

YAMI Java Tutorial

Copyright © 2001-2008 Maciej Sobczak

Contents

1	Introduction	2
2	Echo	2
2.1	Easy client	2
2.2	Easy client with data	3
2.3	Full blown client	4
2.4	Server	6
2.5	Error handling	7
3	Calculator	7
3.1	Client	8
3.2	Server	9

1 Introduction

The tutorial for *YAMI Java Module* differs from tutorials for other supported languages. This is the consequence of the fact that from the design point of view the *YAMI Java Module* provides a bit different way to work with outgoing and incoming messages. Still, the examples presented here are supposed to be exact equivalents of similar examples in other tutorials, with exceptions which are explained below. Do not hesitate to run the examples from different tutorials in pairs to see that *YAMI* provides abstractions that are language-independent and that the connectivity between programs written in different languages is really achieved.

The technical difference between Java client-side examples and examples written in other languages is that *YAMI Java Module* can work only with full duplex messages, which means that Java server will *not* be able to correctly reply to the message sent by client working in the simplex mode. Please modify the client examples in other languages accordingly (by setting full duplex option when sending messages or when registering remote domains) so that Java servers can properly send their replies.

In this tutorial, there are two assumptions:

- Every server runs on the port 12340. This is arbitrary choice.
- Every client runs on the same machine as the server (this is for simplicity only – if you have access to the network, try the examples running the processes on different machines).

2 Echo

As a first example, you will write your own client for the *echo* server. The echo server will wait for messages from clients and will print them on the screen. Apart from that, the server will also accept a special *shutdown* message. There will be no data sent back to clients.

2.1 Easy client

First, you will learn how to write *easy* client, which means the client that does not keep any persistent connection with the server – it connects, sends the message and disconnects immediately after that.

The first client in this tutorial sends a message to the server that tells it to shut down. Here it is (file `ShutDown.java`):

```
1 import yami.*;
2
3 public class ShutDown {
4     public static void main(String [] args) {
5         try {
```

```
6
7         EasyClient.sendOneWay("127.0.0.1", 12340,
8             "echo", "shutdown");
9
10        } catch (Exception e) {
11            System.out.println("Error: " + e);
12        }
13    }
14 }
```

Yes, it is *that* easy.

The first line imports the names from the `yami` package. It assumes that the library is available on the `classpath` pointing to `yami.jar` file.

The line with call to static method `sendOneWay` alone is enough to send a message to the server. There are four parameters used in this command:

- `127.0.0.1` – the server's address. It is a string value and can have the form `comp.company.com` as well.
- `12340` – the server's port
- `echo` – the name of the destination object. In *YAMI*, messages are sent to objects which means that there can be many logical destinations in the same server.
- `shutdown` – the message name. Later you will see that this message causes the server to shut down.

The `sendOneWay` method sends the message in a synchronous manner – it will block until the server reads the whole message, but otherwise does not wait for anything. In particular, the program will continue probably leaving the server still busy processing the message. In this scenario no response is sent back from server to client.

Note that the `sendOneWay` command is overloaded and that there is another version of this command that accepts additional parameter. We will use it later.

Interesting? So let's pass some data with the message.

2.2 Easy client with data

In the previous section, the message had only a name. In many cases it is enough, because it allows the server to make decisions about what action to perform, but there are situations where it would be nice to pass some data together with the message. In *YAMI*, there are two levels of messages with regard to the data that can be sent:

- *Level1* – this level allows only strings and wide strings. It was designed for the components written in languages that do not understand other data types.

- *Level2* – this level allows strings, wide strings, integers, doubles, bytes and arbitrary binary blocks.

The *YAMI Java Module* always sends *Level 2* messages. This makes the interface simpler, but the consequence of this is the fact that there is no way to communicate with *Level 1* servers. Hopefully this will never be a problem.

There can be many values of different types that can go with the same message at once. All the values are stored in the so-called *Parameter Set*, which is basically a list of values of chosen type.

The client that causes the server to print "Hello, YAMI!" message on the screen can look like here (file `Hello.java`):

```

1 import yami.*;
2
3 public class Hello {
4     public static void main(String [] args) {
5         try {
6
7             EasyClient.sendOneWay("127.0.0.1", 12340,
8                 "echo", "print",
9                 new ParamSet().append("Hello, YAMI!"));
10
11         } catch (Exception e) {
12             System.out.println("Error: " + e);
13         }
14     }
15 }

```

Yes, it is *that* easy.

