

YAMI Java Reference

Copyright © 2001-2008 Maciej Sobczak

Contents

1	Introduction	2
2	Parameters	3
3	Easy Client	4
4	Client	6
5	Callback	7
6	Server	8
7	Exceptions	9
8	Limitations and implementation notes	9

1 Introduction

The *YAMI Java* is a small set of classes provided for the Java language and supporting the basic subset of client-side and server-side functionality of *YAMI*. The module is implemented entirely in Java and itself has no dependencies.

The *YAMI Java* consists of the following files, all implementing the `yami` package:

- `ParamSet.java` – encapsulates the sequence of parameters
- `Client.java` – encapsulates the connection with *YAMI* server and provides methods for sending messages
- `EasyClient.java` – provides basic methods for one-time client messages (for non-persistent connections with the server)
- `Callback.java` – defines a callback interface for processing incoming messages
- `Server.java` – encapsulates the server-side functionality
- `YAMIException.java` – defines common exception type for all *YAMI*-related problems; all other exception classes are direct subclasses of this one
- `NoSuchObjectException.java` – defines an exception type for the situation when the message is sent to object that is not recognized by the server
- `OutOfBoundsException.java` – defines an exception type that is reported when the user tries to read from *Parameter Set* an entry that does not exist
- `OverflowException.java` – defines an exception type that is reported when the message is rejected by the server due to overflow of its internal queues
- `RejectedByAgentException.java` – defines an exception type reported when the message is rejected by the server due to incorrect typing level (this can happen only when the server is of *Level1*)
- `RejectedException.java` – an exception type that is reported when the message was received, but the server did not want to (or did not manage to) process it
- `WrongDataTypeException.java` – an exception type that is thrown when the user tries to read entry from the *Parameter Set* and the entry has other type than requested.

Other files are private within this package and provide internally-used utilities.

2 Parameters

The concept of *Parameter Set* is implemented with the help of the `ParamSet` class. It is used for encapsulating parameters for both sending the messages as well as receiving responses.

Public methods of this class are listed below:

```
package yami;
public class ParamSet {

    public ParamSet() {...}

    public ParamSet append(String str) {...}
    public ParamSet appendAsWString(String str) {...}
    public ParamSet append(int i) {...}
    public ParamSet append(double d) {...}
    public ParamSet append(byte b) {...}
    public ParamSet append(byte [] buf) {...}

    public void rewind() {...}

    public int getNumberOfEntries() {...}

    public enum Type { String, WString, Integer,
                      Double, Byte, Binary };

    public Type getType() throws YAMIEException {...}

    public String  extractString()  throws YAMIEException {...}
    public String  extractWString() throws YAMIEException {...}
    public int     extractInt()     throws YAMIEException {...}
    public double  extractDouble()  throws YAMIEException {...}
    public byte    extractByte()    throws YAMIEException {...}
    public byte [] extractBinary()  throws YAMIEException {...}
}

```

The methods of this class are described below.

`ParamSet()`

Constructor creating empty *Parameter Set* object (with no entries).

```
ParamSet append(String str)
ParamSet appendAsWString(String str)
ParamSet append(int i)
ParamSet append(double d)
ParamSet append(byte b)
ParamSet append(byte [] buf)

```

Append a given value to the *Parameter Set* and returns **this** (to allow call-chaining).

The difference in value processing between methods `append(String str)` and `appendAsString(String str)` is that the former expects an ASCII string and creates an entry that can be received as single-byte string, whereas the latter treats its argument as a Unicode string and creates an entry according to the *YAMI* type `WString`.

```
void rewind()
```

Resets the *current index* for reading.

