INFO213 Tutorial 04 (Part 2) - Instructions

Tutorial Objectives

The goal of the second half of tutorial is to introduce the method of defining collections with inverse references, automatic membership updating, and conditional/constraint-based collection membership. You will be able to accomplish the following after completing the session:

- Use conditional instructions for directing the flow of code.
- Use Abstract classes, Superclasses, and Subclasses.
- Define automatic inverse relationships in collections.
- Define conditional collections membership based on a constraint.

Exercise 0: Preparing Development Environment

Here, we will continue developing the '**SimpleBankModel**' code that you finished in the last half of the session.

Downloads Needed

- If you have not completed the previous section of this tutorial, you can download the complete solution from **INFO213 Tutorial 4 (Part 1) - Solutions** on **Week 4** and import it into JADE.
    - Please keep in mind that loading the downloaded SimpleBankModel schema may cause part of your code in the SimpleBankModel schema to be overwritten if it differs from the solution code.

Exercise 1: Conditional, Break, Continue, And While Instructions

To cover all our **basic programming** needs, here we introduce the remaining few instructions that have not been explicitly discussed in the lectures.

- Add the following method in the **JadeScript** class. Please note the following:
    - The **while** instruction is an iterative instruction which in this case is intended to loop until we reach the stop value read from the prompt.
    - The **if ... then ... elseif ... then ... endif** instruction creates a cascade of conditions and corresponding actions.
    - The **continue** instruction skips the rest of the code in the loop body while the **break** instruction exists the loop all together.

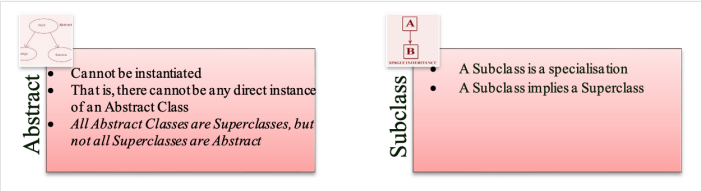
```
breakAndContinue();

vars
    i: Integer;
    stop: Integer;
begin
    read stop;

    while i < stop do
        i := i + 1;
        write i.String & " out of " & stop.String;

        if i = 3 then
            write Tab & "Don't like this number...";
            continue;
        elseif i > 8 then
            write Tab & "Can't go past 9...";
            break;
        endif;

        write Tab & "Done with " & i.String;
    endwhile;
end;
```

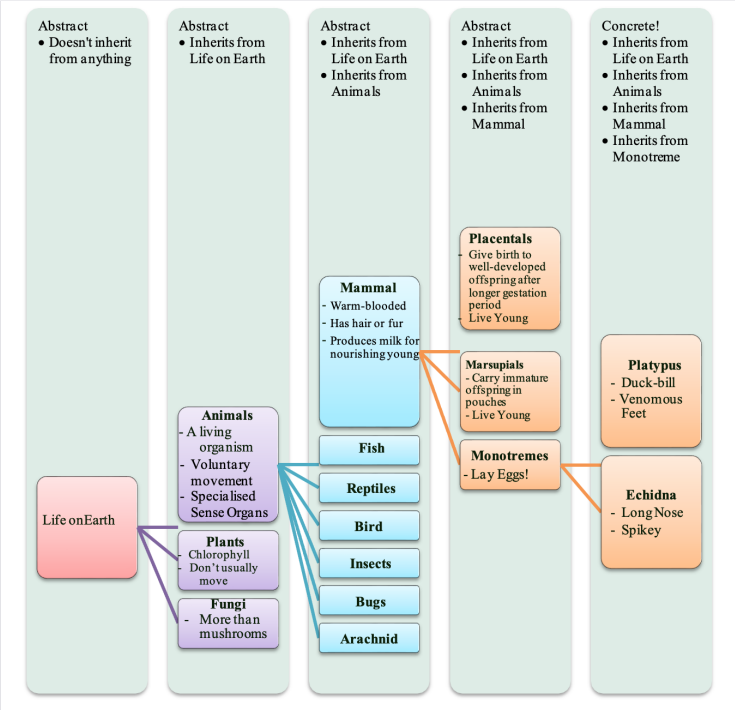
- Start the debugger to step through this method.
  - Instead of using F9 to launch the method, start the **debugger** by pressing **Shift+F9**.
  - Step through a few iterations and observe how the value of variable **i** changes.
  - Please note that the **continue** and **break** instructions can also be use with the other iterative instruction **foreach**.

