JADE 6

JADE Development Centre

Conditions and Constraints

In this paper . . .

CONDITIONS AND CONSTRAINTS 1

INTRODUCTION 1
WHAT IS A CONDITION? 2
WHAT IS A CONSTRAINT? 4
CONSTRAINTS VERSUS MEMBERSHIP? 8
“INVERSE NOT REQUIRED” EXAMPLE 10
SUMMARY 12

Introduction

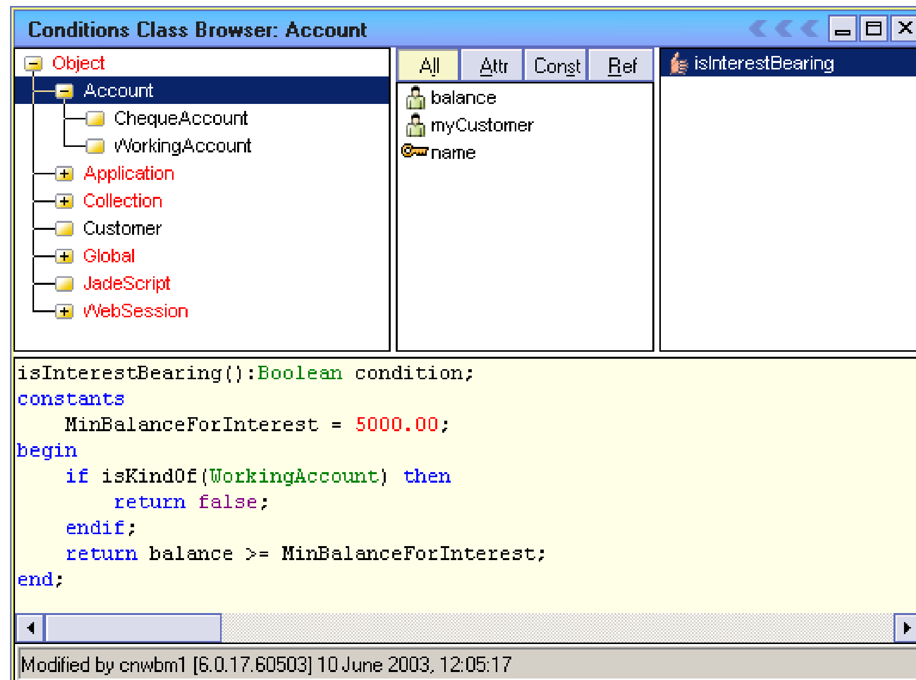
This document discusses the use of conditions in a JADE system. It begins by discussing conditions and their possible applications. It focuses on those cases where one or more inverses have constraints to limit the membership of the inverse. In such cases JADE will automatically maintain the inverse when any of the components of the condition are updated. This document covers some of the issues you should consider when using conditions and constraints in your JADE applications.

Jade Software Corporation Limited believes that the information furnished herein is accurate and reliable and has been prepared with care. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special or consequential losses or damages.

Copyright Jade Software Corporation Limited, 2007.

What is a condition?

A condition is a JADE method which returns a *Boolean* and has the reserved word `condition` in the signature. An example of a condition defined on the abstract class *Account* is the *isInterestBearing* condition shown below, which checks if the account has a balance of at least $5,000.



A condition is marked in the methods' browser with a question mark icon. One of the main uses of conditions is to put constraints on inverse references, so they have a number of restrictions that ensure they always produce the same result without side effects. In order to guarantee this, they must conform to a number of strict limitations. The only statements they may contain are `if` and `return` statements; they can contain constant definitions but no variable definitions, and can contain constant parameters (but not `input`, `output` or `io` parameters). They can only reference parameters, properties of the class, or other conditions. Note, in particular, there can be no reference to any environmental object, such as *app* or *global*, as these may not be available in the context in which the condition is evaluated. Methods may not be invoked except for a small collection of system methods, deemed to be `conditionSafe`, that are guaranteed to produce the same result in any context. An example of such a method is the *month* method on *Date* to allow conditions such as the following.

Copyright Jade Software Corporation Limited, 2007.

```
monthHas30Days(day: Date):Boolean condition;
constants
    April = 4;
    June = 6;
    September = 9;
    November = 11;
begin
    if day.month() = April then
        return true;
    elseif day.month() = June then
        return true;
    elseif day.month() = September then
        return true;
    elseif day.month() = November then
        return true;
    else
        return false;
    endif;
end;
```

NOTE: The *monthHas30Days* condition could have been written using a single `return` statement but by using `if..elseif..endif` the code is easier to read and maintain.