The concept of *current index* or *current entry* has no significance from the *YAMI* point of view, but is used at the level of client code for convenient inspection of the *Parameter Set* object and is relevant to all methods described later on.

```
int getNumberOfEntries()
```

Returns the length of *Parameter Set* object.

```
enum Type { String, WString, Integer, Double, Byte, Binary };
Type getType() throws YAMException
```

Defines available types of entries and returns the type of *current* entry.

```
String extractString() throws YAMException
String extractWString() throws YAMException
int extractInt() throws YAMException
double extractDouble() throws YAMException
byte extractByte() throws YAMException
byte [] extractBinary() throws YAMException
```

Return the value in the *current entry*.

An exception is thrown when the type of *current entry* does not match the method.

Every function that accepts the *Parameter Set* can also accept `null` as an indication that the *Parameter Set* is empty. Convenience overloads are provided where the *Parameter Set* argument can be omitted entirely.

3 Easy Client

The basic functionality for sending synchronous messages with non-persistent connections is provided by the `EasyClient` class.

This class is provided for convenience where messages are sent rarely or where the performance of communication infrastructure is of no concern. For purposes where many messages are sent to remote objects, it is preferred to create full `Client` (see below) object once and use it for sending many messages.

```

package yami;
public class EasyClient {
    public static void setTimeout(int tm) {...}

    public static void sendOneWay(String address, int port,
        String objectName, String messageName)
        throws YAMIEException {...}

    public static ParamSet send(String address, int port,
        String objectName, String messageName)
        throws YAMIEException {...}

    public static void sendOneWay(String address, int port,
        String objectName, String messageName, ParamSet ps)
        throws YAMIEException {...}

    public static ParamSet send(String address, int port,
        String objectName, String messageName, ParamSet ps)
        throws YAMIEException {...}
}

```

All methods in this class are *static*.

```
static void setTimeout(int tm)
```

Sets the timeout, in milliseconds, for all subsequent connections. This timeout is the maximum time the client will wait for response, which means that it does not relate to *one-way* methods.

By default this timeout is set to 5000ms.

```

static void sendOneWay(String address, int port,
    String objectName, String messageName)
    throws YAMIEException {...}
static void sendOneWay(String address, int port,
    String objectName, String messageName, ParamSet ps)
    throws YAMIEException {...}

```

Sends a single *one-way* message to the given network address (for this parameter both "xxx.xxx.xxx.xxx" and "host.company.com" forms can be used) and port. `objectName` and `messageName` are used for message routing at the server side.

The `ParamSet` argument can be null or omitted.

The message is *one-way*, which means that its further status is not checked and no response is received.

```
static ParamSet send(String address, int port,
```

```

        String objectName, String messageName)
        throws YAMIEException {...}
static ParamSet send(String address, int port,
        String objectName, String messageName, ParamSet ps)
        throws YAMIEException {...}

```

Sends a single message to the given network address and port.

The `ParamSet` argument can be `null` or omitted.

The response from the server is received and returned as *Parameter Set* object.

4 Client

The basic functionality for sending synchronous messages with persistent connections is provided by the `Client` class.

```

package yami;
public class Client {
    public void setTimeout(int tm) {...}

    public Client(String address, int port)
        throws YAMIEException {...}

    public void close() {...}

    public void sendOneWay(String objectName, String messageName)
        throws YAMIEException {...}

    public ParamSet send(String objectName, String messageName)
        throws YAMIEException {...}

    public void sendOneWay(String objectName, String messageName,
        ParamSet ps)
        throws YAMIEException {...}

    public ParamSet send(String objectName, String messageName,
        ParamSet ps)
        throws YAMIEException {...}
}

```

The methods of this class are described below.

```
void setTimeout(int tm)
```

Sets the timeout, in milliseconds, for all subsequent connections. This timeout is the maximum time the client will wait for response, which means that it does not relate to *one-way* methods.

By default this timeout is set to 5000ms.

`Client(String address, int port)` throws `YAMException`

Constructs the object encapsulating the persistent connection with the server. For the `address` parameter both “numeric” “xxx.xxx.xxx.xxx” and “human-readable” “host.company.com” forms can be used.

`void close()`

Closes the connection.