Exercise 2: Abstract Classes, Subclasses, And Superclasses

Information - Classes

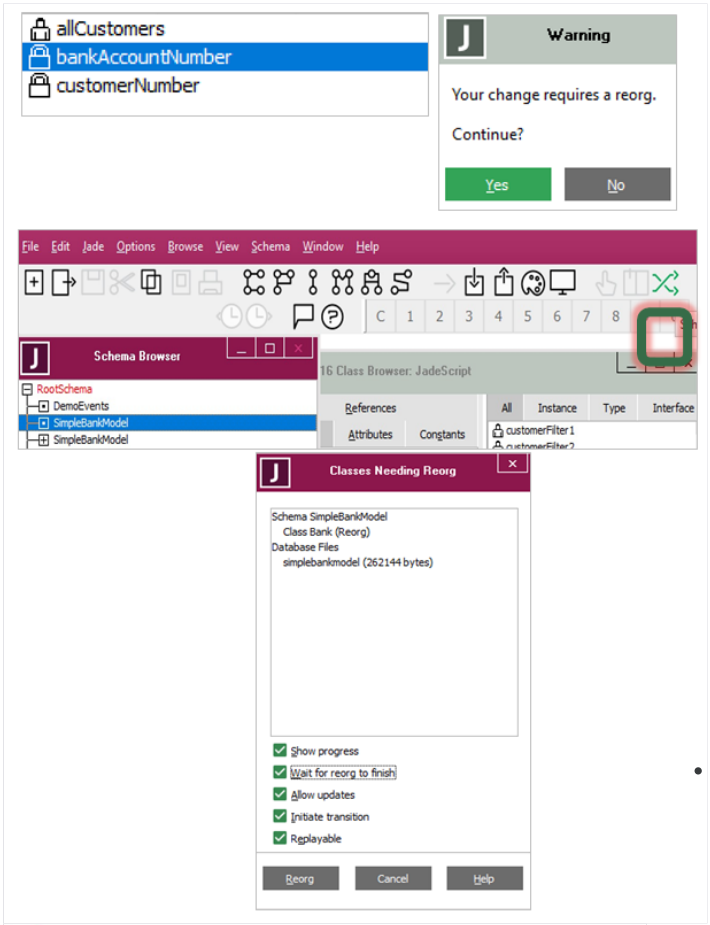
- We will define three new classes in this exercise to represent various types of bank accounts.





How To Work With Classes

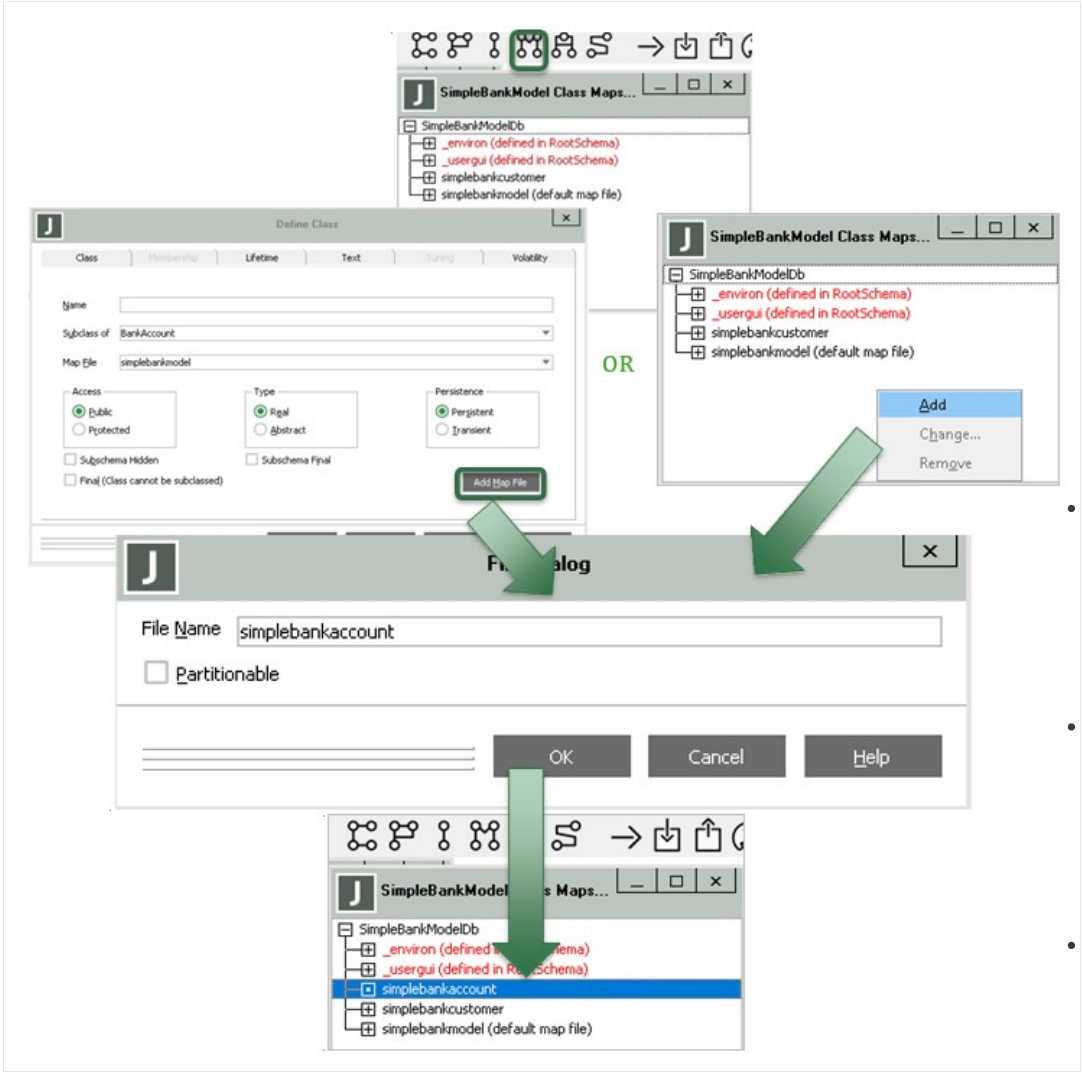
- Start by adding a new **Integer** type **protected** attribute called '**bankAccountNumber**' to the '**Bank**' class.
  - **Your code may require a Reorg after adding this attribute. Please do so.**



