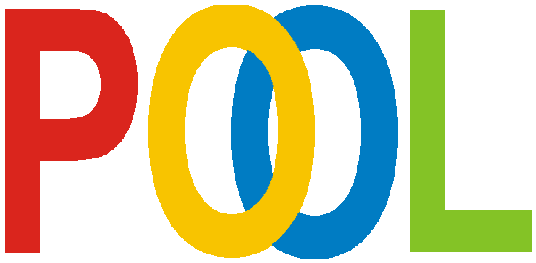

Portable Object Oriented Language

Tutorial, Part 2: Object Oriented Programming

Revision Level: See Revision Index

SIEMENS VDO
Automotive

VDO-Straße 1
12345 Babenhausen
Germany

bfk

Büro For Datentechnik GmbH
D-35418 Buseck
Germany

1 Contents

1 CONTENTS 2

2 REVISION INDEX 3

3 INTRODUCTION 5

3.1 Important instructions on how to work with the tutorial 5

3.2 A first example 6

4 INTRODUCTION INTO OOP 13

4.1 Why do we use object oriented programming? 13

4.2 Basic OOP elements 13

4.2.1 Encapsulation 13
4.2.2 Classes and objects 14
4.2.3 Properties of objects 17
4.2.4 Methods of objects 17
4.2.5 Inheritance 18
4.2.6 Polymorphism 20

5 IMPLEMENTATION OF OOP IN POOL 21

5.1 Encapsulation and creation of libraries 21

5.2 Classes, objects, and library import 24

5.3 Properties 27

5.4 Methods 27

5.4.1 Functions 27
5.4.2 Procedures 28
5.4.3 Constructors 29
5.4.4 Constructor with parameters 32
5.4.5 Destructors 33
5.4.6 Copy Constructor 36

5.5 Inheritance 42

5.5.1 Parent Class 42
5.5.2 Access rights 43
5.5.3 Overriding properties 48
5.5.4 Overriding methods 48
5.5.5 Overriding constructors 49

5.6 Polymorphism 50

5.6.1 Virtual methods 50
5.6.2 Virtual constructors 63
5.6.3 Virtual destructors 63

6 IMPORTANT DIFFERENCES BETWEEN C++ AND POOL 64

7 SUMMARY 66

ATTACHMENT 67

A1 Bibliography 67

A2 Index 68

Revision Index

Date	**Author**	**Rev.**	**Ref.**	**Type**	**Description**
2003-05-23	Harald Ebert	0.91.40	div	cont.	Revision of the English version
2002-11-06	Thomas Locker	0.91.30	div.	cont.	Comments removed for the translation. Examples and figures: Designations changed into English for the translation.
2002-11-06	Thomas Locker	0.91.20	div.	cont.	Comment on SMK in chapter 4. Latest German version.
2002-11-01	Uwe Kühn	0.91.10	div.	cont. auth. auth.	Content was revised Text was edited Formatting, form templates
2002-10-17	Thomas Locker	0.91.00	-	-	Initial Revision

Acronyms:

AIDA

Automotive and Industrial Diagnostic Assistance. System that is used to implement computer supported diagnosis of control modules and bus systems.

GUI

Graphical User Interface.

MFC

Microsoft Foundation Classes. Microsoft's class library that allows the user to access operating system functions.

OOP

Object Oriented Programming (see appropriate chapter).

POOL

Portable Object Oriented Language. Object oriented programming language by BSK. Used for programming in the AIDA system.

SMK

Software Method Kit. Description of the programming guidelines by SiemensVDO.

VMT

Virtual Method Table. Takes care of the correct assignment of the virtual methods.

2 Introduction

2.1 Important instructions on how to work with the tutorial

The second part of the POOL tutorial covers object oriented programming (OOP) with POOL and is intended for beginners of OOP, as well as for users switching from other OOP languages such as C++ or Java. Prerequisite for part 2 is a basic understanding of the POOL programming language and the mastery of the required tools. These fundamentals can be found in the first part of the tutorial.

Users switching from other programming languages can use this tutorial as a reference that includes examples and simply skip the first four chapters. Chapter 6 includes a table that lists the major differences between POOL and C++.

Beginners should thoroughly work through the individual sections including the exercises.

In order to be able to use the tutorial as a reference and to deepen the knowledge of important relationships, some facts are explained several times. You may skip sections with familiar contents.

Following an introductory example we will discuss the importance and the advantages of OOP in comparison to procedural programming.

Next, you will learn how to implement OOP in POOL and how the object oriented approach is used in practice. Illustrative examples will be added to the explaining text. The code examples are focused on important aspects to avoid confusion. That is not all methods are implemented in detail. In particular the prevention of user errors is omitted to the extent possible.

Exercises are following each individual section and should be solved independently. The solutions and examples in the document are limited to major changes in comparison to previous solutions and examples to improve clarity. For complete source codes please see the enclosed disk or CD. The assignment of files to documents is done via consecutive numbering.

2.2 A first example

We take a closer look at the operating principle of an ATM as an example for the object oriented point of view.

