaceURI` you specify the endpoint of your space. When you only want to connect to spaces running on your local machine, you can leave it like it is. If you want to use it on other machines you must set the `spaceURI` to a hostname/IP which is reachable by those remote peers. The `spaceURI` is even needed when you have no embedded space.

By using the `${remoting.transports.tcpsocket.receiverPort}` placeholder the `receiverPort` is added automatically.

The `embeddedSpace` option defines if a local space is started up when you create a new `MszCore` instance. This is useful when you use a dedicated server and don't want to start up a space at the clients.

A configuration file covering these options looks like this:

```
<mozartspacesCoreConfig>
  <!-- Determines whether the core has an embedded space or is just a thin
  client -->
  <embeddedSpace>true</embeddedSpace>

  <remoting>
    <transports>
      <tcpsocket scheme="xvsm">
        <receiverPort>9876</receiverPort>
      </tcpsocket>
    </transports>
  </remoting>

  <!-- Configuration of the Space URI, identifier and also locator of a
  space -->
```

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

The full configuration file can be found in the examples you can download from the web page.

It contains the default configuration. If you remove or comment out configuration parameters, their default values are still used, because they are also stored as constants in the code.

There is also the possibility to set the configuration parameters programmatically, see the Javadoc (DefaultMzsCore) for details.

MozartSpaces uses logback for logging, which has its own configuration. An example configuration file for logback, logback.xml, is included in the examples.

Chapter 3: The Data structures

There are several possibilities to read and write data from and to the XVSM space. As a precondition to understand what kind of data you'll be able to write to the space, it is necessary to show some basics about what the space actually is and how you can work with it. Many of the methods described in this tutorial are overloaded, so if you are interested in the full functionality of the space have a look at the Javadoc.

3.1 *The Container, an introduction*

The Container is the place where the data is stored in. You can visualize the container in such a way:

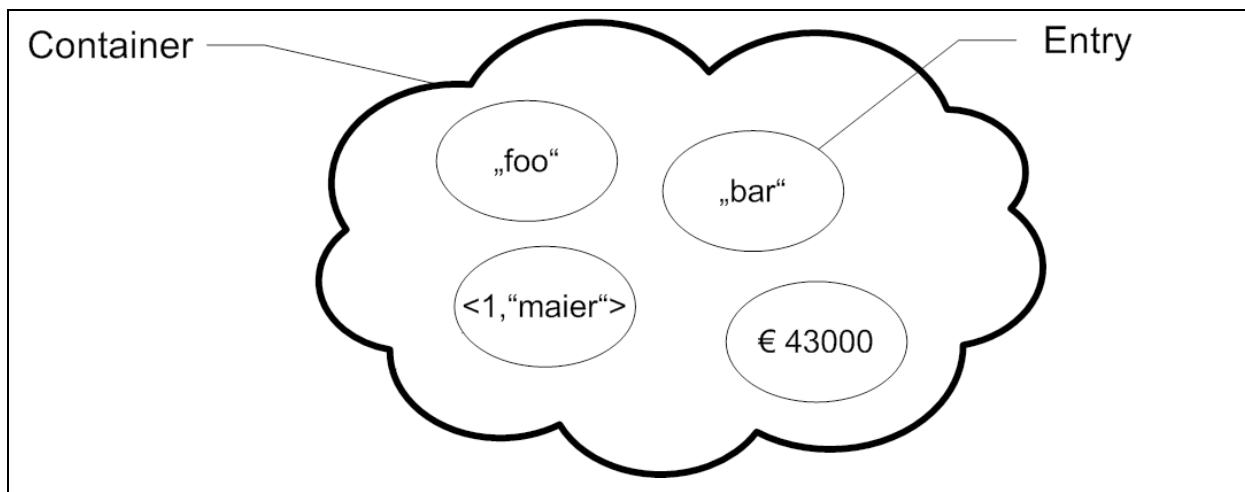
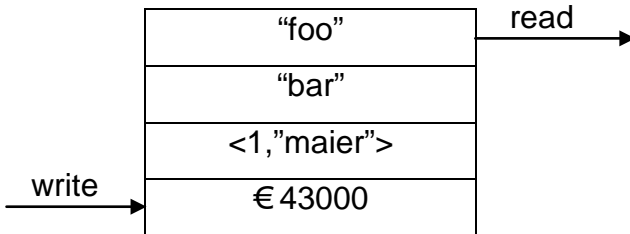


Figure 3.1: Overview of a container with objects

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:



The data items that are stored in a container are called "Entries". An Entry can be any serializable object.

A Container can either be located on your local machine or on a remote machine:

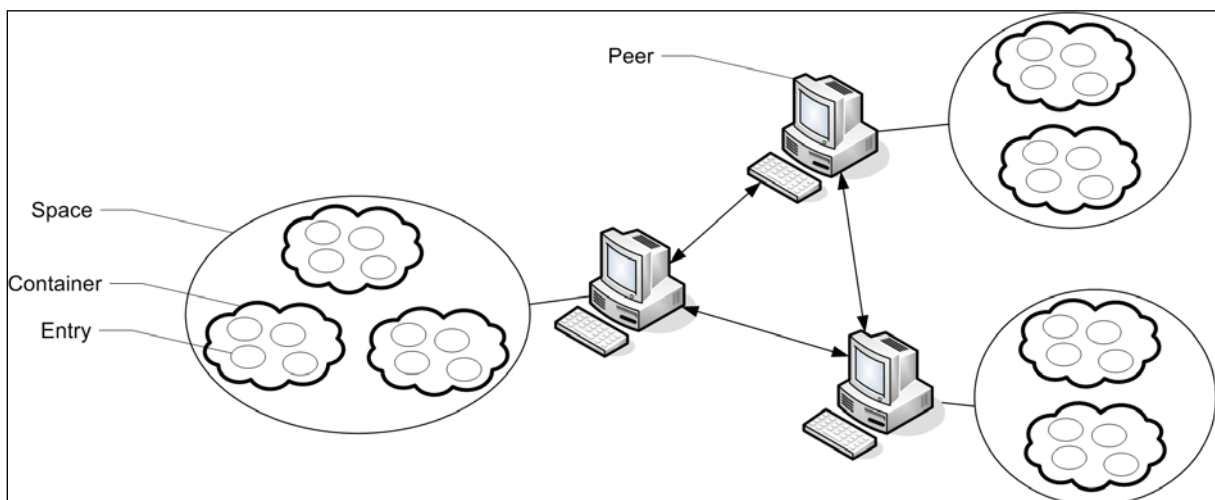


Figure 3.2: Communication of a set of peers, each with its own Space, Containers and Entries

The `MzsCore` class is the main management class of the containers and XVSM itself. For easier synchronous usage the `Capi` class is used. You can shutdown and restart the space, create and destroy containers, or read, write and delete Entries in a container. First of all, you start your local space by calling

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

Every time a new `MzsCore` instance is created on a computer, a local Space is created, which is accessible by other peers by default on port 9876. This port can be

changed via the configuration file described in Chapter 2.2. Another possibility to create a Space on a computer is to start the standalone Server, which you can find in the `org.mozartspaces.core` package and which has its own main method. It only starts a Space to handle the requests by other peers but doesn't offer an interface to handle the Space on its own, thus only other peers can manage the content of the Space. Keeping this in mind, there are two possibilities to start a new Space: by instantiating a new `MzsCore` by calling `DefaultMzsCore.newInstance()` or by starting a standalone Server.