A condition can be invoked in a method in which it behaves just like a normal method returning a *Boolean*, but with the additional advantage that anyone reading the code knows the method can have no side-effects. However, the most useful application of conditions is when they are used as a constraint, as explained in the next section.

Copyright Jade Software Corporation Limited, 2007.

What is a Constraint?

A constraint is a condition that is applied to a reference with an inverse so that when the manual side of the inverse is set, the automatic side is set only if the condition is satisfied.

Building on the *isInterestBearing* condition from the previous section, consider a banking system shown in Figure 1.

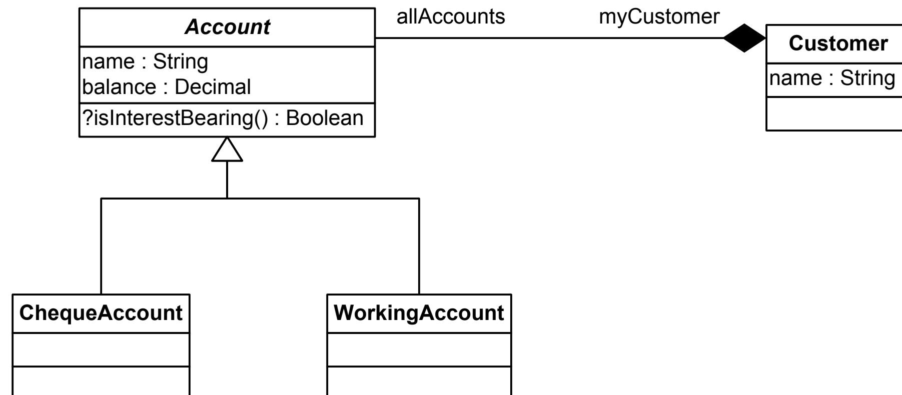


Figure 1: Simple banking system without using constraints

The abstract *Account* class has attributes *name* (of type *String*) and *balance* (of type *Decimal*). The *Account* class has real subclasses *ChequeAccount* and *WorkingAccount*. The *Account* class also has a *myCustomer* reference of type *Customer*. The *Customer* class has an attribute *name* (of type *String*) and a collection *allAccounts* (of type *AccountByNameDict*) with an inverse *myCustomer* of type *Account*. The *AccountByNameDict* collection type is a member key dictionary with membership *Account* using the account name as a key.

Suppose that at the end of each month our application was required to calculate the interest and add it to the balance of every *Customer*'s account that was eligible for interest. This could be achieved with a method on *Customer*.

```
addInterestToAccounts();
vars
    acc : Account;
begin
    foreach acc in allAccounts
            where acc.isInterestBearing do
        acc.addInterest;
    endforeach;
end;
```

However, there is a more elegant solution possible by using constraints and maintaining two collections on *Customer*, *allAccounts* and *allInterestBearingAccounts* both of type *AccountByNameDict* with the inverse *myCustomer* on *Account* as shown in Figure 2.

Copyright Jade Software Corporation Limited, 2007.

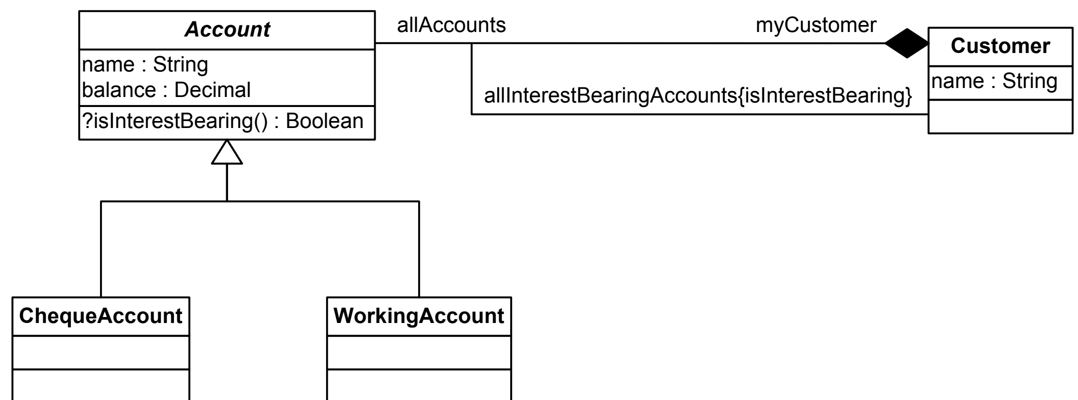


Figure 2: Simple banking system implemented using constraints

When the reference *allInterestBearingAccounts* is set up it is associated with the condition *isInterestBearing,* which imposes a constraint on the reference as shown in Figure 3. Constraints are only allowed on automatic (not manual or man-auto) references. When the manual side of the inverse is set, the corresponding constrained inverse is set only if the condition is true (in addition to any membership limits on the inverse). In our example, that means if the assignment statement

```
vars
    account : Account;
    customer : Customer;
begin
    …
    account.myCustomer := customer;
```