When you withdraw money from an ATM, you are not interested in the processes that run in the background. It is important to you to know that you have to enter your PIN number to gain access to the functions of the teller machine like obtaining your account balance or withdrawing cash.

To withdraw cash you simply enter the desired amount and your money will be disbursed. You will receive your money unless you have exceeded your credit limit. The programmer of the ATM has to implement the access to the software of the bank system. You, the customer, do not have to know exactly how the teller machine works nor how to charge a bank account. The user interface hides the quite complicated debiting process and the functions of the teller machine from you.

The restriction of a complex problem to the major components, in our case the entry of the desired amount and the disbursement of money is called abstraction. The result of this kind of abstraction is called model.

Street maps are another commonplace example for a model. A street map is a drastically simplified representation of reality. It is only possible to show all German interstates on a single page by omitting detailed information. Only by doing this it becomes possible to find the correct interstate for a certain itinerary. If all German streets were depicted on one map, you would not be able to have an overview and select the correct highway, apart from the fact that this map would have to be huge.

Once you have reached a certain town, though, you need a more detailed map. Though the map has a lower level of abstraction.

You will also need the different levels of abstraction to create complex programs. First, work on a possible solution on a high level of abstraction (interstate), next deal with the lower levels of abstraction (city map).

To map complex problems in one simple model there are two fundamental approaches.

The procedural (functional) point of view

In the procedural approach you are not working with objects such as ATM's. Instead, problems are solved through functions.

For this approach it is necessary to have a deeper understanding of how to withdraw money from a checking account

Let’s take for example the request for disbursement of a certain amount of money. First you need some functions to get your checking account balance and your credit limit from your bank's database. The next step is to check whether you can debit your account with the desired mount of money. Then the money can be disbursed, a process which also requires a number of steps such as take money from the safe, open door,

and push out money. Additionally, your bank account has to be charged with the withdrawn amount. This, of course, is also done using functions. From a functional point of view, all these tasks have to be done by the user, in other words by you through the call of functions. These functions have to be provided by the programmer. The abstraction is achieved by splitting the problem into individual function units or function modules.

This simplified example illustrates that executing a simple task like using an ATM will lead to complex processes with a variety of interfaces and functions, if a procedural solution approach is used. The relationships will become even more complicated, if you consider that money transfer, cash deposit and direct debit transactions can also influence the account balance.

You will rightfully object that in practice there is no need to execute all these steps in order to withdraw your money. This is due to the fact that, in real life, we work with objects such as ATM's quite naturally.

The object oriented point of view

How does the solution to our problem look like from an object oriented point of view? OOP achieves abstraction by mapping the involved entities of the real world to software objects. In the ATM example we limit these objects to three to keep things simple:

- The account holder (including his/her debit card)
- The ATM
- The account

As you can see we are already working with drastically simplified objects, since your debit card of course is also an independent object and even the ATM, too, can be subdivided into a variety of objects for a practical realization.

Each of these objects has a number of properties (attributes, features) and functions. We will implement the properties as variables later on, and the functions as so called methods.

Let’s take the account object as an example. It has, among others, the following properties:

- Account number
- Balance
- Credit limit

In the procedural point of view, everyone could now freely access these properties. In object oriented languages, the properties (variables) of an object are only accessible through the object's methods!

Thus, we need two methods for withdrawing and depositing, which can influence the account balance:

- Withdraw
- Deposit

So access to the account balance by another object is only possible using these two methods. Our account object will then look as shown in Fig. 1:

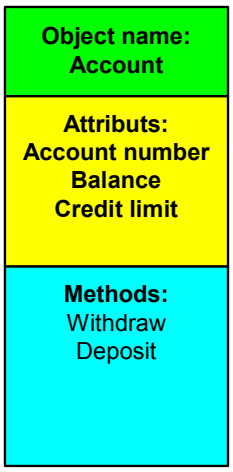

Figure 1: Account object

How does the process of withdrawing money look like in practice? First, we will have a closer look at the account and ATM objects.

The ATM object receives a request by the account holder object to pay a certain amount. In order to execute this order, the ATM calls the Withdraw method of the account object. The desired amount is passed to the method as parameter. The Withdraw method checks, whether the disbursement is possible, using the checking account balance and the credit limit. If a payment is possible, the bank account is debited with the desired amount resp. the amount is deducted from the account balance variable, and a return value indicates to the ATM object that a disbursement took place. If no disbursement is possible, then the ATM is also notified of this fact.

Thus, the inquiry whether charging the bank account is permissible is hidden from the user as opposed to the procedural approach.

With this method the user is relieved from the task of directly accessing the properties of the object. He/she only has to know the methods that are provided by the object and its parameters. Thus, these methods represent the interface between the various objects.

The separation between the interface and the property of an object is called **encapsulation** and is one of the most important principles of object oriented programming.