To create a new container, use this method:

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

To create a container, you need to pass the following parameters:

- 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 Container's site: this is the URI to access spaces on a remote peer. By default, the port to connect to the remote peer is 9876, but of course the peer could use another port. The URI of a remote Server is constructed the following way: `new URI("xvsm://mycomputer.mydomain.com:9876")`. The default port can be changed in the `mozartspaces.xml` configuration file. Use null to connect to your local machine.
- Container's maximum size: The maximum number of entries that the container can hold. If you try to write an Entry 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 internal coordination of the container. The different types of Coordinators will be described in Chapter 3.3. Obligatory Coordinators must be added on write to each Entry.

- List of optional Coordinators: They are used like the obligatory Coordinators, but they are optional on write.
- Transaction: transactions are explained in chapter 4. Use null to perform the action with an implicit transaction (this means that the action is automatically committed)

The `createContainer` method returns a Container Reference, which is in turn used to refer to this container. In the later examples, you will see that it makes sense to use multiple Container References. The Container Reference is passed as parameter in most of the methods that are used to access data.

If you know that the container already exists or a `ContainerNameNotAvailableException` is thrown on the creation, you can use the following method to get access to an existing Container:

```
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`. This way, you can access a container which already exists in a space.

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 XVSM-specific Exceptions and also stands for an Exception that is thrown because of an internal error.

When you'd like to shut down the Capi peer, you can pass the URI where the peer is running. Thus, you can smoothly turn off a remote peer, which makes sense especially for the standalone Server:

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

Keep in mind that in the current implementation, no security mechanisms exist that prevent the shutdown of a peer. Security mechanisms to prevent unauthorized users from performing these actions will be implemented in future releases of MozartSpaces.

Further note that currently no persistency is implemented, so if you stop your peer, all information is lost.

3.2 Entry

As already described, an Entry can be everything that implements the `Serializeable` interface.

In MozartSpaces you can handle these objects directly, except for the write method. To be able to handle different Coordinators, each Entry has to be encapsulated in an Entry object.

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

The second parameter is a list of Coordination Data which are required for the Coordinators.

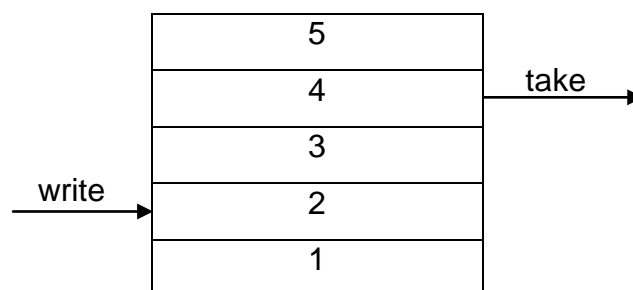
3.3 Coordinating the contained things

In a space, entries can be organized by so called coordinators. These coordinators are registered to containers and take care of all further entry organization. This gives you the advantage to choose the coordination pattern you need, instead of implementing it in the application. MozartSpaces provides a set of pre-defined coordinators like FIFO or KEY but also allows the developer to introduce user-written coordinators. Every coordinator provides a so-called Selector and Coordination Data. The Selector is responsible for selecting a specific result set from the container. For example, a LABEL Selector would be used to select entries with a specific label from the container. The Coordination Data is used when writing entries to a container, to present the coordinator the necessary book-keeping information (e.g.: the label to associate this entry to). [4]

Optional you can give each Coordinator a name, in case you want to use the same Coordinator more than once in a Container.

In the current version, the following possibilities exist to coordinate the order of the retrieved Entries:

AnyCoordinator: This is the standard Coordinator (if you don't use a Coordinator at all). It is used by default when you don't pass a Selector for reading, deleting etc. of entries (see Chapter 3.4). The entries are returned in a not specified way. It works like the Random Coordinator, but doesn't guarantee real randomisation and so it's faster than the Random Coordinator.



```

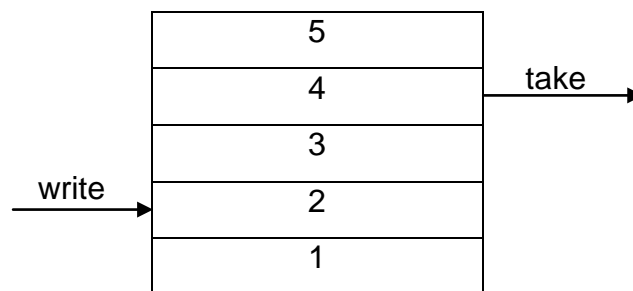
// Create new Container using no transaction, no remote server/peer,
// container-size of 10 and an AnyCoordinator
ContainerReference cref = capi.createContainer(null, null, 10,
        Arrays.asList(new AnyCoordinator()),
        null,
        null);

/* Fill the queue with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new Entry(i, AnyCoordinator.newCoordinationData());
    capi.write(cref, 0, null, entry);
}
// read and print one entry
ArrayList<Integer> readEntries = capi.read(cref,
        Arrays.asList(AnyCoordinator.newSelector()),
        0, null);
System.out.println(readEntries.get(0));

```

Listing 3.3.1: AnyCoordinator (default)

RandomCoordinator: The entries are returned in a real random way. You can compare this to a bag with lottery numbers:



```

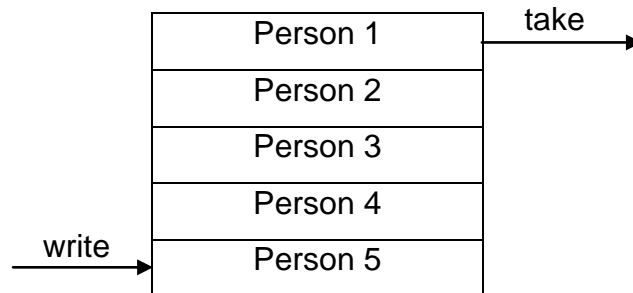
// Create new Container using no transaction, no remote server/peer,
// container-size of 10 and a RandomCoordinator
ContainerReference cref = capi.createContainer(null, null, 10,
        Arrays.asList(new RandomCoordinator()),
        null,
        null);

/* Fill the bag with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new Entry(i, RandomCoordinator.newCoordinationData());
    capi.write(cref, 0, null, entry);
}
// read and print on random entry
ArrayList<Integer> readEntries = capi.read(cref,
        Arrays.asList(RandomCoordinator.newSelector()),
        0, null);
System.out.println(readEntries.get(0));

```

Listing 3.3.2: RandomCoordinator

FifoCoordinator: If you use the `FifoCoordinator` (FIFO stands for first-in-first-out), the entries are retrieved in the same order as you wrote them to the container. This means, if you write 5 entries to the container, then you call the `read(cref, fifoSel, timeout, transaction)` method, the entry that you wrote FIRST is returned, thus the name “first-in-first-out”. You can visualize this like a queue of people standing in front of the vendor’s desk.

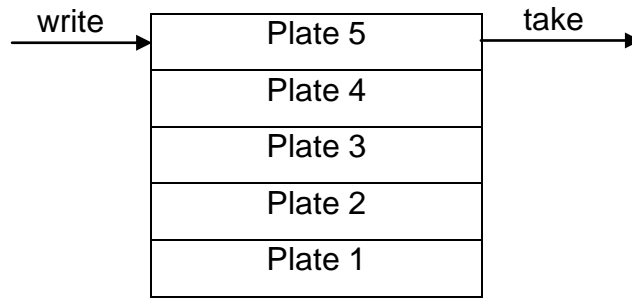