is executed, then the *Account* object is added to the *allAccounts* collection of *Customer* but is only added to the *allInterestBearingAccounts* if the condition *isInterestBearing* is true (that is, the balance is at least $5,000). Each time the balance is changed the condition will be re-evaluated and the account removed or added from the collection to maintain the constraint. The previous method can now be changed to use the *allInterestBearingAccounts* collection.

```
addInterestToAccounts();
vars
    acc : Account;
begin
    foreach acc in allInterestBearingAccounts do
        acc.addInterest;
    endforeach;
end;
```

Copyright Jade Software Corporation Limited, 2007.

Note that the collection *allAccounts* guarantees that an inverse is always possible when the statement

```
account.myCustomer := customer;
```

is executed. Without this collection, if the account was not interest-bearing then a 1223 (*Cannot Establish Inverse*) exception would be raised. At least one inverse back from the *Customer* object to the *Account* object is required in order to maintain the referential integrity of the system. For example, if there was no such inverse and the parent *Customer* object was deleted, then there would be no way for JADE to be able to locate the corresponding children *Account* objects to update the *myCustomer* reference. The result would be a loss of referential integrity with *Account* objects still having references to the deleted *Customer* object.

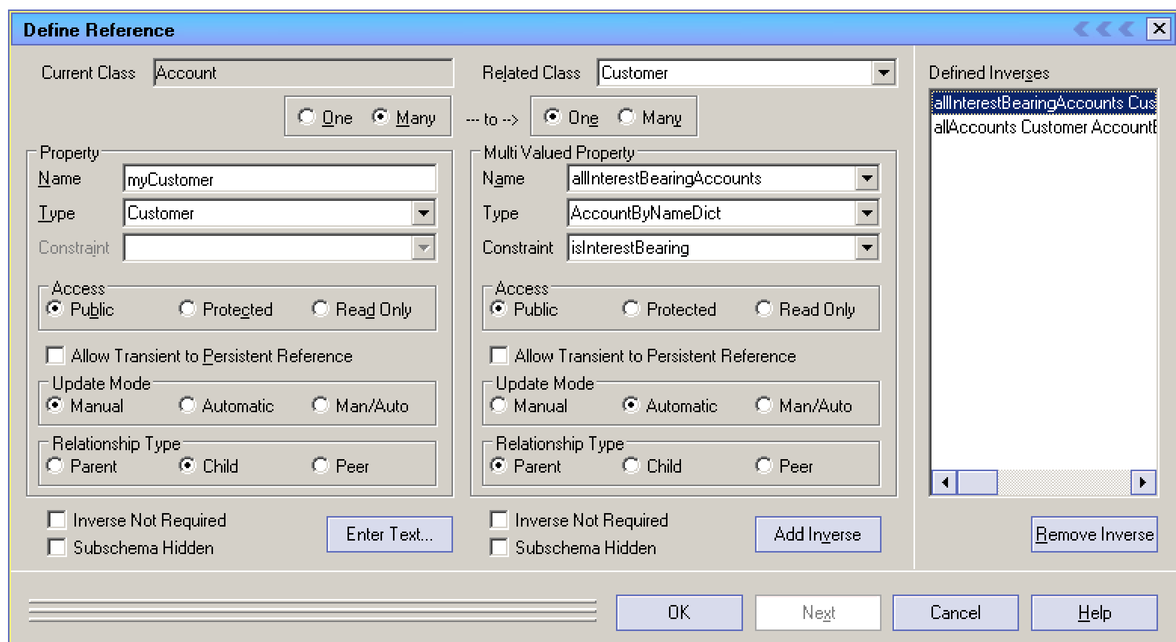


Figure 3: Defining the allInterestBearingAccounts reference

A closer inspection of the dialog box in Figure 3 shows that there is an "*Inverse Not Required*" check box that allows you to specify that the absence of an inverse should not raise a 1223 (*Cannot Establish Inverse*) exception in such circumstances. Normally you would not check this box as you could end up with problems similar to that discussed above, but in a later section we discuss an example where this is desirable to provide an efficient solution.