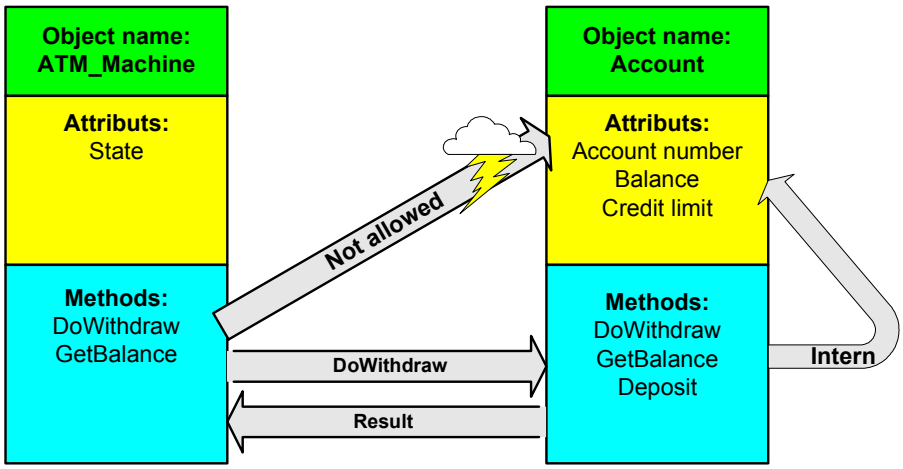

Figure 2: Encapsulation

The encapsulation principle is illustrated once again in Fig. 2. The account balance request was added to the account object. This method calculates the checking account balance. The ATM object can only access the attributes of the account object via the methods of the account object. A direct access is not allowed. The methods of the account object on the other hand can access and change their own attributes without any problem.

An advantage of encapsulation is the separation of interface and implementation. As long as the interfaces are sustained in regard to parameters and functionality, an object can be modified any time. To do this, it is not necessary to modify other objects.

Let's take a closer look at the interface between the account holder resp. the customer and the ATM. The ATM provides the user with a number of methods via a graphical user interface (GUI). It does not matter to the customer how this interface is created and administered, since these functions are encapsulated in the ATM object.

(We want to point out that graphical user interfaces like windows, buttons, etc. generally are objects as well).

All the customer needs to know is what kind of entries he/she has to make in order to withdraw his/her money. If the customer asks for an amount, the ATM object automatically calls the account's withdrawal method. If the withdrawal method sends the approval, the money will be disbursed. If the approval is denied because the credit limit is exceeded for instance, the customer receives an appropriate notice.

A big advantage of OOP is that individual objects can be replaced easily. A bank can easily decide to introduce a new generation of ATMs with a modified user interface, as long as the software interface of the ATM object to the Account object does not change. That is the ATM still has to use the methods of the account object to withdraw money.

It is also possible to change the account administration or the internal software any time, as long as the existing interfaces are sustained.

As you have probably noticed already, we did not make a distinction between pure software objects such as the account, interface objects such as the ATM, and real

objects such as the account holder. This shows one of the strengths of the object oriented approach: The representation of real entities as objects.

In Summary, OOP allows us to create simple models that represent real entities. The number of necessary interfaces is reduced by encapsulation, and programming is made simple by splitting the problem into sub-problems, which enhances maintainability and reusability of the code.

Note:

Not all problems are equally suitable for OOP. As a result, you have to think before you are working on a project whether a representation in objects is beneficial in solving your problem.

Exercise

Think about how the task of ordering and delivering food in a restaurant could look like in an object model.

Solution (one of many possibilities)

3 objects

- Customer
- Waiter
- Kitchen

Properties of the customer

- Hunger
- Favorite food
- Ordered food
- Cash

Methods of the customer

- Ordering food
- Eating
- Paying

Properties of the waiter

- Order slip
- Tip

Methods of the waiter

- Receive order
- Pass on order to kitchen
- Receive food from kitchen
- Bring food to customer
- Receive payment

Properties of the kitchen

- Food
- Equipment
- Order list
- Food status

Methods of the kitchen

- Receive order from waiter
- Prepare food
- Inform waiter that food is ready
- Give food to waiter

Based on this example you can clearly see the interfaces between the individual objects. Abstraction clearly frees ourselves from much more complicated sub-processes, like cooking food and limits the problem to the core components. The responsible chef could now divide the kitchen object into other objects, for example (chef, assistant, etc.), as long as the interface is not modified by this (and our waiter could be fired any time and replaced by a more capable one, if he/she does not do the job well).

In other words, with the help of this model you can approach the solution of the individual tasks separately, which means on different abstraction levels. The individual components could also be easily exchanged in the actual realization. Perhaps we will replace the waiter with a robot one day.

Exercise

Think of more abstraction examples that can be used to divide a problem into objects.

3 Introduction into OOP

3.1 Why do we use object oriented programming?

The necessity for OOP results from the steadily growing size and complexity of programs and the requirement to reuse code to save cost and time. In classical, procedurally created programs, reusing software was generally only possible at a high level of effort.