```
// Create new Container using no transaction, no remote server/peer,
// container-size of 10 and a FifoCoordinator
ContainerReference cref = capi.createContainer(null, null, 10,
    Arrays.asList(new FifoCoordinator()),
    null,
    null);

/* Fill the queue with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new Entry(i, FifoCoordinator.newCoordinationData());
    capi.write(cref, 0, null, entry);
}
// read and print the first entry (1)
ArrayList<Integer> readEntries = capi.read(cref,
    Arrays.asList(FifoCoordinator.newSelector()),
    0, null);
System.out.println(readEntries.get(0));
```

Listing 3.3.3: `FifoCoordinator`

LifoCoordinator: If the container uses the `LifoCoordinator` (LIFO stands for last-in-first-out), the entries are retrieved in the reverse order you wrote them. Thus, if you write 5 entries and then call the `read(cref, lifoSel, timeout, transaction)` method, the entry that was written LAST will be returned. You can compare this to a stack of plates – The last one you put on the stack will be the first one that you take from that stack.

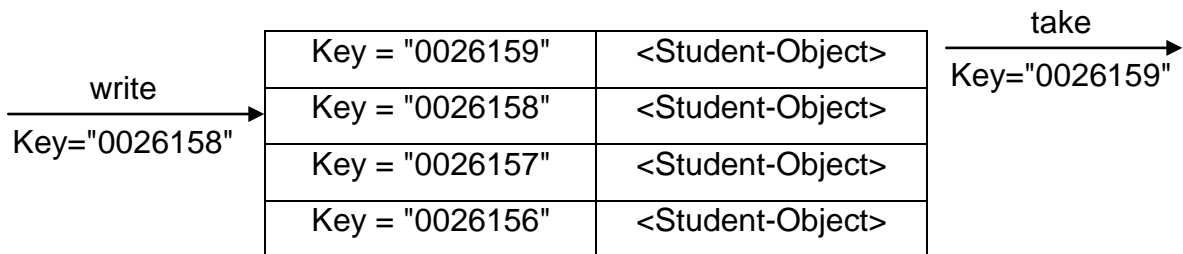


```
// Create new Container using no transaction, no remote server/peer,
// container-size of 10 and a LifoCoordinator
ContainerReference cref = capi.createContainer(null, null, 10,
    Arrays.asList(new LifoCoordinator()),
    null,
    null);

/* Fill the queue with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new Entry(i, LifoCoordinator.newCoordinationData());
    capi.write(cref, 0, null, entry);
}
// read and print the first entry (1)
ArrayList<Integer> readEntries = capi.read(cref,
    Arrays.asList(LifoCoordinator.newSelector()),
    0, null);
System.out.println(readEntries.get(0));
```

Listing 3.3.4: LifoCoordinator

KeyCoordinator: When you use the KeyCoordinator, you can define a key when you write an entry. When you want to find this entry again, you use the same key to get exactly this entry. So, for example, you have an entry which holds an object of type Student, which has several properties. When you write the entry to the container, you use the Student's matriculation number as key. When you later want to find this Student in the container, you need to use the KeySelector together with the Student's matriculation number, which is the key. The key must be a String.



```

/* Create new Container using no transaction, no name, an infinite */
/* container-size and KeyCoordinator */
ContainerReference cref = capi.createContainer(null, null,
    MzsConstants.Container.UNBOUNDED,
    Arrays.asList(new KeyCoordinator()),
    null,
    null);

/* Create a Student instance and write it into the container */
Student writeStudent = new Student(1000, "Max", "Muster", 20);
Entry entry = new Entry(writeStudent,
    KeyCoordinator.newCoordinationData(writeStudent.get_Surname()));

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

/* Get entry using KeySelector from the container */
ArrayList<Serializable> readEntries = capi.read(cref,
    Arrays.asList(KeyCoordinator.newSelector("Muster")),
    0, null);

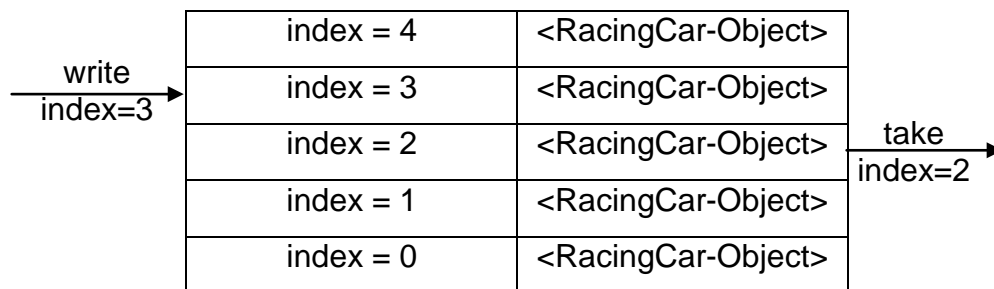
if (readEntries.get(0) instanceof Student)
{
    Student readStudent = (Student) readEntries.get(0);
    System.out.println("MatNr.: " + readStudent.get_MatNr());
    System.out.println("Forename: " + readStudent.get_Forename());
    System.out.println("Surname: " + readStudent.get_Surname());
    System.out.println("Age: " + readStudent.get_Age());
}

```

Listing 3.3.5: KeyCoordinator

LabelCoordinator: With the Label-Coordinator, entries are accessible by labels. This approach is quite similar to the KeyCoordinator, however, labels do not have to be unique, in contrast to keys. This allows multiple entries to have the same label.

VectorCoordinator: The VectorCoordinator is used when you want to address the Entries by index. You can compare this with the position of Racing cars.



```

/* Create new Container using no transaction, no name, an infinite */
/* container-size and VectorCoordinator */
ContainerReference cref = capi.createContainer(null, null,
    MzsConstants.Container.UNBOUNDED,
    Arrays.asList(new VectorCoordinator()),
    null,
    null);

```

```

for (int i = 1; i <= 5; i++)
{
    Entry entry = new Entry(i, VectorCoordinator.newCoordinationData(i);
    capi.write(cref, 0, null, entry);
}

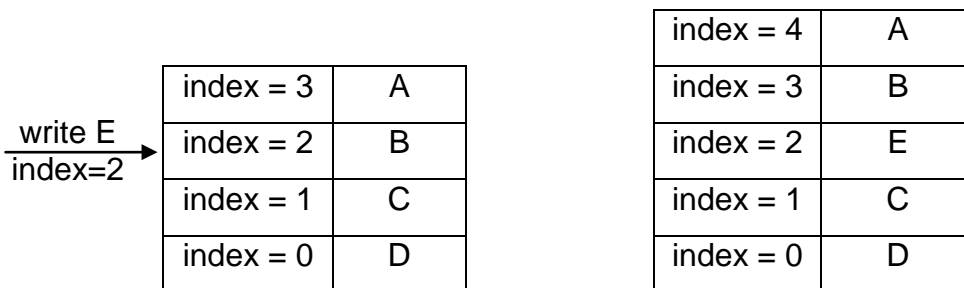
/* Get entry using VectorSelector from the container */
ArrayList<Integer> readEntry = capi.read(cref,
    Arrays.asList(VectorCoordinator.newSelector(3)),
    0, null);