Let us now consider what happens if the code of the condition *isInterestBearing* is changed, for example to:

```
isInterestBearing():Boolean condition;
constants
    MinBalanceForInterest = 10000.00;
begin
    if isKindOf(WorkingAccount) then
        return false;
    endif;
    return balance >= MinBalanceForInterest;
end;
```

Copyright Jade Software Corporation Limited, 2007.

The change means that a *WorkingAccount* does not earn interest and a *ChequeAccount* needs to have a balance of $10,000 before it earns interest. Note that the condition invokes the *RootSchema* condition *isKindOf* to determine if the *Account* object is of type *WorkingAccount*. When the condition is changed and saved you will be warned that the change will require a reorg and given the opportunity to not make the change. After the change is compiled and the reorg initiated, the *allInterestBearingAccounts* collections are updated to conform to the new constraint. In this case, it will involve removing all *WorkingAccount* objects from the collection and removing *ChequeAccount* objects that satisfied the old condition but not the new one (that is, those with balances between $5,000 and $10,000).

Copyright Jade Software Corporation Limited, 2007.

Constraints versus Membership?

In addition to there being a constraint on a reference that is only established if a condition is true, constraints also exist for collection references that result from the membership type of the collection. These two concepts have some interaction and it is instructive to understand their relative advantages and disadvantages. Consider our previous example but this time with the model of Figure 4.

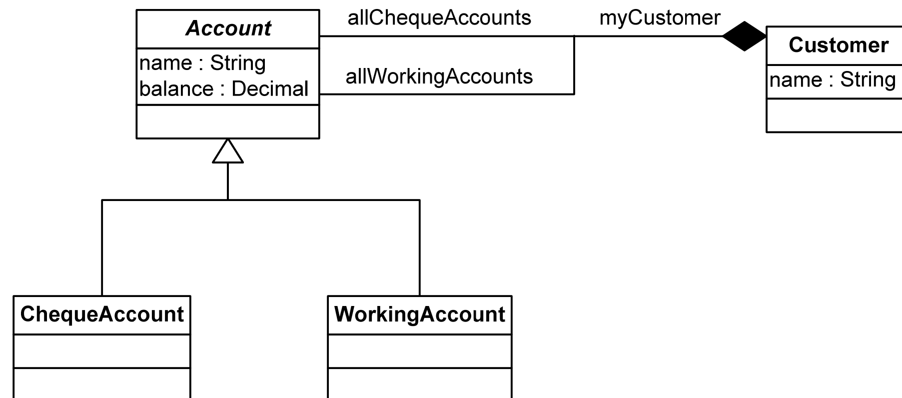


Figure 4 Simple Banking System using collection membership

Again there are two collections on *Customer*, *allChequeAccounts* and *allWorkingAccounts* with the inverse *myCustomer* on *Account,* but this time the types of the two collections are different. The collection *allChequeAccounts* is of type *ChequeAccountByNameDict* with membership *ChequeAccount* while *allWorkingAccounts* is of type *WorkingAccountByNameDict* with membership *WorkingAccount*. Both collections are member key dictionaries keyed on the *Account* name.

Now when the assignment statement

```
account.myCompany := customer;
```

is executed, JADE will attempt to add the account to both collections but it will only go into one because of the membership of the collections. If it is a *ChequeAccount,* it can be added to a collection of type *ChequeAccountByNameDict* and is thus added to the *allChequeAccounts* collection but it is not of the correct type to be added to the *allWorkingAccounts* collection. All accounts are either type *ChequeAccount* or *WorkingAccount* (remember *Account* is abstract so there can be no instances), so each account goes into exactly one collection and an inverse is always established.

This situation has a number of similarities to collections controlled by constraints. In fact we could have modelled this using constraints, by having the membership of the two collections being of type *AccountByNameDict* and having two conditions *isChequeAccount* and *isWorkingAccount* defined on *Account*. The two references *allChequeAccounts* and *allWorkingAccounts* would then have the constraints *isChequeAccount* and *isWorkingAccount* respectively applied.

Copyright Jade Software Corporation Limited, 2007.

Which of these two models is to be preferred? The membership approach would be more efficient, as the decision to add an account to a collection is a fast membership test performed by the JADE system, whereas the constraint approach will require a JADE method call. However the constraint approach is much more flexible and can deal with cases that cannot be handled by membership tests alone. An added advantage of the constraint approach is that it does not require a proliferation of *Collection* types whose only purpose is to limit the membership of collections.

Copyright Jade Software Corporation Limited, 2007.