Due to the division of programs into objects with clearly defined interfaces the existing program code can be used in new projects at any time and extended if necessary. OOP and the so-called inheritance offer a big progress in comparison to the current programming concepts in particular for the extension of objects. With inheritance, a new object takes over the properties and methods of an old object and adds its own properties and methods.

Another advantage of OOP is that it is geared towards the human way of thinking. Humans interact with the environment in the form of objects and communicate with them via methods without understanding every detail of their operating principles. In order to use a TV for instance it is not necessary to know anything about communication technology or electronic circuits. The only thing one needs to be familiar with are the remote control and the power plug, which are the user interfaces to the TV. The complexity of the modern world becomes easy for us to comprehend and control, if we limit ourselves to the necessary methods of everyday objects.

This concept can also be found in OOP. You do not have to understand how an object works; you only have to know how to use it, how to apply it to your own projects, and how to extend it via derivation if necessary. An exception of course will be those objects that you will create yourself (just like the TV developer, since you can rightfully expect that he/she knows how the TV works).

3.2 Basic OOP elements

This section will take a detailed look at the individual OOP components that you learnt about in the introduction and explains them.

3.2.1 Encapsulation

As mentioned before, the combination of data (attributes, properties) with the related functions (methods) is part of the basic OOP concept. The data is mainly manipulated through methods. A direct access to the elements is intentionally prevented. This concept is called encapsulation.

With the definition of such an object you will receive an abstract data type that is called a class.

3.2.2 Classes and objects

A class defines a type of physical or logical object with the help of its properties (data) and its methods (functions).

You already learnt about another abstract data type, the structures (records), in the first part of the tutorial.

In contrast to a class, a structure only consists of data (properties). Functions that are used to manipulate this data have to be declared and implemented outside of the structure.

In order to be able to work with a structure it is necessary to create a structure variable. The same applies to classes. To use a class in a program, a class variable has to be created. This is also called the instance of a class; the process is called instantiation. The result of this instantiation is an instance or variable of a class. It is called the object.

Warning: A clear distinction is made in languages that are similar to Pascal, and therefore also in POOL, between an "object type" and an "object instance." However, developers like to neglect this differentiation and talk about "objects" in both cases. For didactical reasons this carelessness is catastrophic. An "object type" is called "object" in POOL and an "object instance" is called "instance" in POOL. However, for didactical reason we will use two different expressions in the tutorial. We use the terms that are used in everyday language i.e. usually we are "classifying" when we group things with the same properties so we use the term "class" for "object types" and when we are talking about an object we usually mean an specific existing entity (some thing you can touch) so we use the term "object" for "object instance". These terms are also used in C++ and Java.

We will once again use the ATM example. The ATM class represents all teller machines of the same type. An instance of the ATM class is an object that represents a real existing ATM in a bank. The teller machine can be clearly identified through its name or its identification number.

Another example is used in Fig. 3 to show the relationship in the account class:

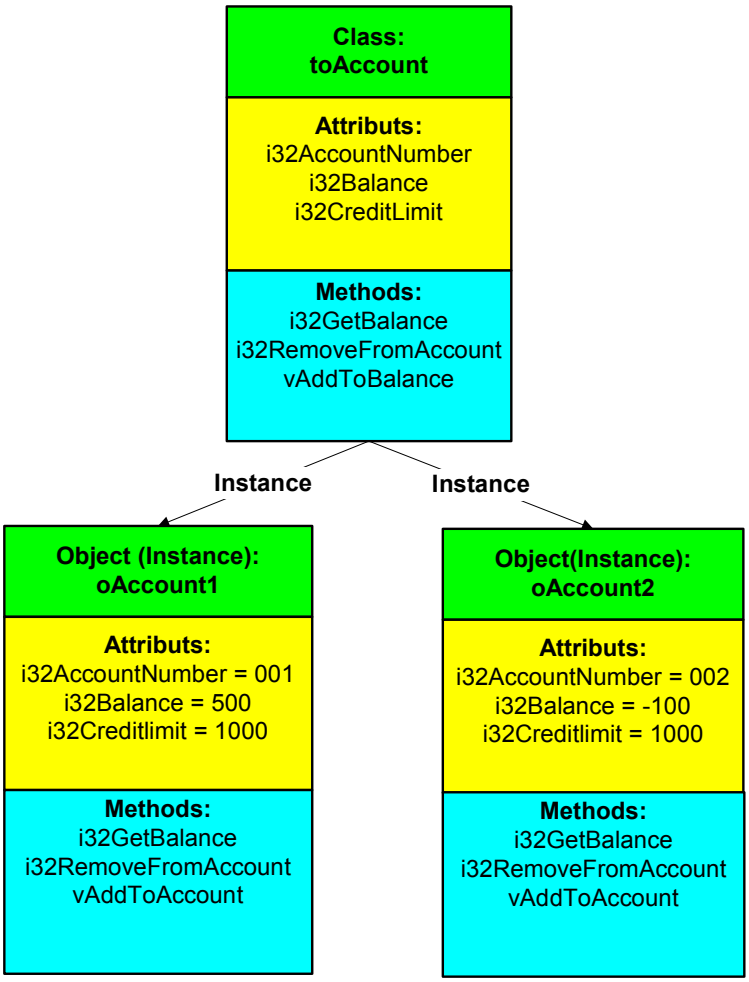

Figure 3: Class with instanced objects

Two objects of the account class are created in Fig. 3. They differ from one another through their different object names (oAccount1, oAccount2) and the different values of their attributes.