System.out.println(readEntry);

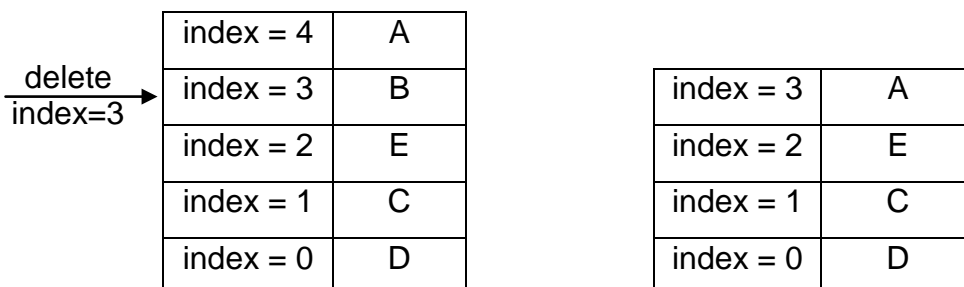
```

Listing 3.3.6: VectorCoordinator

One property of the VectorCoordinator which might be a bit tricky for beginners is that when you write an Entry to an index where another Entry already resides, this existing Entry is moved by one (its index is automatically incremented). All Entries subsequent indexes are also incremented by 1. You can imagine this like pushing all Entries with higher indexes one position further:



Note that B was stored with index 2, but now it is stored with index 3. On the other hand, when you delete an Entry from a container that is coordinated by a VectorCoordinator, the indexes of the Entries that have a higher index than the deleted one are decremented by one:



The Entry A was stored with index 4, after the delete command, it was moved to index 3. Thus you need to keep in mind that the indexes of Entries may be changed when adding/removing Entries.

LindaCoordinator: This is the LINDA Template Matching coordinator. This coordinator matches the template supplied by the selector with the entries registered to the coordinator.

When reading entries via a LindaCoordinator, you generate a template object of the same type as the entries you want to retrieve. Desired values are assigned to attributes, which must match the corresponding attribute values of objects stored in the container. Also, wildcards are possible, which are indicated by setting a field value of the template object to null. Thus, entries with arbitrary values for that field can be selected.

As MozartSpaces accepts standard Java Serializable objects, some kind of transformation has to be done. For this purpose, two special Java annotations are introduced. The @Queryable annotation can be set on classes and signals the space that this object should be queryable, i.e. usable for the Linda- and QueryCoordinator. The @Index annotation can be set on fields which should be accessible by a Query. All fields that are not indexed are ignored by the coordinator. Consider the following object, where only the title and duration properties can be accessed by the coordinator.

```
@Queryable
public class Movie {

    @Index
    private String title; // = "The Godfather"

    private List<String> cast; // = { "Marlon Brando" , "Al Pacino" }

    @Index
    private int duration; // = 175

    public Movie(String title, List<String> cast, int duration) {.....}
}
```

Listing 3.3.7 annotation example

By default, the field name is used as label, but the `@Index` annotation can also be used to overwrite the label. This is necessary, as the XVSM model allows duplicate labels, but Java does not allow duplicate field names.

```
@Queryable
public class Movie {

    @Index
    private String title; // = "The Godfather"

    @Index (label = "cast")
    private String cast1; // = "Marlon Brando"

    @Index (label = "cast")
    private String cast2; // = "Al Pacino"

    @Index (label = "cast")
    private String cast3; // = "James Caan"

    private int duration;
}
```

Listing 3.3.8 annotation example

Collections are interpreted as multi-sets whereas lists are interpreted as sequences, both containing elements with empty labels.

```
@Queryable
public class Movie {

    @Index
    private String title; // = "The Godfather"

    @Index
    private Collection<String> cast;
        // = {"Marlon Brando ", "Al Pacino", "James Caan"}

    private int duration ;
}
```

Listing 3.3.9 annotation example

The `@Queryable` annotation also gives the possibility to automatically index all fields with their variable name:

```
@Queryable( autoindex=true )
public class Movie {

    private String title; // = "The Godfather"

    private List<String> cast; // = { "Marlon Brando" , "Al Pacino" }

    private int duration ;
}
```

Listing 3.3.10 annotation example

The current implementation of the LindaCoordinator only allows matching objects of the same Java class.

```
/* Create new Container using no transaction, no name, an infinite */
/* container-size and LindaCoordinator */
ContainerReference cref = capi.createContainer(null, null,
    MzsConstants.Container.UNBOUNDED,
    Arrays.asList(new LindaCoordinator()), null, null);

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

movie = new Movie("Shrek", Arrays.asList(
    "Mike Myers", "Cameron Diaz"), 87);
entry = new Entry(movie, LindaCoordinator.newCoordinationData());
capi.write(cref, 0, null, entry);

Movie template = new Movie("Shrek", null, null);

/* Get entry using LindaSelector from the container */
ArrayList<Integer> readEntry = capi.read(cref,
    Arrays.asList(LindaCoordinator.newSelector(template)),
    0, null);

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

Listing 3.3.11: LindaCoordinator

QueryCoordinator: The Query Coordinator uses the same annotations as the Linda Coordinator. As the use of a string-based query language is antique, in contrast to other Java technologies, an object-based query language is introduced:

The root-object is the Query object, which consists of a list of filters (which are equal to sub queries). Each filter can either be an atomic function like reverse or consist of an arbitrary number of matchmakers.

Query

The Query object provides the following methods:

- *cnt(n)*: Takes n entries of the input.
- *reverse*: Reverse the input sequence.
- *sortup(Property)*: Sort the input sequence up by the supplied property.
- *sortdown(Property)*: Sort the input sequence down.
- *filter(Matchmakers)*: Build a XVSM query by matchmakers.

Matchmakers

Classical, selecting queries are built by the use of Matchmakers in the object-based Query. Each matchmaker can evaluate to true or false. The following Matchmakers exist:

- *and(Matchmakers)*: Logical conjunct an arbitrary number of Matchmakers. This is the representation of the logical function \wedge .
- *or(Matchmakers)*: Logical disjunct an arbitrary number of Matchmakers. This is the representation of the logical function \vee .
- *not(Matchmaker)*: Negate a Matchmaker. Represents \neg .
- *equalTo(Object)*: Checks equality of an Property to an object. Represents $=$.
- *notEqualTo(Object)*: Checks equality of a Property to an object. Represents \neq .
- *exists()*: Checks the existence of a Property on an entry.
- *greaterThan(Object)*: Checks if a property is greater than the object. Represents $>$.
- *greaterThanOrEqualTo(Object)*: Checks if a property is greater than or equal to the object. Represents \geq .
- *lessThan(Object)*: Checks if a property is lesser than the object. Represents $<$.
- *lessThanOrEqualTo(Object)*: Checks if a property is lesser than or equal to the object. Represents \leq .