"Inverse Not Required" example

As a final example of the power of constraints, we include the example we promised where the "*Inverse Not Required*" option is used. The outline of the model is shown in Figure 5 and involves a *Company* with a collection of *Clients,* each of which has made a number of *Orders*. The references between *Company* and *Client* objects are normal one-to-many references. Each of the other references has constraints that depend on the attribute status of an *Order* object. This may have one of the values *NotReady*, *ReadyToShip,* and *Shipped*. There are three conditions defined on class *Order* that test the value of its status as given in the following table.

Name of condition on *Order*	**Possible values of attribute status**
isReadyToShip	*ReadyToShip*
isNotShipped	*NotReady* or *ReadyToShip*
isShipped	*Shipped*

The collections *shippedOrders* and *pendingOrders* on *Client* (both with inverse *myClient* on *Order*) are constrained by condition *isShipped* and *isNotShipped* respectively. All *Orders* can be accessed from the *Company* object via a *Client* object and one of the two collections.

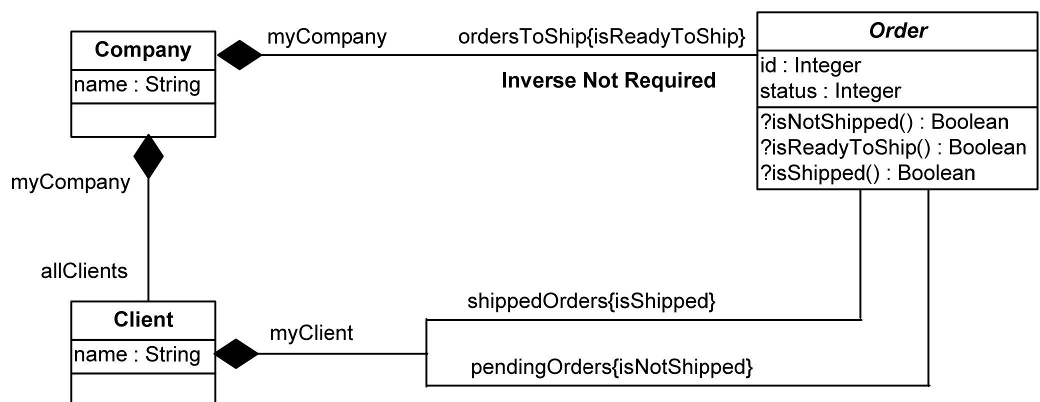


Figure 5 System illustrating "Inverse Not Required"

Suppose the design requires a process to run once a day to find all of the *Orders* that are ready to ship and do whatever is necessary to ship them and change their status to *Shipped*. This would be possible by accessing all *Orders* from the *Company* object via a *Client* object and searching the *pendingOrders* collection for suitable *Orders*. As the status was updated, the *Orders* would move to the *Client* objects *shippedOrders* collection. However, a more efficient design is possible by maintaining a collection of *Orders* that are ready to ship on the *Company* object.

Copyright Jade Software Corporation Limited, 2007.

The collection *ordersToShip* on *Company* (with inverse *myCompany* on *Order*) is constrained by the condition *isReadyToShip* and contains only those *Orders* that are ready to ship. This might be expected to be only a small percentage of the total number of *Orders* in the database. The reference *myCompany* has been marked as "*Inverse Not Required*" because for most *Orders*, even though *myCompany* will reference the *Company* object, there will be no inverse referencing the *Order* object from the *Company* (only those *Orders* with status being *ReadyToShip* will have such an inverse via the *ordersToShip* collection). If *myCompany* was not marked "*Inverse Not Required*", we would need to maintain another collection (say *allOrders* on *Company* with inverse *myCompany* on *Order*) whose only purpose was to maintain the referential integrity. This collection would contain every *Order* in the database and would be expensive to maintain, with very little benefit. As the nightly process traversed the *ordersToShip* collection and changed the status to *Shipped*, *Order* objects would be both removed from the *ordersToShip* collection on the *Company* object and moved from the *pendingOrders* to *shippedOrders* collection on a *Client* object.

The only issue that arises as a result of the missing inverse from the *Company* object to an *Order* object is that any change to *Company* that needs to change *Order* objects will only be able to access *Order* objects that are ready to ship. In our design, this is not an issue.

Copyright Jade Software Corporation Limited, 2007.

Summary

Conditions are simple methods returning *Boolean* results that do not have side effects and can be used as a flexible way of constraining the automatic side of inverses. This paper has shown a number of examples of how they can be used to produce efficient and easy-to-maintain JADE models. Although all the examples presented involve one-to-many inverses, they can also be used in many-to-many and one-to-one inverses.

Copyright Jade Software Corporation Limited, 2007.