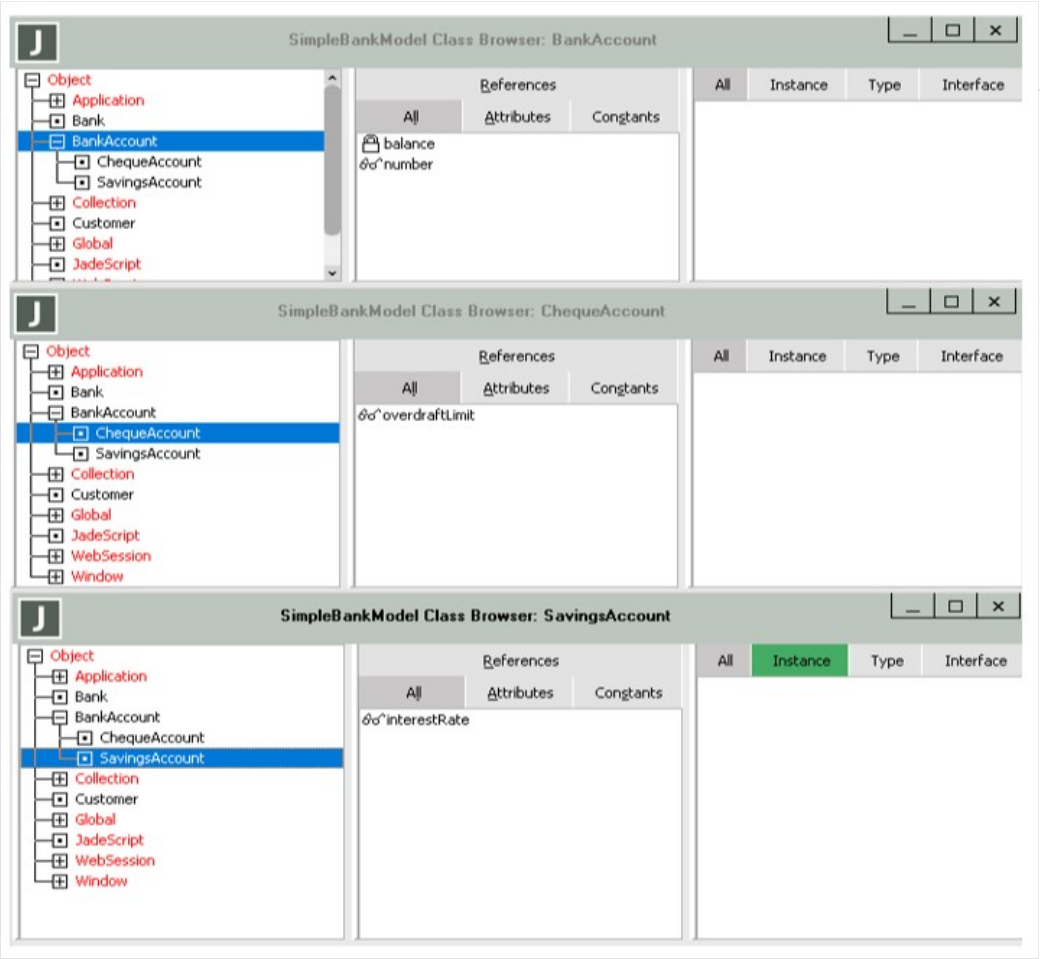
- Still in the '**Bank**' class, create a method to get a new bank account number ('**nextBankAccountNumber**'), which is logically equivalent to the '**nextCustomerNumber**' method.

```
/*
 * Increment the bank account number and return the new value.
 */
nextBankAccountNumber(): Integer updating;

begin
    // Insert your code to increment the account number.
    return self.bankAccountNumber;
end;
```

- Create three new classes: '**BankAccount**', '**ChequeAccount**' , and '**SavingsAccount**'.
  - The '**BankAccount**' class is a **superclass** for the other two.
  - The '**BankAccount**' class should be defined as an **abstract** class, subclass of '**Object**'.
  - The **mapfile** for the '**BankAccount**' class is '**simplebankmodel**'
  - The **mapfile** for '**ChequeAccount**' and '**SavingsAccount**' is '**simplebankaccount**'.
  - You may need to check the required mapfiles exist before you define the classes and add them if necessary.



