



J A D E <sup>TM</sup>

## Server Methods

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2009 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

---

# Contents

Contents ii

<b>Server Methods</b>	<b>3</b>
Introduction .....	3
What is a server method? .....	3
An example of a server method .....	3
Passing parameters to a server method.....	4
“Mixing-and-Matching” client and server methods.....	4
Using client methods .....	5
Updating persistent objects .....	6
Transaction considerations .....	6
Design implications .....	7
Transient objects .....	7
Performance implications .....	8
Server method execution threads.....	8
Simultaneous connections limit .....	8
Abnormal server method termination.....	9
The ‘write’ instruction .....	9
Application and Global system variables .....	9

---

# Server Methods

---

## Introduction

One significant capability that sets JADE apart is that of server methods. With its unique ability to run the identical code on either a server node or a client node, JADE gives applications a remarkable flexibility. By simply changing the signature on a specific method, that method will run at the 'new' location the next time it is requested.

## What is a server method?

A *server method* is a method that has been designated to run on the JADE database server node (the device on which the JADE server program **jadrap.exe** is running). The method may be called/initiated just as any other method, and neither the developer nor the end-user needs to know that it is a server method.

## An example of a server method

A simple example of a server method would be when you want to calculate the total balance for all accounts.

The code in this example 'reads' every instance of **Account** within the **Company**, and accumulates the balances into **tBal**. On completion, it returns the total of the balances to the calling method.

```
computeTotalBalance(): Decimal serverExecution;
vars
  company : Company; // Company is a persistent class
  account : Account; // Account is a persistent class
  tBal : Decimal [10:2]; // Variable for accumulating balances
begin
  company := Company.firstInstance; // Get the Company instance
  // Retrieve each instance of Account within the specific Company
  foreach account in company.allAccounts do
    tBal := tBal + account.balance; // Add balance
  endforeach;
  return tBal; // Return total
end;
```

In this example, we saved the transfer of all instances of the **Account** class from being passed from the server node to a client node, as only the actual result of the computation was returned to the client node.

## Passing parameters to a server method

A server method is no different from a 'standard' method. It is written in the same style and language, and can accept parameters. The parameters may be io and of any type. A server method can obviously also return an object of any type.

For example, you may want to add all of the Account balances by month, having passed the method a specific year. Your code may therefore look similar to the following example.

```
computeTotalBalanceByMonth(year: Integer): DecimalArray
    serverExecution;

vars
    company : Company; // Persistent class of Company
    account : Account; // Persistent class of Accounts
    mBal : DecimalArray; // For accumulating monthly totals
    trans : Transaction; // Persistent class of Transactions
begin
    // Get the first instance of Company
    company := Company.firstInstance;
    // Create our transient object for values
    create mBal transient;
    // Retrieve each account
    foreach account in company.allAccounts do
        // Get each transaction with same year
        foreach trans in account.myTransactions
            where trans.year = year do
                mBal[trans.month] := mBal[trans.month] +
                    trans.value[trans.month];
            endforeach;
        endforeach;
    // Return transient object containing values
    return mBal;
end;
```

In this example we have looked at each instance of **Account**, and for each account we have looked at all transactions that took place in the specified year, totalling them into a decimal array called **mBal**. Upon completion, the **mBal** transient object is returned to the calling method.

## “Mixing-and-Matching” client and server methods

A client method can call other client methods or other server methods. Any called server method can also call any other server method or client method.

Methods will actually execute on the node of the caller unless the method has a signature indicating otherwise.

Consider the following example.

```
initialMethod();
vars
begin
    doMethodA();
    doMethodB();
end;

doMethodA() serverExecution;
vars
begin
    doMethodB();
end;

doMethodB();
vars
begin
    . . .
end;
```

What is the sequence here? The first method, called *initialMethod*, is executing on the client node. It calls *doMethodA*, and as *doMethodA* has the **serverExecution** method option specified in the signature, it will execute on the server node.

The *doMethodA* method calls the *doMethodB* method. Now, as *doMethodB* has no method option, it will execute in the same node as the calling method; that is, the server node. On completion, execution is returned to *doMethodA*, which in turn will return to *initialMethod*.

Next, the *initialMethod* method calls the *doMethodB* method. This time the *doMethodB* method will execute in the client node, as the method has no method option.

As you can see, the *doMethodB* method was able to run in both the server and client nodes, depending on who called it. This example demonstrates JADE's 'nested execution location' capability.

## Using client methods

In the previous example, we could specify the **clientExecution** method option for the *doMethodB* method. If we did so, then *doMethodB* would *always* run on the client node (and in the same thread of execution as the calling method), even when called from a server-executing method.

The new signature for the method would be:

```
doMethodB() clientExecution;
```

In other words, it is a way of enforcing execution location in a similar manner to the **serverExecution** method option. You would use this when the method in question was likely to be called from server-executing methods and contained code that accessed GUI objects, for example.

## Updating persistent objects

Server methods can update persistent objects as long as a client node transaction is not in effect at the time.