In this example, we append the list of parameters to the message. The list has only one parameter of the string value "Hello, YAMI!". Note also that the message name has changed.

Note that the *Parameter Set* object is prepared “on the fly”, without creating any named reference. This is possible thanks to call-chaining. The explicit and longer version will be used in later example.

2.3 Full blown client

The examples written so far have one thing in common: they send only one message to the server and quit. There is nothing that prevents you from sending millions of messages using the methods presented so far, but in terms of performance it will not be a good idea. The static methods from `EasyClient` class create separate network connections for each message and close them after that, leading to unnecessary overhead if many messages are sent to the same destination. Another scheme is available for this purpose:

1. Create the *YAMI* connection object.

2. Send the message with the help of this connection object. Repeat this step as many times as you wish.
3. Close the connection object.

In this section you will write the client according to this scheme. The client itself will be smarter than before, too – it will read the lines of text from its standard input and send each line as a separate message to the *echo* server. The code is below (file `PrintAll.java`):

```
1 import yami.*;
2 import java.io.*;
3
4 public class PrintAll {
5     public static void main(String [] args) {
6         try {
7
8             Client client =
9                 new Client("127.0.0.1", 12340);
10
11             BufferedReader in = new BufferedReader(
12                 new InputStreamReader(System.in));
13
14             String line;
15             while ((line = in.readLine()) != null) {
16                 ParamSet ps = new ParamSet();
17                 ps.append(line);
18
19                 client.sendOneWay("echo", "print", ps);
20             }
21
22             client.close();
23
24         } catch (Exception e) {
25             System.out.println("Error: " + e);
26         }
27     }
28 }
```

The program performs a loop where it reads lines of text from standard input. Each line is then sent as a parameter with the message to the server. After reaching *end-of-file* condition on the standard input, the connection to the server is closed and the client quits.

Note that the `sendOneWay` method allows to send the message without any response from the server. The responses will be used in later examples.

2.4 Server

Finally, we will write the server (file `Server.java`):

```

1 import yami.*;
2
3 public class Server {
4     public static void main(String [] args) {
5         try {
6             Callback myHandler = new MyHandler();
7             yami.Server server = new yami.Server(12340, myHandler);
8
9             // everything happens in background, sleep for 60s
10            Thread.sleep(60000);
11        } catch (Exception e) {
12            System.out.println("Error: " + e);
13        }
14    }
15 }
16
17 class MyHandler implements Callback {
18     public ParamSet process(String objectName,
19         String messageName, ParamSet parameters)
20         throws YAMIEException {
21
22         if (objectName.equals("echo")) {
23             if (messageName.equals("shutdown")) {
24                 System.exit(0);
25             } else {
26                 String s = parameters.extractString();
27                 System.out.println(s);
28                 return yami.Server.noReply;
29             }
30         }
31
32         throw new NoSuchObjectException();
33     }
34 }

```

With *YAMI Java Module*, the server accepts messages directed to any object. This is different from other languages, where objects had to be explicitly registered before any communication could take place. Here the user is responsible of logical object management and route messages that target different logical destinations.

The server functionality is handled in the background by additional threads that are started for each incoming client connection and the user is passively called via the given `Callback` reference.

When the message is finally retrieved, the server checks its name. If it is **shutdown**, it just terminates the whole process. There is also a separate **close** method in the **Server** class, which closes down the listening socket and terminates the thread that is responsible for accepting new messages, but with this method threads that handle already established connections are not terminated – they will naturally complete when their respective clients disconnect.

Otherwise (which means that the message name is **print**; in real system you would also take some steps if the message has unknown or unacceptable name), the server gets the parameter list from the message and prints its first element on the standard output.

Note that the server code returns the special value **Server.noReply** to indicate that no reply should be sent to the client – this is a one-way message.

Note also that in this example it was necessary to use a full **yami.Server** name to resolve conflict with application's own **Server** class.

2.5 Error handling

Many different things can go wrong in the distributed system. Some of them will be your fault, some not. Almost every *YAMI* method can throw an exception. In the example programs presented so far there was no real exception management, but in real programs you will for sure want to check for and manage the exceptions.