- Add a **read-only Integer** attribute called number to the '**BankAccount**' class to store the account number.
- Add a **protected Decimal (12, 2)** attribute called '**balance**' to the '**BankAccount**' class to store the current balance value.
- Add a **read-only Decimal (12, 2)** attribute called '**overdraftLimit**' to the '**ChequeAccount**' class.
- Add a **read-only Decimal (12, 2)** attribute called '**interestRate**' to the '**SavingsAccount**' class.

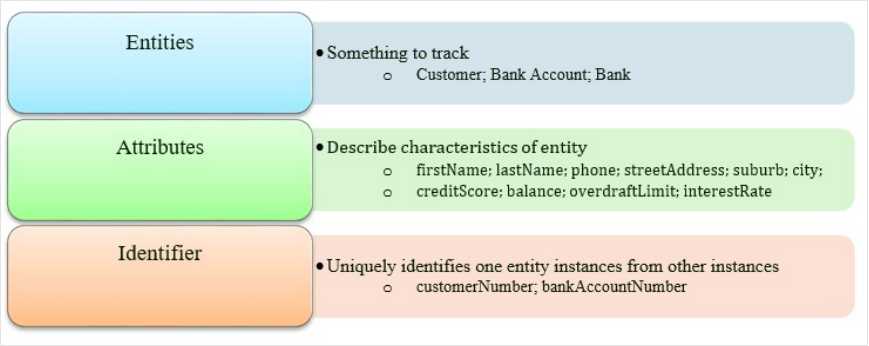


Exercise 3: Inverse Relationships

We will define a collection (dictionary) named '**allBankAccounts**' in this exercise.

This collection will have **automatic inverse maintenance** and it will be used to maintain track of all accounts in the '**Bank** ' root object.