A good example would be for calculating totals. In the following example, a server method computes the interest on each account. It looks up balances, updates the interest for each account, and returns the total interest.

```
updateInterest(interestRate: Decimal): Decimal serverExecution;
vars
    company : Company;
    account : Account;
    tInt : Decimal [10:2];
begin
    company := Company.firstInstance;
    beginTransaction;
    foreach account in company.allAccounts where
        account.balance > 0 do
        account.interest := account.interest +
            (account.balance * interestRate);
        tInt := tInt + account.interest;
    endforeach;
    commitTransaction;
    return tInt;
end;
```

Server methods should not try to access objects that the process is currently updating on the client node. If the updates are not committed, they will not be visible to the server method. In fact, if the server method attempts to lock such objects, a 1276 exception is raised (“Potential cache inconsistency”). This is particularly relevant for collections, which are implicitly locked whenever accessed.

## Transaction considerations

If a **beginTransaction** instruction is executed on a client node, the matching **commitTransaction** instruction must also be done on the client node, although not necessarily in the same method. The transaction is active on the client node.

Conversely, a **beginTransaction** instruction initiated on a server node must have its matching **commitTransaction** instruction also executed on the server node. This makes the transaction active on the server node.

Server methods can be called while in transaction state, but cannot update persistent objects. The objects can only be updated on the node which has the transaction active.

The restriction does not apply to **beginTransaction** and **commitTransientTransaction**. Transient transactions can be “mixed-and-matched” across execution locations.

## Design implications

There are very few design implications. One implication is detailed in the previous item about **beginTransaction** and **commitTransaction** instructions.

The most obvious design restriction is that of not being able to reference any GUI-type objects. Remember that the code is running on the server, and has no GUI interface or capability. For example, you would not try to update data on a window (in a ListBox or Label control, for example) from within a server method. Should you attempt to do so, JADE will raise an appropriate error message.

The third restriction is that, as stated previously, a server executing method cannot “see” any persistent object updates carried out by a client executing method that have not yet been committed to the database.

Take a look at this example.

```
clientMethod() clientExecution;
vars
    account : Account;
begin
    beginTransaction;
    create account persistent;
    . . .
    serverMethod();
    . . .
    commitTransaction;
end;
```

The **serverMethod** method, which is a server executing method, cannot “see” or access the newly created **Account** instance, as the instance is not actually in the database until the **commitTransaction** instruction is executed. If the **commitTransaction** instruction preceded the **serverMethod** call, then it would have visibility to the new Account instance.

## Transient objects

Care should be taken when server methods use transient objects, to avoid excessive data transfer between nodes.

When a server method accesses an existing transient object, it has to be fetched from the client node. Updated transient objects have to be sent back to the client node whenever execution returns to the client node. Similarly, transient objects created in server methods have to be sent back to the client node when the method returns.

This is a particularly important consideration when a server method calls a client method (that is, a method marked as **clientExecution**). Any transient objects that have been updated get sent to the client node before the method is called. When accessed again by the server method, they have to be sent back from the client node. If you are not careful, this can create unexpected extra traffic between nodes.

Note that transient objects created by server methods do not get sent back to the client node if they are deleted before the method returns. Also, when a server method makes a nested call to a client method, the created transient objects are not sent beforehand. They are sent on request if the client method needs to access them.

## Performance implications

Apart from the transient object implication mentioned in the previous item, the basic performance trade off is that of centralised processing (on the server node) versus network traffic. The more work that is done on the server, the more likelihood there is of contention or queuing taking place on the server. The more work done on the client, the more chance there is of network delays, and so on.

As it is so simple to designate the execution location, the decision as to where a method should run can be left until very late in the development cycle, and can easily be changed later.

JADE provides both a Monitor and a Profiler that allow you to quickly determine what is running where, and what activity the methods are generating.

## Server method execution threads

Each client node does not require a dedicated thread for server method executions. The JADE server is dynamic, and has multiple threads. When a request is made for a server execution, the JADE server uses any available thread. The thread becomes available for any other use the moment the server method is complete.

You can determine the number of server threads that are allowed to run concurrently by setting the maximum number in the [JadeServer] section of the JADE initialization file (Jade.ini), as shown in the following example.

```
MaxServerThreads=<n>
```

## Simultaneous connections limit

Using server methods does not affect the simultaneous connections limit. This limit is based on the number of simultaneous client nodes, and is not related to the number of server threads.

## Abnormal server method termination

JADE handles abnormal server method termination very cleanly. Any uncommitted updates to persistent objects will be discarded, and an exception is raised.

Exception handling is accomplished by first looking for handlers armed in the local node where the current method is executing, and then on the node where the calling method resides.

The default exception handler for server methods will log the exception then raise a 1242 exception on the client node ("A method executing in another node was aborted"), to be handled by client node exception handlers.

## The 'write' instruction

The 'write' instruction can be used in server methods, but it should be noted that the output generated by the write instruction will be displayed on the device where the JADE database server is executing.

If you do not have access to the server, the easiest way to debug the method would be to remove the **serverExecution** method option, thus causing the output from the write instruction to be displayed on the client node.

## Application and Global system variables

The Application and Global system variables (**app** and **global**) can be used in server methods just as they are in client methods.

Remember that there are very few differences between the two execution locations.