Property

Two shapes of properties exist: ComparableProperties which are properties that are comparable by the $>$, \geq , $<$, \leq methods and normal Properties which are not comparable. Properties can be created with a static method of the Property class:

- *forName(String...)*: The argument is the path of the property, which specifies a field in an entry object whose value is used for evaluating the query. An asterisk can be used as a wildcard for any path element. To access the fields of an entry, the path must be preceded with such an asterisk.

Example Queries

```
Property publication = Property.forName("*", "publication", "by");
Property date = Property.forName( "*", "published", "date" );
Query test = new Query().filter(
    publication.equalTo( "Warner Bros." )
).sortup(date);
```

Listing 3.3.12 Query example

```
Property cast = Property.forName("*", "cast");
Query test = new Query().filter(
    Matchmakers.and(
        cast.equalTo("Al Pacino"),
        cast.equalTo("Marlon Brando")
    )
);
```

Listing 3.3.13 Query example

```
/* Create new Container using no transaction, no name, an infinite */
/* container-size and QueryCoordinator */
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, QueryCoordinator.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 LindaSelector from the container */
ArrayList<Integer> readEntry = capi.read(cref,
    Arrays.asList(QueryCoordinator.newSelector(query)),
    0, null);

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

Listing 3.3.14: QueryCoordinator

3.4 Selectors and CoordinationData

The Selector is the counterpart to the Coordinator. It tells additional information to the Coordinator when you want to read or delete data, like the key value that you're searching for or the number of values that you'd like to retrieve.

Coordination Data are used for passing needed information to a Coordinator for write operations.

For each coordination type, there exists the appropriate Selector and Coordination Data.

Each Selector has a count parameter. The count parameter is the number of entries that you would like to retrieve using this Selector. If you leave away the count parameter, one entry is returned. If you would like to have all entries of the container that fulfil the selection, use `MzsConstants.Selecting.COUNT_ALL` as count parameter.

With `MzsConstants.Selecting.COUNT_MAX` all unlocked entries are returned.

Some Selectors only offer the constructors mentioned above, as the sequence of the retrieved entries is given implicitly (remember, these selectors are: FIFO, LIFO, Any, Random).

Other selectors on the other hand offer a bit more information to the Coordinator, therefore, they are explained separately.

3.4.1 VectorSelector, VectorCoordinationData

```
VectorCoordinator.newSelector(int index)
```

```
VectorCoordinator.newCoordinationData(int index)
```

If you want to append the Entry at the end of the vector use the `VectorCoordinator.APPEND` constant as index.

The VectorSelector has the additional parameter index, which indicates the index of the entry in the container that you would like to access. Note that if you write an Entry to the container, the numbers of the other entries with higher and equal numbers are shifted by one, which means that each index of an Entry that has a higher index is incremented by 1.

For example:

```
Entry entry = new Entry(  
    "The first one",  
    new VectorCoordinator.newCoordinationData(0));
```

3.4.2 KeySelector, KeyCoordinationData

```
KeyCoordinator.newSelector(String key)
```

```
KeyCoordinator.newCoordinationData(String key)
```

The KeySelector uses Strings as keys.

If you want to write an Entry to the container with a key that already exists, an exception is thrown.

3.4.3 LabelSelector, LabelCoordinationData

```
LabelCoordinator.newSelector(String label)
```

```
LabelCoordinator.newSelector(String label, int count)
```

```
LabelCoordinator.newCoordinationData(String label)
```

The LabelSelector uses Strings as keys.

It is quite similar to the KeySelector, however, labels do not have to be unique.

3.4.4 LindaSelector, LindaCoordinationData

```
LindaCoordinator.newSelector(Serializable template)
```

```
LindaCoordinator.newSelector(Serializable template, int count)
```

```
LindaCoordinator.newCoordinationData()
```

The LindaSelector uses an instance of the searched object as a template. Each annotated field of an entry is compared to the corresponding template field for equality. Null values are ignored.

3.4.5 QuerySelector, QueryCoordinationData

```
QueryCoordinator.newSelector(Query query)
```

```
QueryCoordinator.newSelector(Query query, int count)
```

```
QueryCoordinator.newCoordinationData()
```

The QuerySelector uses the passed query for finding matching entries.

3.5 Methods to access data

In this section, the used ContainerReference objects are the same instances that you got when you created the container. For now, ignore the Transaction parameter, when programming, you can use null instead of the tx. Transactions will be explained in chapter 4.

3.5.1 read

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

The read method returns a number of Entries from the container, without deleting them. You need to use a Selector that is corresponding to the Coordinator. If you leave away the Selector parameter, RandomSelector is used. The Selectors can have a count parameter in the constructor, which tells the container the number of entries that you'd like to retrieve (see chapter 3.4 about Selectors). Thus, the return value of the read is an ArrayList of serializable objects, because you can define a Selector to read and return multiple Entries at once. If you don't use the parameter to tell the number of entries that should be read, you get one entry (if available).

One important feature that can be used for synchronization is the fact, that if there is currently no entry in the container that can be retrieved, and the count parameter is not COUNT_MAX or COUNT_ALL, the read method blocks and waits for an Entry. And that's the reason why there is a timeout parameter: It tells the read method to wait at least the timeout in milliseconds. So giving a timeout of `MzsConstants.RequestTimeout.TRY_ONCE`, you have a non-blocking read method call – it either reads an entry or if there is none, it immediately returns an Exception. `TRY_ONCE` waits for locked Entries. If you don't want this behaviour use `MzsConstants.RequestTimeout.ZERO`. The opposite would be to use the `MzsConstants.RequestTimeout.INFINITE`, which waits forever, if no one ever writes a suitable entry to the container.

```

//Create new Container using no transaction, container-size of 5 and a
//FifoCoordinator
ContainerReference cref = capi.createContainer(null, null, null,
    5, new FifoCoordinator());

/* Fill the queue with the numbers 1 to 5 */
for (int i = 1; i <= 5; i++)
{
    Entry entry = new Entry (i, FifoCoordinator.newCoordinationData());
    capi.write(cref, 0, null, entry);
}
ArrayList<Integer> readEntries = capi.read(cref, 0, null, new FifoSelector());
/* ... prints 1 */
System.out.println((readEntries.get(0)));

/* Read from the container with FifoSelector of size 2 */
readEntries = capi.read(cref, 0, null, new FifoSelector(2));
/* ... prints 1 */
System.out.println((readEntries.get(0)));
/* ... prints 2 */
System.out.println((readEntries.get(1)));

/* Read from the container with timeout */
readEntries = capi.read(cref, 5, null, FifoCoordinator.newSelector());
/* ... prints 1 */
System.out.println((readEntries.get(0)));

```

Listing 3.5.1: Reading entries using a FifoSelector. One time with size = 2 and the second time without size given. Also one read with a Timeout given and one without. As transaction, null is used.

3.5.2 delete