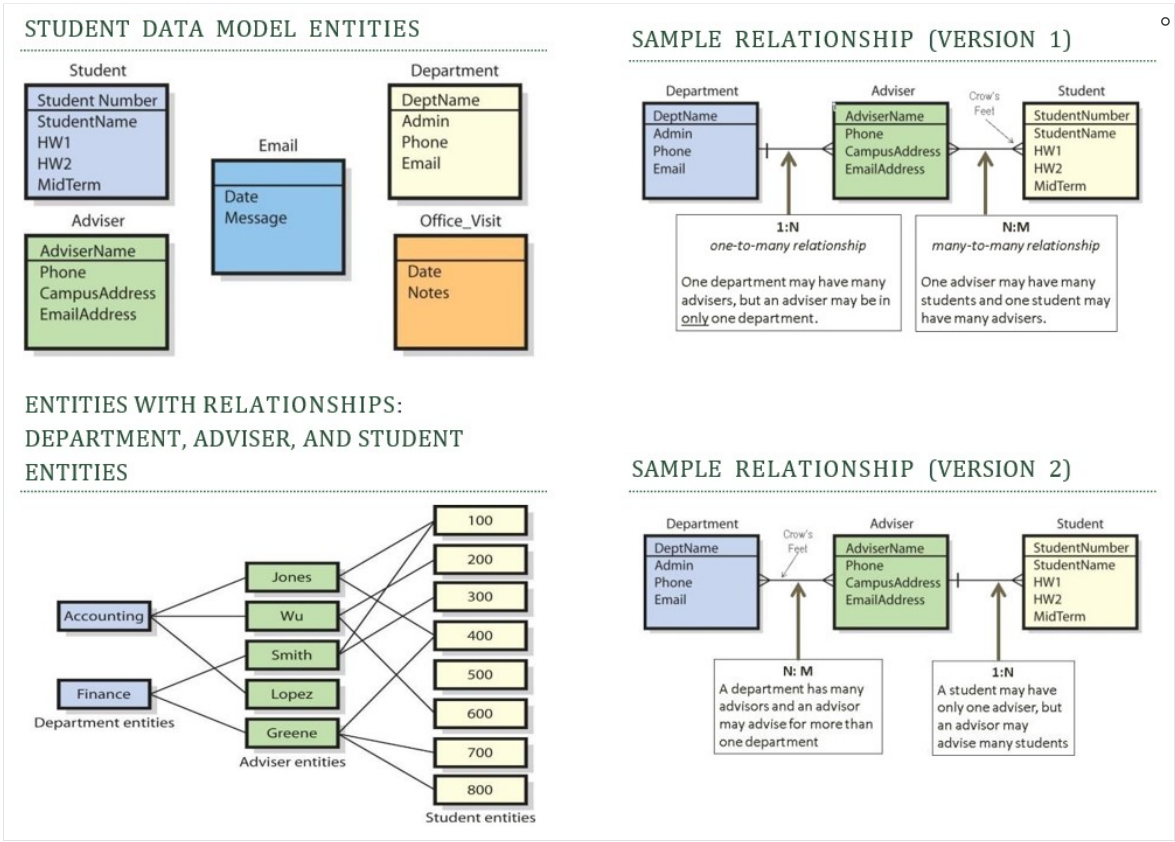
Information - Entity-Relationship



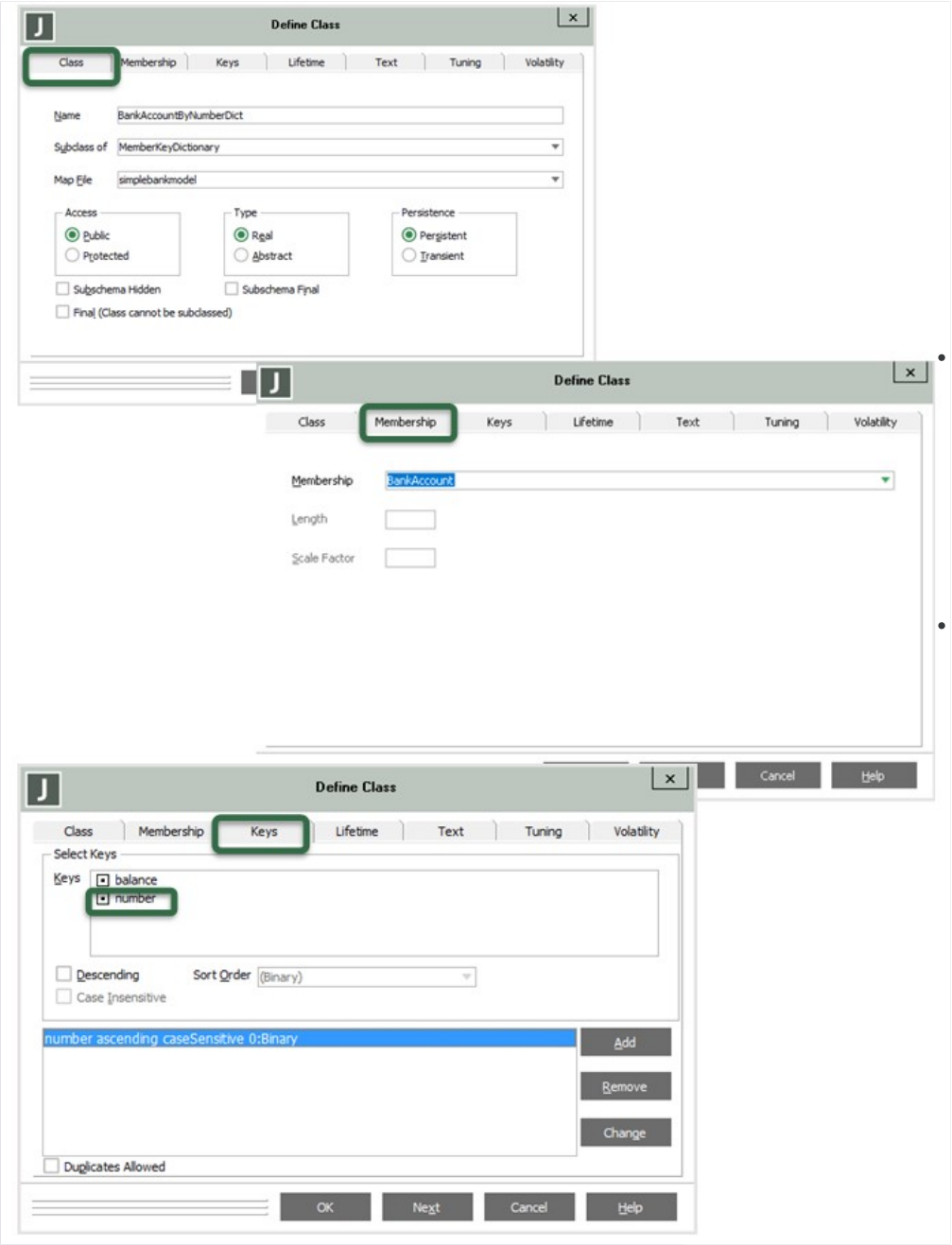
Example

How To Work With Collection

- Define the accounts collection (dictionary) which will be appeared as a member of the '**Bank**' root object.



- Create a new class called '**BankAccountByNumberDict**' as a subclass of '**MemberKeyDictionary**' that is mapped to the '**simplebankmodel**' mapfile.
- On the **Membership** tab, select the '**BankAccount**' class in the combo box.
- On the **Keys** tab, select the '**number**' attribute as the collection sorting key with the **default (binary) sorting order**.



Information - Inverse Relationships

- The one-to-many relationship between a customer and the customer's bank accounts will be created using a one to many relationship – that is, one instance of class customer may have 0 to infinite bank accounts attached to it.
- If the bank account is created and its reference is set to an instance of class Customer, that particular instance of Customer must contain the bank account in its allBankAccounts collection. This is similar to a referential integrity in a relational database.