Classes always consist of a private area, in which the properties and methods are located, which are not visible from the outside, and a public area, which is visible to the outside. If OOP is implemented correctly, no properties (data) are usually located in the public area. This makes it possible to clearly specify and control the way data is accessed. This kind of access control is not possible with procedural programming, since each function has to freely access data, such as the data of a record.

The properties can only be used and influenced from the outside via public methods. Public methods can access the public and private areas of the object thus providing an interface between the object and the outside world.

Because of the clear separation between data and interface it is possible to modify objects subsequently without the need for changes to the calling program. Only the interfaces have to be sustained.

Our account class for instance could receive a password for online access as an additional element. To make this possible, it would be necessary to create a number of methods to work with the new password. Since ATM's continue to only work with the normal Password, no change will be necessary to the ATM class. The interface to the ATM object has not changed as you know.

Comment:

Up until now we did not use any password in our class. The reason for this is that we will not create a checking account class using our account class until we have reached section 3.2.5 below on inheritance, which has a password.

As you can see, the extension of the class does not have any influence on the present interfaces and can easily be done.

A practical example for the use of objects

In order to deepen the knowledge of classes and objects, we will now use an example that occurs in everyday programming practice. Imagine you wanted to implement a graphical user interface that accepts user entry through a button and a text field and then carries out a calculation. The result of the calculation should then be displayed in a text field. If you were to program the button yourself, you would have a number of tasks:

- Set the individual pixels to draw the button
- Change the size and color during runtime
- Move the button with the window
- Always keep the button in the foreground
- Calculate the mouse position and monitor mouse input
- Monitor the keyboard input for the button
- Reaction to the input
- Change the appearance, if the button is pressed
- Display the text on the button
- etc.

You would also have to implement the window and the input and output of text yourself. This would be a little bit too much effort just to do a simple calculation, wouldn't it?

You can also find the Windows elements that are needed to do this in the provided classes of the MFC (Microsoft Foundation Classes).

The MFC is a class library by Microsoft, which provides graphical objects (such as buttons) and Windows functionalities in the form of classes. The complex functionality of the individual objects is encapsulated in these classes. All you get is an interface that helps you manipulate and query the object. For the button this means that many of the simple tasks, such as moving a button with the containing window, changing the appearance, calling a function when the mouse button or keyboard is pressed, and of

course the representation on the screen, can be done by the button object without the need for you to explicitly do something. Thus, this functionality is encapsulated.

Functions and public attributes are available to you as interfaces, and you can use them to change the properties of the button during runtime.

The object oriented approach is not fully implemented in this case, since it is possible to directly access some properties (public attributes) of the object. Anyway you should not allow direct access to properties in your own projects.

In order to react to user input the button itself calls a number of functions (e.g., button pressed), and in these functions you have the opportunity to call your own functions to react to these events. In contrast to "normal" programs that are only used to process the code procedural, we are dealing with an event controlled program.

As you can see from this practical example, classes and objects are used to make programming easier. All the user has to know is the interfaces that he needs. Usually he/she does not have to be concerned with the fundamental operating principles of the objects. Besides, inadmissible object manipulations and operations that don't make sense are prevented.

3.2.3 Properties of objects

The property of an object is the data that makes up the object. For graphical objects this could be for instance the size of the object, its position, the color, and the way it is displayed.

In our account example it is the account number, the account balance and the credit limit.

If OOP is consequently implemented, all properties are located in the non-public area of the object. The only possible access to the properties will be through the object's inherent methods.

3.2.4 Methods of objects

As you could see in the last sections, we made a distinction between private methods, which can only be called from within the object, and public methods, which can also be accessed from the outside.

Next to this general classification that is based on the visibility there are also other criteria for the differentiation of methods:

Functions.

A return value such as the value of an element is expected when calling functions. A good example for a function is the method of the account class i32GetBalance. Objects of the ATM class can request the checking account balance of a user by using this function.

Procedures

No return value is expected when procedures are called. An example for this is the vAddToBalance method; an amount is passed without expecting a return value.

Constructors

A constructor is a method that has to be called when an object is created. The main task of the constructor is to initialize variables and allocate memory that is administered dynamically through pointers. Memory that was allocated using this method has to be deallocated again using a destructor (see the next section)!

In procedural programs the tasks of a constructor are usually realized via an initialization function.

Destructors

Destructors have to be called to destroy an object. They have the task of deallocating memory that is no longer needed.

Virtual methods

In section 3.2.6 on polymorphism you will get to know the virtual methods in addition to the methods that were presented here.

3.2.5 Inheritance

One of the particularly important features of object oriented programming languages is inheritance. It makes it significantly easier to reuse program code.