```

void delete(ContainerReference container,
            List<? extends Selector> selectors,
            long timeoutInMilliseconds,
            TransactionReference transaction)
    throws MzsCoreException

```

The delete method adheres to the same rules as the read method. Depending on the container's Coordination type, an entry is deleted, but without returning the deleted entry. Note that also, when deleting entries, the method blocks until a suitable entry is found in the container. One might think that it is not very meaningful to wait for an entry just to immediately delete it, but even though there are better techniques to perform this task, one might use this technique to notify a single listener. The listener calls `delete(cref, MzsConstants.RequestTimeout.INFINITE)` and thus is blocked. The notifier writes an entry to the container and doing so, the entry is deleted and the listener is unblocked. As already said, there are better possibilities in XVSM to perform such tasks – in most cases, you will have entries when you call delete, so this method will not block.

3.5.3 take

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

This method is the same as first calling read and then delete: The read Entry is immediately deleted after reading in one atomic step. This functionality is also commonly referred to as “consuming read”. Also the take method could block and thus has a timeout parameter.

3.5.4 write

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

By using this method, you can write one or more entries to the space. Also here, you use the ContainerReference object to refer to the corresponding container. When the container’s size is limited, and there are already as much entries in the container as allowed (the number of entries in the container is equal to the maximum capacity), and you want to write further entries to the container, this operation blocks. The write operation waits until an entry is taken or destroyed or – if given – until the timeout is reached. You have to use one or more Coordination Data objects together with a specific Entry (e.g. a KeyCoordinationData object).

Some Coordinators are implicit Coordinators. That means if a Container has implicit Coordinators assigned they are automatically attached to a write operation.

At the moment, implicit Coordinators are: AnyCoordinator, RandomCoordinator, FifoCoordinator, LifoCoordinator, LindaCoordinator, QueryCoordinator.

3.6.1 Exercise: The Ticket Queue (FIFOCoordinator)

A bunch of people is standing in a queue to buy cinema tickets. One person after the other is served by the shop assistant. A person who wants to buy a ticket must pay

and wait for the ticket to be printed. This takes some time (we assume that this is a quite fast action and takes 2 seconds in total).

Please program a Capi peer that handles the persons in the queue one after the other. As person objects, use an extra class "Person" that has a payTicket() and a waitForTicket() method. Both methods only wait for 1 second. In later examples, this exercise will be enhanced. The shop assistant only gets the first person in the queue, receives the payment and prints the ticket. Afterwards, the person object in the queue leaves the queue. You don't need to have a shop assistant class, just write it in the main method of the main class.

3.6.2 Exercise: Formula 1 Race (VectorCoordinator)

You're the IT professional in managing the Race results of a Formula 1 Race. You have the task to implement a piece of software (using XVSM of course!) to show the current race position. The values of the entries are RacingCar objects, whereas they have the property "driverName". When someone accesses the container to retrieve the current status of the race, of course, all cars should be listed together with their current position. For simplicity, just write 10 cars to the container. In the first round, car number 6 crashes and thus is removed from the race. In the second round, car 4 overtakes car 3. As this is a short race, let's say the race is over after this lap. List the current racing positions at the end of each lap. Also this example will be improved later. Each Racing car is running on a separate Peer (thus, one peer creates and holds the container, whereas all the other peers are connecting and using this container then).

Summary:

In this chapter, you've learnt the most important basics to handle a container and a container's content. The basic access to the container is done using the Capi class, which offers methods to create, lookup and destroy a container and to read, write, delete and take entries from/to the container.

You also learnt how to set the internal coordination of the container and how to select specific data in the container.

A typical use case scenario is that you would like to create a new container and pass it the internal coordination type. Then, you'd like to create some entries and set the information in the entries. Afterwards, you'll write the entries to the container and then read or take them from the container. At the end, most probably you'll shutdown the Capi to unlink from the container.

Chapter 4: Transactions

We assume that most programmers are familiar with the concept of transactions: A transaction is helping to collect various actions to perform them as one single, atomic step to avoid consistency problems of the data storage. In the XVSM implementation, the transaction is performed as a pessimistic transaction; this means a transaction builds up locks on the data.

The lifecycle of a transaction from the user's perspective is as follows:

- Create a new transaction
- Perform various actions
- If everything went fine, commit the transaction to write the changes to the data storage
- If there was an error, rollback the changes to discard all actions you made.

In XVSM, if you start a transaction and then only perform read operations, read locks may be obtained (dependent on the applied isolation level). But as soon as you perform an operation that changes the content of a container (write, take, delete), the accessed Entries are locked exclusively for this one transaction. Depending on the involved selectors, also whole coordinators can be locked exclusively in a container, which means that other operations using these coordinators on the container with a different transaction may be blocked if required by the coordinator semantics. For example, the FifoCoordinator will lock itself on a container in this case, because the sequence the Entries will be returned in must remain the same during the transaction, whereas using a RandomCoordinator, the locks will be obtained only on the accessed Entries. For each entry, there are multiple read locks allowed, but only one write lock at a time. All other application parts that would like to access the entry but don't use this transaction are blocked until the transaction is committed or rolled back. Note that a single transaction can be used for operations on several containers in a space.

The current default isolation level is REPEATABLE_READ, which sets write locks as well as read locks for entries. Thus, deleted but not yet committed entries cannot be read and read entries cannot be deleted until the transaction that issued the read operation has committed. However, if you set the isolation level parameter in the overloaded versions of the read/write/take/... methods to READ_COMMITTED, read locks will not be set any more. This allows transactions to read already deleted entries if the delete has not yet been committed. In some cases, this behavior may be more feasible than using the default isolation level.

You create a transaction by calling

```
TransactionReference tx =  
    capi.createTransaction((long)timeout, (URI)site);
```

As you remember from the previous chapter, this Transaction object is passed as parameter in various operations (read, delete, write,...). So, if you call the write operation with the Transaction tx, the corresponding part of the container may be locked exclusively for this transaction (depending on the coordinator). Concurrent access to the container that is in conflict with the write operation will be blocked. Only the methods that use Transaction tx are executed without blocking. One thing that you need to keep in mind: if you don't set a transaction (you use null as parameter), an implicit transaction is created. This means, internally, a transaction object is created – thus, also here, concurrent actions run isolated from each other, although the action is committed automatically if successful.

The site parameter is the URI where the container is located at (see Chapter 3.1) and the timeout parameter can give a maximum validity time of the Transaction in milliseconds. If the given time has elapsed, all changes within this Transaction are rolled back and all further actions with this Transaction are rejected.

You commit the transaction by calling:

```
capi.commitTransaction(tx);
```

This means all your changes are applied to the space and all locked objects are unlocked. After committing, the transaction object no longer is valid.

You rollback the changes in this transaction by calling:

```
capi.rollbackTransaction(tx);
```

This means that the actions that you've done so far are discarded. A rollback might be intended if some error occurred, so most probably, the rollback statement will be standing in an error handling part.