Inverse References ensure that we can enforce this consistency.

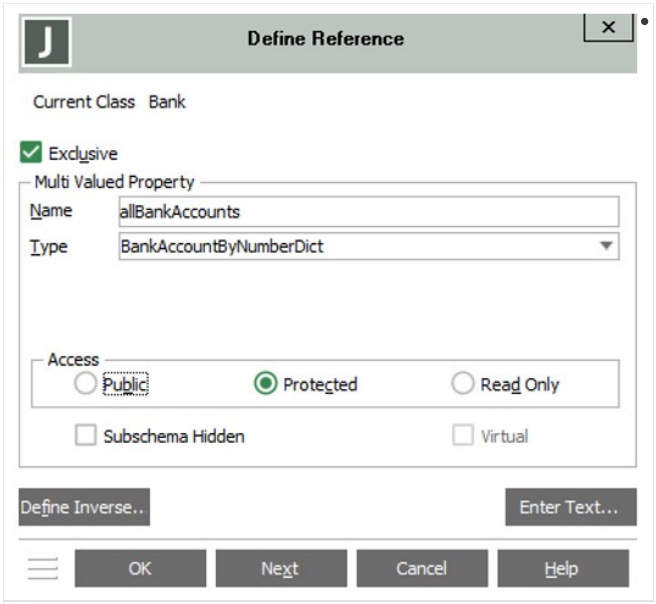
- myCustomer is the inverse of allBankAccounts; and
- allBankAccounts is the inverse of myCustomer

Inverse references are great, because:

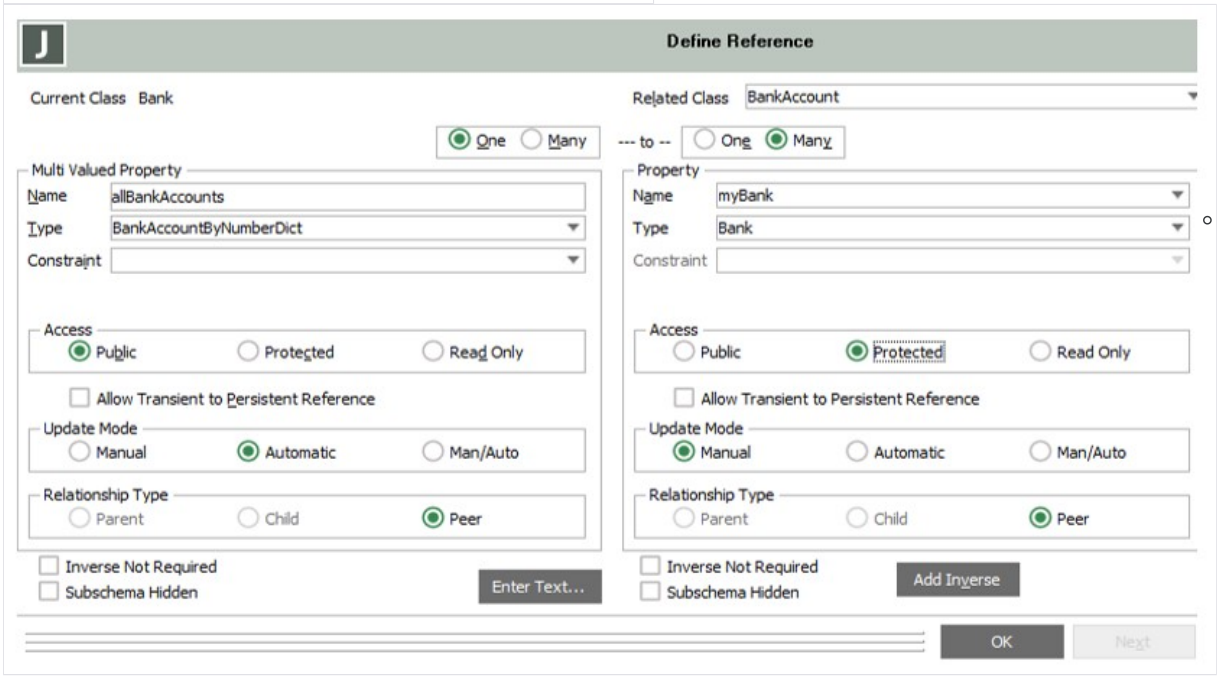
- You write code for an object at one end of the relationship only
- The object or objects at the other end of the relationship are maintained consistently
- Less code: = fewer problems
- **DRY** principles of coding: **D**on't **R**epeat **Y**ourself

How To Work With Inverse Relationships

- Start defining a new reference in the '**Bank**' class like below, but do not rush to click the OK button:



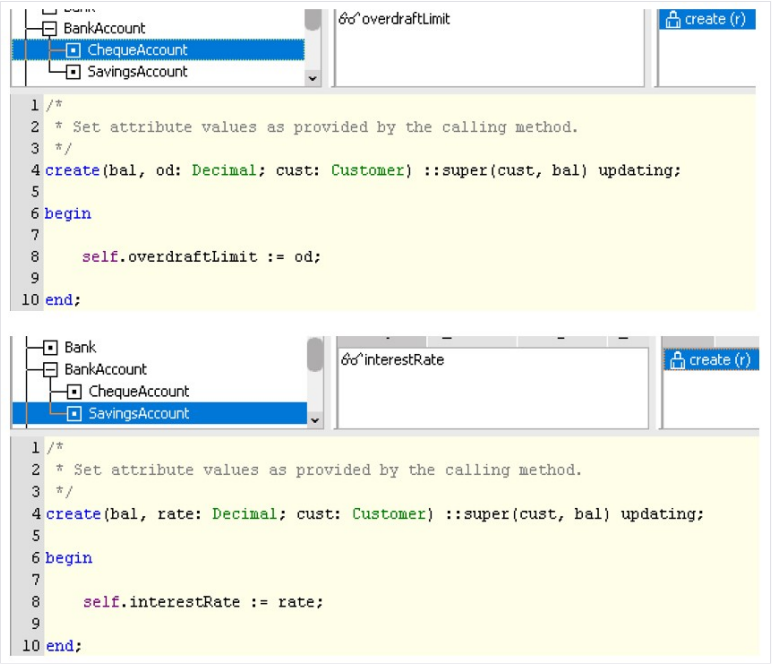
- Click the **Define Inverse** button which will bring up the next Define Reference dialog.



  - Here, we define **one-to-many** **relationship** between the '**Bank**' and '**BankAccount**' classes.
    - The **Manual** update mode on the '**BankAccount**' class side in combination with the **Automatic** update mode on the '**Bank**' class side means that as soon as a '**Bank**' instance is assigned to the '**myBank**' reference in the '**BankAccount**' instance, this '**BankAccount**' instance is **automatically** added to the '**allBankAccounts**' collection in the '**Bank**' instance.
    - For simplicity of our code we define the '**allBankAccounts**' access as **public**.
      - This point will be reviewed when we create a constructor for the '**BankAccount**' class.
  - Note that '**myBank**' reference in the '**BankAccount**' class appears automatically after you are done with this dialog.
- Define a **Protected** reference named '**myCustomer**' in the '**BankAccount**' class, of type '**Customer**'.
- Define a constructor '**create**' in the '**BankAccount**' class.

```
1  /*
2   *Initialise the account number and the reference to the Bank Account
3   */
4  create(cust : Customer; balance : Decimal) updating;
5
6  begin
7
8      //Assign a new unique number to the customer.
9      self.number := app.myBank.nextBankAccountNumber();
10
11     //The following line will not compile if commented in
12     // self.myBank.allBankAccounts.add(self);
13
14     //This customer's Bank instance
15     self.myBank := app.myBank;
16
17     //This account's myCustomer reference
18     self.balance := balance;
19
20     //This account's balance reference
21     self.myCustomer := cust;
22
23 end;
```

- **Q: Why would Line 12 cause a compiler error if uncommented?**
- Now we will reimplement the '**create**' method as a parameterised constructors for the '**ChequeAccount**' and '**SavingsAccount**'.
  - Both the '**ChequeAccount**' and the '**SavingsAccount**' now include '**super()**' which calls the '**create**' function from the '**BankAccount**' class.
    - **Q: In what order are the 'create' methods called? Subclass first or Superclass first?**



- To verify this code works, add two methods called '**addChequeAccount**' and '**addSavingsAccount**' in the **JadeScript** class. Create one instance of each of the '**BankAccount**' subclasses and their properties in these methods.
- Hints:
  - You can use the '**Customer**' instances collection to get access to the first '**Customer**' object.
  - You will need to call the '**initialize**' code again.
- Use the **Schema Collection Inspector** to verify the newly created bank accounts show in the '**allBankAccounts**' collection in the '**Bank**' class.
- Use the **JADE Help** to learn about the difference between defining a simple collection (without automatic inverse maintenance) and an automatically maintained relationship.

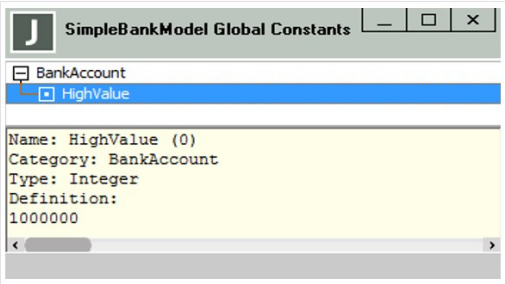
Exercise 4: Conditional Collections Membership

JADE offers a plethora of various efficiencies in database management as we have seen so far, for example, with the automatic inverse references.

In this exercise, we will examine another very powerful feature which allows automated collections maintenance based on a constraint imposed on a collection. For example, we want to maintain a collection for all high-value accounts.

This exercise requires reliance on the JADE Online Help System to discover how to complete the tasks. The instructions are therefore not given in as much detail as usual.