Relationships such as "a checking account is an account" are realized via inheritance.

Through inheritance of the parent class account the checking account class can be created in our example, without having to code a completely new class. The derived checking account class has all the properties and methods of the **parent class** account and the only things that might be added are additional specific properties and methods. The number of derived classes, in other words the classes that are created through inheritance is not limited. If you want to use inheritance in practice, then the parent class should only include basic functions and properties.

A class that was created through inheritance can also be derived again. This allows to create multilayered hierarchies such as a tiger is a predatory animal and a predatory animal is an animal. In this case the animal class is the parent class.

In the following example we see the savings account class and the checking account class, which were derived from the account class:

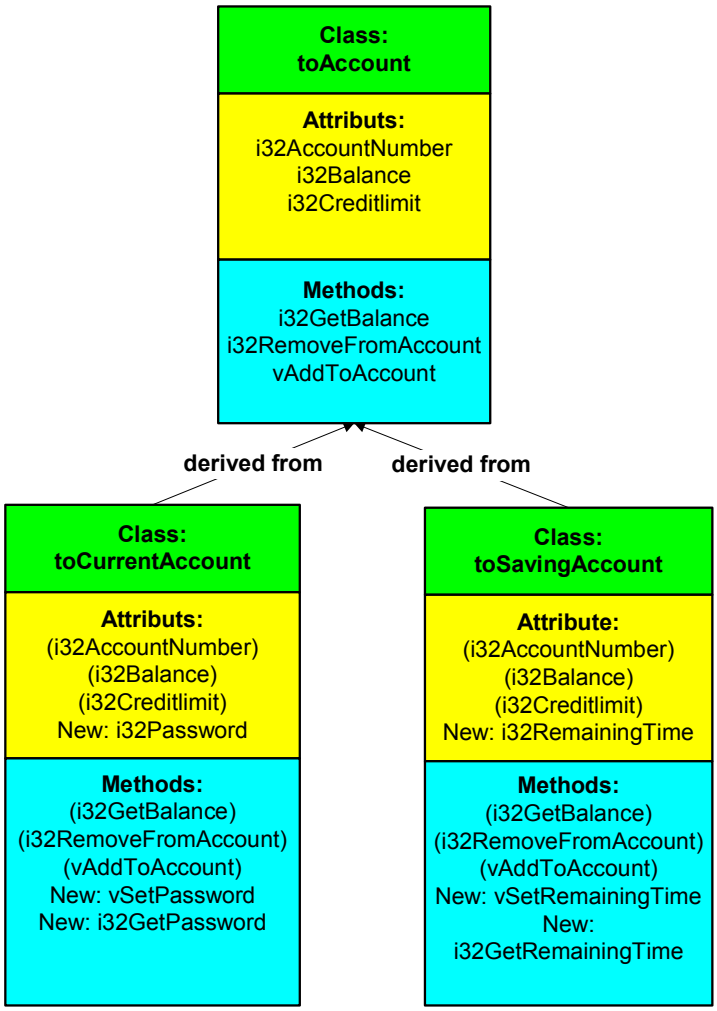

Figure 4: Hierarchy of the account classes

As you can see in Fig. 4, the password attribute was added to the derived checking account class in order to gain access to the account via the ATM. Besides, the i32GetPassword function and the vSetPassword procedure were added to work with the password. The properties and methods in brackets were adopted through inheritance and are not implemented again in the derived classes.

The savings account extends the parent class by adding the remaining time attribute and the appropriate methods. The remaining time indicates how long the account will be closed for withdrawals. This also means a change in the i32RemoveFromAccount method. In this process it has to be checked whether a withdrawal is permissible in general. If it is impermissible, a zero is returned as the withdrawn amount. In the context of inheritance, the i32RemoveFromAccount method of the parent class is overridden by the modified i32RemoveFromAccount method of the derived class. This modification does not have any effect on the parent class!

Methods can also be overridden in POOL, in contrast to C++, if the number and/or type of transfer parameters change. This is due to the fact that there is no overloading mechanism of methods in POOL, and thus the name of the method is distinctive.

The virtual methods constitute an exception in regard to overriding, since they must not be overwritten with modified parameters. We will take a closer look at these methods in the next chapter.

The special access features to private elements of parent objects will be explained in further detail in section 4.5 on the realization of inheritance in POOL.

3.2.6 Polymorphism

Polymorphism is one of the most important fundamentals of object oriented languages. The word means multiple shapes. It does not yet have to be specified during compilation, whether the methods of a parent class or the methods of a derived class should be called. Please remember that a derived class can override a method of the parent class, without changing the method of the parent class.

The decision which method is called is made during the runtime of the program. This is also called late or dynamic binding in contrast to static binding, during which the method that is going to be used is already specified during compilation.

Let us assume that the account of all account holders had to be administered in a linked list. If only static binding existed you would have to change the list each time you add a new account type.

In order to avoid this, you simply create a list that includes a pointer of the parent class account type. An object of the derived class like the checking account can be pointed to by a pointer of a parent class type (since a checking account is an account).