```
MzsCore core = DefaultMzsCore.newInstance();
Capi capi = new Capi(core);
tx = capi.createTransaction(MzsConstants.TransactionTimeout.INFINITE, null);
try {

    /* Create new Container using a container-size of 3 */
    ContainerReference cref = capi.createContainer(null, null, 3,
        Arrays.asList(new FifoCoordinator()), null, null);

    /* Write 3 entries to the container */
    /* After the first write action the container is locked */
    for (int i = 1; i <= 3; i++) {
        Entry entry = new Entry(i, FifoCoordinator.newCoordinationData());
        capi.write(cref, MzsConstants.RequestTimeout.INFINITE, tx, entry);
    }
    capi.commitTransaction(tx);
    /* The container is not locked anymore */

    tx = capi.createTransaction(MzsConstants.TransactionTimeout.INFINITE, null);
    capi.read(cref, FifoCoordinator.newSelector(3), 0, tx);
    /* The container is still not locked */
    capi.delete(cref, FifoCoordinator.newSelector(1), 0, tx);
    /* The container is locked now */

    throw new Exception();
} catch (Exception e) {
    try {
        if (tx != null){
            capi.rollbackTransaction(tx);
        }
        /* The container is not locked */
    }
}
```

Listing 4.1: Transaction handling

4.1 Exercise (Transactions):

Extend the exercise 2.6.2 (Formula 1):

We'd like to let some cars run for a random amount of time. So we'd start a Capi peer for each car, let it wait for some time and then re-register in the container within a transaction to have a consistent container content. As this is a bit tricky, we'll give you a guide how to solve this exercise:

1. Add the properties "runtime" and "lapNumber" to each car.
2. For each car, start a new Capi peer. When having started it, let the peer wait for a random time between 3 and 5 seconds:

```
Random rnd = new Random();
int time = rnd.nextInt(2) + 3; //(random between 0 and 2) plus 3
```

3. If the Time is up, start a transaction.
4. Add the time that you've waited to your runtime.
5. Remove the car that actually has the transaction from the container.
6. Read all actual positions of the cars, and get the runtime for each car. If you've found a runtime that is higher, whereas the next runtime is lower than your runtime, then this is your position.
7. Write the car object to the new position (remember that the entries in the vector with higher index than the one that you use will be shifted down automatically). Don't forget that the car only needs to take 2 rounds, but of course you can use more rounds – you just need to keep the number of rounds for each car the same ☺ ...
8. Every time a car reaches the finish line, write the actual list of race positions together with the runtimes. Don't forget that cars that already have passed the finish line don't change their racing time any more.

Summary:

Transactions are fundamental if there are many peers (or threads) that use containers to avoid that the content becomes inconsistent through concurrent write access. A transaction helps you to keep the space consistent, as it prevents operations from accessing data concurrently in a possibly harmful way. So normally, you will start a transaction, read or write from/to one or more containers and commit. All other application parts that want to access that part of the space but don't use this transaction are blocked.

Chapter 5:

A completely new aspect: Aspects in XVSM

An Aspect in XVSM lets you implement an extension to the existing XVSM functionality. Features like automatic persistency or notification can be implemented using Aspects. You also can use Aspects to add logging, authentication and many more features to the existing XVSM implementation. The idea behind Aspects is to have a hook before and after each space operation, for example a preWrite and a postWrite method. You implement these pre- and post-methods to perform some special action that is done automatically before or after you call one of the following methods of the Capi class. There are two kinds of aspects: Container Aspects and Space Aspects. Container Aspects are aspects that are connected to actions on a specific container, while Space Aspects refer to the space itself or to all containers. Each possibility to listen on a certain performed action is called IPoint (standing for Interceptor Points). Adhering to the just mentioned Container and Space Aspects, the corresponding IPoints are called `ContainerIPoint` and `SpaceIPoint`. You not only have the possibility to create one Aspect to perform an action, but you can also add multiple Aspects to perform various actions in sequence. This possibility will be explained at the end of Chapter 5.1.

5.1 Container Aspects

Container Aspects have the possibility to offer methods for the following actions:

- pre-read, post-read
- pre-delete, post-delete
- pre-take, post-take
- pre-write, post-take
- pre-addAspect, post-addAspect
- pre-removeAspect, post-removeAspect
- post-createContainer
- post-lookupContainer
- pre-destroyContainer, post-destroyContainer
- pre-lockContainer, post-lockContainer

If you want to create a new Container Aspect, you can implement the ContainerAspect interface or extend the AbstractContainerAspect class. If you want to create a new Space Aspect, you can implement the SpaceAspect interface or extend the AbstractSpaceAspect class. The abstract classes already contain the needed pre- and post-methods that per-se don't do anything. If you implement a new Aspect, you override these methods to add new functionality to the method.

An example for a new Container Aspect is:

```
public class LoggingAspect extends AbstractContainerAspect
{
    static final long serialVersionUID = 0;

    public AspectResult postWrite(final WriteEntriesRequest request,
        final Transaction tx, final SubTransaction stx,
        final Capi3AspectPort capi3, final int executionCount)
    {
        Iterator it = request.getEntries().iterator();
        while (it.hasNext())
        {
            /* Print the value of the Entry */
            System.out.println(it.next());
        }
        return AspectResult.OK;
    }

    public AspectResult postRead(final ReadEntriesRequest<?> request,
        final Transaction tx, final SubTransaction stx,
        final Capi3AspectPort capi3, final int executionCount,
        final List<? extends Serializable> entries)
    {
        Iterator it = entries.iterator();
        while (it.hasNext())
        {
            /* Print the value of the entry */
            System.out.println(it.next());
        }
        return AspectResult.OK;
    }
}
```

Listing 5.1: Logging Aspect

In the listing above, you extend the AbstractContainerAspect class and override the various methods and therefore add new functionality. The methods look like this:

```
AspectResult postWrite(
    WriteEntriesRequest request,
    Transaction tx,
    SubTransaction stx,
    Capi3AspectPort capi3,
    int executionCount);
```

With the request object you can access a list of Entries that is used in this action, to perform some special task with these entries, as you can see in the listing. It's also possible to change these Entries by using list operations like add or remove. Also you can access the context, which is free to your disposal, if you want to pass additional information to the aspect. The context can be passed with every Capi method. In MozartSpaces, each operation is executed in a sub transaction. To access the container used by this operation you have to use the Capi3 interface. This is necessary to prevent loops. ExecutionCount is the number of processings of this request. Normally an aspect returns the `AspectResult.OK` state. If the operation and all following pre-aspects should be skipped, use the `AspectResult.SKIP` return result. In this case, the execution will continue with the first post aspect if available. An Aspect has the possibility to throw `RuntimeExceptions` or return a `NOTOK` state to indicate an error, which cancels the execution of the whole operation.

After you created the new aspect, you add it to the Capi by calling

```
ContainerAspect aspect = new LoggingAspect();
Set<ContainerIPoint> ipoints = new HashSet<ContainerIPoint>();
ipoints.add(ContainerIPoint.POST_WRITE);
ipoints.add(ContainerIPoint.POST_READ);
capi.addContainerAspect(aspect, cref, ipoints, null);
```

You pass the aspect itself and a set of IPoints, for which the aspect should be called. The reason that you need to pass a number of IPoints is that you can choose if you want to perform only a part of the actions that an Aspect would offer. Imagine for example a Logging Aspect which offers Logging possibility for each Pre-/Post method, but you only would like to use it for the write action. Instead of writing a new Aspect, you use the IPoints `PreWrite` and `PostWrite` and that's it.

As explained in the introduction of Chapter 5, you have the possibility to add multiple Aspects to let them work one after the other. You only need to call the `add` method with the Aspect in the same order as you'd like them to be executed.

To remove an Aspect you the Capi offer the `removeAspect` method. It is used with the `AspectReference` you get when you add an Aspect. It's also possible to remove only selected interception points.