- Create a global constant '**HighValue**' (with the value of 1000000) under the '**BankAccount**' category.
    - The topic of defining global constants has **not** been presented in lectures or tutorials. However, as part of this exercise**, you are required to explore the JADE documentation to find out how to define a global constant**.
        - Search the JADE online help system for these terms: "adding a global constant".
    - On the topic of good practices, the idea of adding a global constant provides a more maintainable solution than using numeric values in code.
- Once you have defined the required constant, it should appear in the **Global Constants** browser as follows:



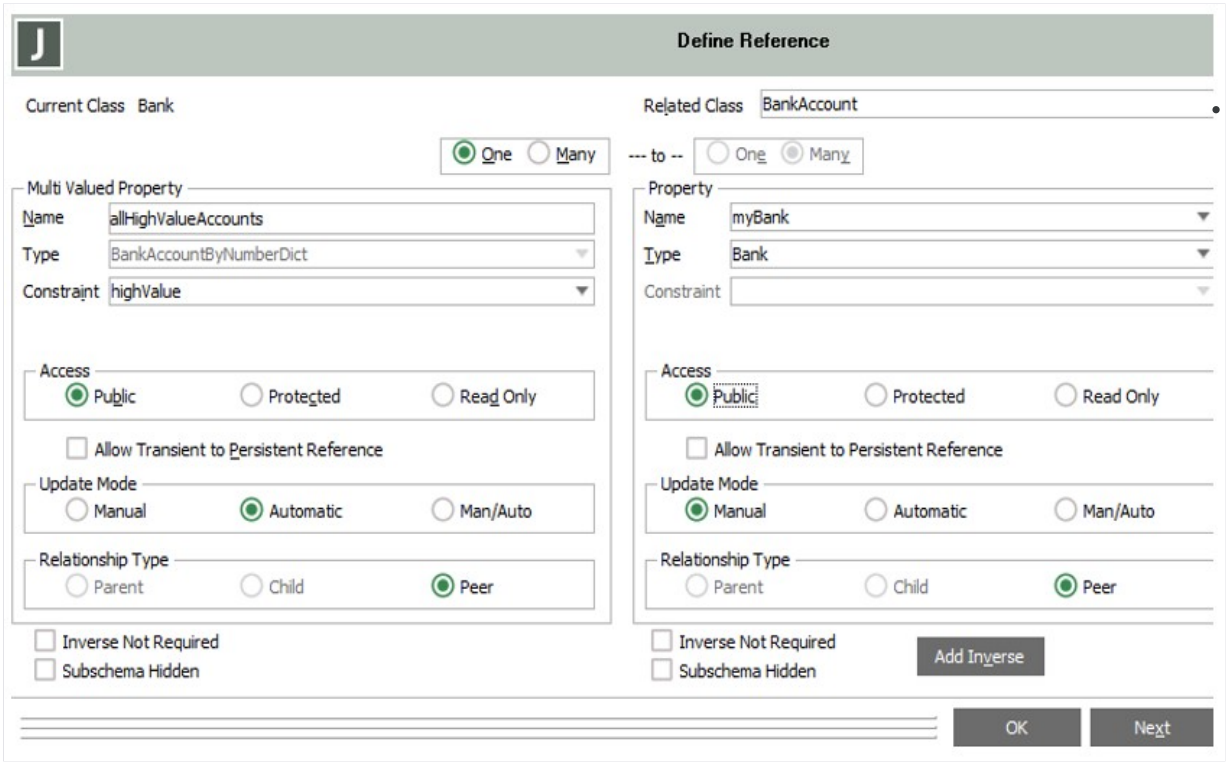
- Create a **condition method** (for the '**BankAccount**' class) called '**highValue**' by selecting the **New Condition** option from the **Methods** menu or from the right-click context menu from the **Methods** pane in the **Class Browser** .
    - Condition methods are usually simple declarative methods which return a **Boolean** value.
    - Condition methods do not allow the use of method parameters or local variables.
        - The following code is incomplete. Add your code and remember to use the global '**HighValue**' constant you defined earlier:

```
1 /*
2  * Return true if the account is worth at least $1,000,000.00.
3  */
4 highValue():Boolean condition;
5
8     return false;
9 end;
```

◄ INFO213 Tutorial 04 (Part 1) - Solutions

Jump to...

INFO213 Tutorial 04 (Part 2) - Solutions ►



- **Note the 'highValue' constraint (as defined earlier in the 'BankAccount' class) on the 'Bank' class side. This change may require a reorg.**

- Please verify the '**allHighValueAccounts**' collection gets updated every time a new account with a balance equal to or higher than a million dollars. If your database already has high-value accounts which would match the '**highValue**' condition, the database will require a reorg and the '**allHighValueAccounts**' collection will be updated. Otherwise you can edit the ' **addChequeAccount**' or '**addSavingsAccount**' method you created for the previous exercise.

- **Q: At this stage, how would you go about verifying your code works without adding any extra code (only by using the object inspector)?**

Last modified: Tuesday, 25 January 2022, 3:37 PM