YAMI Java Module reports all problems by throwing exceptions of the **YAMIException** class or one of its derived classes.

The exceptions are also used to inform *YAMI Java Module* about various problems when processing the incoming message:

- When the object name is not recognized (in other words, when the clients sends a message to the destination that is not know at the server side), the user can express this by throwing an instance of **NoSuchObjectException**. The client will get appropriate notification.
- When the server cannot successfully process the given message, then *any* exception thrown from the **process** method of the **Callback** interface, other than **NoSuchObjectException** will indicate *message rejection* and the client will also get an appropriate notification.

3 Calculator

In this second example, you will write the client-server pair where the client asks for something and the server *responds* to the messages, sending some data back to client. This is probably the most common way of developing distributed systems.

The calculator example consists of:

- The server that accepts messages: `add`, `sub`, `mul` and `div`, each with two integer parameters. Each message will be replied to, with the appropriate integer result.
- The client that sends messages asking the server to perform some computation.

I'm sure that after this example you will be able to write distributed systems of arbitrary complexity.

3.1 Client

Let's start with easy things. The client (file `CalcClient.java`) looks like here:

```

1 import yami.*;
2
3 public class CalcClient {
4     public static void main(String [] args) {
5         try {
6
7             Client client =
8                 new Client("127.0.0.1", 12340);
9
10            ParamSet arguments = new ParamSet();
11            arguments.append(100);
12            arguments.append(20);
13
14            ParamSet response = client.send(
15                "calculator", "add", arguments);
16
17            int result = response.extractInt();
18
19            System.out.println("Result is " + result);
20
21            client.close();
22
23        } catch (Exception e) {
24            System.out.println("Error: " + e);
25        }
26    }
27 }
```

The main difference to the previous example is that the `send` method is used instead of `sendOneWay`. This method returns the *Parameter Set* object that contains response from the server, so that the client can inspect it and use the results.

As you can see, the message is sent with two integer parameters in the list of parameters (100 and 20). The message is sent synchronously, which means that the program will wait for response. How long it will take depends only on the network latencies and server processing time. The `send` method returns the *Parameter Set* object with all the data that server has sent back. Here we just print its only (first) parameter.

This Calculator example shows also that there are two contexts where the list of parameters can be used:

- When the client sends a message to the server. The list of parameters can be supplied at the client-side and retrieved at the server-side.
- When the server replies to the message. The server can supply a list of parameters and the client can extract it after receiving the response.

These two contexts allow to send data in both directions.

3.2 Server

The server is presented below (file `CalcServer.java`):

```
1 import yami.*;
2
3 public class CalcServer {
4     public static void main(String [] args) {
5         try {
6             Callback myHandler = new MyHandler();
7             Server server = new Server(12340, myHandler);
8
9             // everything happens in background, sleep for 60s
10            Thread.sleep(60000);
11        } catch (Exception e) {
12            System.out.println("Error: " + e);
13        }
14    }
15 }
16
17 class MyHandler implements Callback {
18     public ParamSet process(String objectName,
19         String messageName, ParamSet parameters)
20         throws Exception {
21
22         int a = parameters.extractInt();
23         int b = parameters.extractInt();
24         int result;
25
26         if (messageName.equals("add")) {
```

```
27         result = a + b;
28     } else if (messageName.equals("sub")) {
29         result = a - b;
30     } else if (messageName.equals("mul")) {
31         result = a * b;
32     } else if (messageName.equals("div")) {
33         if (b == 0) {
34             // cannot divide by zero!
35             throw new Exception();
36         }
37         result = a + b;
38     } else {
39         // unknown message, reject it
40         throw new Exception();
41     }
42
43     return new ParamSet().append(result);
44 }
45 }
```

This server has one design quirk: the server does not perform any interaction with the user and there is no way to remotely shut the server down. Locally, the administrator can just kill the server process.

I do not advocate this style of writing servers. The two clean solutions that I recommend are:

- Provide additional shut-down message, like in the Echo example. Exit cleanly from the server process when you receive the *shutdown* message.
- Provide additional logical shut-down *object* and accept messages also for this second object. The first message that will be received for this special shut-down object will tell you that it is time to terminate the server in a clean way.

Enjoy! And, of course, tell your friends about *YAMI*.