If a method of the object is called, dynamic binding assures that the method of the derived class is called instead of the parent class method. In order to use dynamic binding instead of the static binding, the method has to be declared as virtual. Virtual methods always have to contain the same prototypes during overriding, in other words the same parameter list, as the methods of the parent class that is to be overidden.

Imagine you wanted to withdraw an annual bank charge from all accounts. If this charge is the same for all accounts, you can simply call the i32RemoveFromAccount method for each list element (each object), without having to think about, whether the account is a checking account or a savings account.

The i32RemoveFromAccount method can be implemented in different ways. It would be possible for instance that, when the function for the savings account is called, the bank account is not actually charged with the amount. Instead, it is either charged to your checking account or an invoice is sent to you.

However, with the checking account it is always permitted to charge the account, so that the method of the parent class can be used.

The benefit of virtual functions is that the user of the list does not have to deal with the different mechanisms that are used to charge an account. He/she will always call the same function.

In section 4.6.1 we will go into further detail on how this is implemented in POOL.

4 Implementation of OOP in POOL

Since you have become familiar with the principle of OOP, we can now deal with the implementation in POOL.

4.1 Encapsulation and creation of libraries

As already discussed earlier, all properties that are not supposed to be called or viewed from the outside are declared in the private area of the class for the purpose of encapsulation. Methods that are used as an interface for other classes and modules, however, have to be declared in the public area. The key word `private` is used for the declaration of private properties and methods. Public properties and methods are declared by using the key word `public`. Please remember that all properties have to be declared `private` to achieve a consequent implementation of OOP. If an access to the object from the outside should be made possible, then the appropriate public methods have to be provided. All methods of the class have full access to the public and private properties and methods of their own class.

Although the key words `private` and `public` can be nested in the object in any order, you should declare the `private` area first and then the `public` area for a better overview. Since an object can be used in the methods of its own class, the declaration of the properties always has to be done before the declaration of the methods. The following example shows how to separate the public and private areas:

Example (TestLib1.pli)

```
type
  TestLib_toAccount = object                          {Declaration of the class
                                                       account}
    private                                           {Private properties of the
                                                       class}
    i32AccountNumber: Int32;
    i32Balance:       Int32;
    i32CreditLimit:   Int32;

    public                                            {Public methods of the
                                                       class}
    function  i32GetBalance:Int32;                    {Function to calculate
                                                       the account balance}
    procedure vAddToAccount(i32ValuePar:Int32);       {Procedure to increase the
                                                       account balance}
    {Other methods.....................................................}
  end;
```

As you can see, the body of the object only includes the declaration of the methods. The actual method implementation is done in an include file .

It would not be absolutely necessary to do this, but it will improve clarity in larger projects and is popular in particular in the Java programming environment. Include files or library files receive a .pli ending. The following figure shows the structure of a program, in which libraries are used:

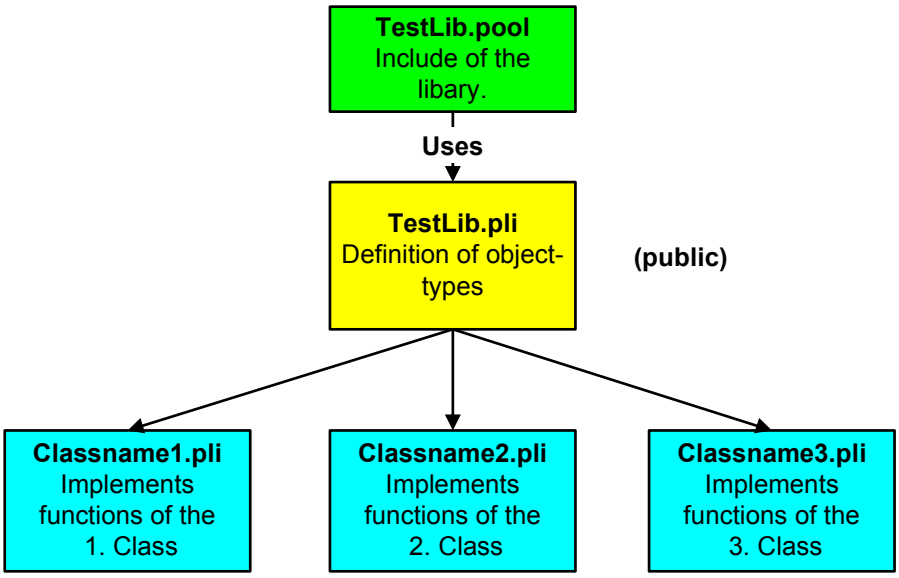

Figure 5: Structure and use of an object library

Start out creating three files. Call the first file TestLib.pool, the second TestLib.pli and the third TestLib_toAccount.pli. In order to find the examples and solutions on the disk, the file names are numbered using a consecutive index. To avoid confusion we will not use the numbering in this text, though.

You can copy the above source code into the TestLib.pli file. Copy the definition of the object methods into TestLib_toAccount.pli.