`void sendOneWay(String objectName, String messageName)`
throws `YAMException {...}`

`void sendOneWay(String objectName, String messageName,
ParamSet ps)`
throws `YAMException {...}`

Sends a single *one-way* message. `objectName` and `messageName` are used for message routing at the server side.

The `ParamSet` argument can be null or omitted.

The message is *one-way*, which means that its further status is not checked and no response is received.

`ParamSet send(String objectName, String messageName)`
throws `YAMException {...}`

`ParamSet send(String objectName, String messageName, ParamSet ps)`
throws `YAMException {...}`

Sends a single message to the server.

The `ParamSet` argument can be null or omitted.

The response from the server is received and returned as *Parameter Set* object.

5 Callback

The `Callback` interface defines the protocol for passive processing of incoming messages.

```
package yami;
public interface Callback {
    ParamSet process(String objectName,
        String messageName, ParamSet parameters)
        throws java.lang.Exception;
}
```

There is only one method defined by this interface.

```
ParamSet process(String objectName,
    String messageName, ParamSet parameters)
    throws java.lang.Exception;
```

This method is called when the server receives a new incoming message. The `objectName` and `messageName` denote the *target* of the message. The `parameters` can be `null`, meaning that no parameters were attached to the message.

This method, when implemented by the user, should return one of:

- Fully constructed `ParamSet` object, which will be sent back to client as a response to the given message.
- `null` to indicate that reply should be sent without parameters.
- `Server.noReply` special object (see below) to indicate that no reply should be sent whatsoever (for one-way messages).

If this method throws the `NoSuchObjectException`, then the client receives the *unknown object* status. Any other exception thrown from this method is interpreted as the failure at the server side and the client will receive *rejection* notification.

6 Server

The basic functionality for receiving and replying to incoming messages is provided by the `Server` class.

```
package yami;
public class Server {

    public static ParamSet noReply = ...

    public Server(int port, Callback messageHandler)
        throws YAMException { ... }

    public void close() {...}
}
```

The members of this class are described below.

`ParamSet noReply`

A special object that server-side code can use to indicate that no reply should be sent to client for the message that is currently being processed.

`Server(int port, Callback messageHandler)`
 throws `YAMException`

Constructs the object encapsulating the server-side functionality with the listening socket bound on the given port number.

The `messageHandler` parameter should point to the user-provided implementation of the `Callback` interface. The given instance will be used for passive callbacks with incoming messages.

```
void close()
```

Closes the listening socket. The currently active connections established by clients are not forcibly shut down and are kept in memory until clients disconnect.

7 Exceptions

The *YAMI Java Module* reports run-time errors by throwing exceptions of `YAMIException` class or some of its derived classes:

```
package yami;
public class YAMIException
    extends Exception {...}

public class NoSuchObjectException
    extends YAMIException {...}

public class OutOfBoundsException
    extends YAMIException {...}

public class OverflowException
    extends YAMIException {...}

public class RejectedByagentException
    extends YAMIException {...}

public class RejectedException
    extends YAMIException {...}

public class WrongDataTypeException
    extends YAMIException {...}
```

8 Limitations and implementation notes

The following issues should be taken into account:

- Both client- and server-side of *YAMI Java* operate only in the full duplex mode with typing level 2.
- The server uses the thread-per-connection strategy to handle clients. This means that this implementation should be used with care in those environments where very high number of clients can simultaneously connect to the single server. This strategy influences also the concurrency model of the notifications in the sense that callbacks (via the `Callback` interface) can be concurrent if the messages come from separate clients at the same time.

The user should therefore ensure that the implementation of the `process` method can properly handle this concurrency. The final consequence of this model is that threads created for handling incoming connections are not terminated until clients disconnect, which means that the `shutdown` method of the main server class does not guarantee that after shutting the server down there will be no more invocations via the `Callback` reference, only that the listening socket is closed and no new incoming connections will be established.