```
void removeAspect(AspectReference aspect) throws MzsCoreException
```

```
void removeAspect(AspectReference aspect,  
    Set<? extends InterceptionPoint> iPoints,  
    TransactionReference transaction)  
    throws MzsCoreException
```

```
public static void main(String[] args)  
{  
    try  
    {  
        MzsCore core = DefaultMzsCore.newInstance();  
        Capi capi = new Capi(core);  
        /* Create new Container*/  
        ContainerRef cref = capi.createContainer(null, new FifoCoordinator(),  
            null);  
  
        ContainerAspect aspect = new LoggingAspect();  
        /* Create the IPoints for the aspect */  
        Set<ContainerIPoint> p = new HashSet<ContainerIPoint>();  
        p.add(ContainerIPoint.POST_WRITE);  
        p.add(ContainerIPoint.POST_READ);  
        capi.addContainerAspect(aspect, cref, p, null);  
  
        /* Write 3 entries to the container */  
        for (int i = 1; i <= 3; i++)  
        {  
            Entry entry = new Entry(i, FifoCoordinator.newCoordinationData());  
            capi.write(cref, 0, null, entry);  
        }  
        capi.read(cref, 0, null);  
  
        capi.shutdown(null);  
    }  
    catch (Exception e)  
    {  
        e.printStackTrace();  
    }  
}
```

Listing 5.2: Adding the LoggingAspect to Capi

5.2 Space Aspects

Using Space Aspects, you can listen to the following actions, additionally to those already mentioned in Container Aspects:

- pre-createContainer, post- createContainer
- pre-lookupContainer
- pre-createTransaction, post-createTransaction

- pre-prepareTransaction, post-prepareTransaction
- pre-commitTransaction, post-commitTransaction
- pre-rollbackTransaction, post-rollbackTransaction
- pre-shutdown

The usage of a Space Aspect is similar to the usage of a Container Aspect. The only differences are the methods that are offered and that a Space Aspect is associated with a space, not with a specific container. So, for example you overwrite the following method:

```
AspectResult preCreateContainer(
    CreateContainerRequest request,
    Transaction tx,
    SubTransaction stx);
```

and then, you add an Aspect as follows:

```
URI uri = new URI(...); //URI to a space (may be null for local)
SpaceAspect aspect = new SecurityAspect();
Set<SpaceIPoint> p = new HashSet<SpaceIPoint>();
p.add(SpaceIPoint.PRE_CREATE_CONTAINER);
capi.addSpaceAspect(aspect, uri, p, tx);
```

You also have the possibility to use the Container Aspects together with Space Aspects or use a Space aspect as Container aspect, because a Space aspect extends the Container aspect. If you use the Container specific actions in a Space aspect, the actions are performed on ALL containers and not only a specific one. Using this functionality, you can for example implement a method that writes a log entry whenever an entry is read from ANY container in your space. Consider that Space aspects are executed before Container aspects.

Summary:

Using Aspects, you can easily add new functionality to your XVSM system. You can define some actions that are executed either before or after an operation is performed with the container or space. So for example you can add the functionality that before a write, the permissions of the writer is checked. This way, security checks can be introduced. For a new Aspect, you overwrite the pre- and post-methods to add new functionality and then, you register the Aspect in the Capi.

Chapter 6: Don't miss a thing: Notifications

Using Aspects, a very important feature was added to the XVSM implementation: Notifications. With notifications, you can listen on various actions and be called back if such an action is performed. For example:

You have a listening class that would like to be notified if a write is called (and thus, a new entry is added). This class must implement the `NotificationListener` interface and so it must implement the method

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

Notifications are managed by the `NotificationManager` class. To create the manager just call

```
NotificationManager notifManager = new NotificationManager(core);
```

The signature of the `createNotification` method is:

```
Notification createNotification(  
    ContainerReference container,  
    NotificationListener listener,  
    Set<Operation> operations,  
    TransactionReference transaction,  
    RequestContext context)  
    throws MzsCoreException, InterruptedException
```

In our example you need to register this class as listener for the call of the `write` method:

```
Notification notification = notifManager.createNotification(  
    cref,  
    this,  
    Operation.WRITE,  
    tx,  
    null);
```

If you want to remove a notification, the `Notification` object offers you a `destroy` method. You can also shut down the `NotificationManager`, which destroys all notifications.

```
notification.destroy();  
notifManager.shutdown();
```

There exist the following notification targets, which should be self-explanatory:

```
Operation.ALL  
Operation.WRITE  
Operation.READ  
Operation.TAKE  
Operation.DELETE
```

Putting all together, you can find that with the usage of notifications, you can much more efficiently program some of the examples/exercises of the recent chapters. Just to take one example:

In the `TicketQueue` example, you've created one main peer which handles the Persons in the queue one after the other. Now imagine that you start one peer for each person and one peer for the shop assistant (which now is a new class). Both the Person and the Shop assistant implement the `NotificationListener` interface. When the Shop assistant is instantiated, it is registered as listener on a request to pay a ticket. The Person is registered as listener of the ticket when it has paid the ticket to receive the printed ticket.

And as you might expect, there is a final exercise:

6.1 Exercise (*TicketQueue with Notifications*)

Try to program the just-described `TicketQueue` Example. You will need to have

- A `Ticket` and a `Payment` object that are used to notify the other party to do something.
- A second container to store these `Ticket` and `Payment` entries independently from the FIFO queue of the persons. You may name this container "SalesDesk".
- A couple of Persons which have a `payTicket()` method – the `waitForTicket()` method is not needed anymore, as you now have a `entryOperationFinished()` method which tells the Person that the Ticket is printed. In the `payTicket()` method, write the `Payment` object to the `SalesDesk` container and register the Person as listener for a call of the write method. In the `entryOperationFinished` method, you need to take care if really a `Ticket` was returned (imagine multiple `ShopAssistants` and `Persons` at a time writing to the container – if you don't

check if the Ticket is "yours", the person could either "steal" Payment or Tickets from others). If it really is a ticket, take it from the container, give a `System.out.println` that you've got the ticket and leave the queue ("delete") the Person object from the container and shutdown this peer).

- A single ShopAssistant class which implements NotificationListener. In its `entryOperationFinished` method, take the Payment from the SalesDesk container, give a `System.out.println()` that you're printing the ticket and wait for 1 second. Then store the Ticket object in the container. Again, in the `entryOperationFinished` method, take care that you only take Payment objects, not Ticket objects.

Summary:

A Notification is used if your class is interested in a special event in the Container, for example if an Entry was written or read. This Entry can be passed to the `notify` method in order to enable the listening class to perform some action using this Entry. Using Notifications and looking at the last example, the character of Space-based computing is shown best: Instead of imperatively telling the ShopAssistant or the Person to do something, you write a request to the space instead. The other party takes the request, performs the requested action and writes the result back. If the requested party is overloaded or too slow – no problem: you add one more of such participants; they collaboratively work on the requests in the container. With the possibility to spread the container over multiple peers and with help of transactions, you can implement load balancing scenarios in the easiest way. Moreover, you can add or remove working peers on-the-fly.

Chapter 7: Bibliography

[1] Mozartspaces. Website. <http://www.mozartspaces.org>; last visited on October 23th 2010.

[2] Space based computing group. Website. <http://www.complang.tuwien.ac.at/eva/SBC-Group/sbcGroupIndex.html>; last visited on October 23th 2010.

[3] David Gelernter. Generative Communication in Linda. ACM Trans. Program. Lang. Syst., 7(1):80-112, 1985.

[4] Martin Barisits. Master-Thesis, Design and Implementation of the next Generation XVSM Framework