Example (TestLib1_toAccount.pli - Methods of the account object)

```
{Function to retrieve the account balance}
function TestLib_toAccount.i32GetBalance:Int32;
begin
  i32GetBalance := i32Balance;
end;

{Procedure to increase the account balance}
procedure TestLib_toAccount.vAddToAccount(i32ValuePar:Int32);
begin
  i32Balance := i32Balance + i32ValuePar;
end;

{Other methods.....................................................}
```

If you add other classes to the TestLib.pli file, you should create a separate include file (.pli) for each class that implements the class methods. This will significantly improve the clarity. Please use the appropriate class name as file name.

Now we get to the TestLib.pool file. The include files are added to this file and are encapsulated in regard to their scope.

Example (TestLib1.pool - Interface declaration of the library)

```
module TESTLIB;

public                  {Public methods}
{$i TestLib1}           {Integrating the file with the classes –
                         TestLib.pli. All elements of the classes, which are
                         declared public, are as public elements of the
                         library and thus accessible from the outside.
                         Usually only class declarations and public functions
                         of the classes are declared as public.}

private                 {Private implementation of the methods. The
                         implementation is done in the private area, so that
                         an access to these elements in not possible from the
                         outside.}
{$i TestLib_toAccount1}{Integration of the file containing the implementation
                         of the functions of the account class –
                         TestLib_toAccount.pli}

{Add other classes at this point}

begin
end.
```

The syntax for the integration is {$i NameOfThePliFile}, in other words in this case the curled brackes do not indicate that we are dealing with a comment!

The TestLib.pli file, in which the classes are defined, is located in the public area of the TestLib.pool module. Since the methods within the file are declared public, they can be accessed from other objects and modules.

The TestLib_toAccount file, in which the methods are implemented, is placed into the private area of the module, and thus it is invisible to other modules, which import the compiled TestLib module. This separation between public definition and private implementation of methods has to be generally used in POOL.

After TestLib.pool is compiled, you will get a TestLib.pi file, which can be imported in other projects and which provides the classes of the library. The user can use them without gaining any insight into the source code. Thus confidentiality is maintained.

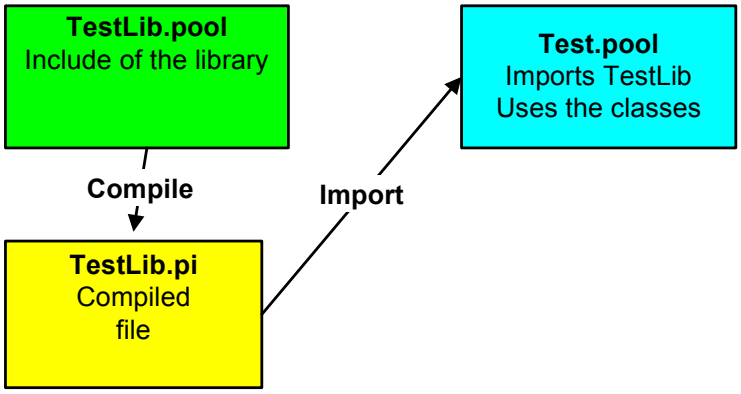

Figure 6: Using classes in external projects

Next, create the TestLib.pool file and compile the project.

Note:

Since we are adding the include files, they are automatically compiled as well. The result will be the executable TestLib.pi program.

Afterwards, you can import the classes in a test file and use their public elements. In the next section you will learn how to do this.

4.2 Classes, objects, and library import

In the last section you already read about the implementation of encapsulation in a class. Now we want to take a closer look at the use of classes.

As you know, a class consists of its name, its properties and its methods. In our case the name is composed of the name of the library, followed by an underscore and the actual name of the object type including the prefix 'to'.

As already discussed in the previous section, the declaration of the properties has to be done before the declaration of the methods.

Once again the class declaration:

Example

```
type
  TestLib_tpoAccount = ^TestLib_toAccount; {Declaration of a pointer to the
                                            object}

  TestLib_toAccount  = object              {Declaration of the object (of the
                                            account class}

   private
    {Private properties of the class - always in first place}
    {Private methods of the class}
```

```
  public
   {Public methods of the class}
 end;
```

As you can see, we have also defined a pointer to an object of the class. This is not always necessary, but it eases later on to work with the objects and thus it should generally be done since no memory is necessary for the declaration.

In order to use a class in a project, you first have to import the library . We will use our compiled TestLib.pi file as an example.

Example (Test1.pool)

```
{Test module for the account object - Test.pool file}
module TEST;

import TestLib; {Import the library (TestLib.pi)}
```

Next, you can declare an object of the TestLib_toAccount class. We will once again use the key word `var` to do this:

```
private

procedure vMain;
var
  oAccount1: TestLib_toAccount;       {Declaring an instance of the
                                       account class}
```

At this point you should not forget to plan the call of the constructor (see next section) for dynamic storage allocation and initialization of the variables, if the object has a constructor.

To reference the object, specify the object name followed by a period and the element name of the object. The elements that can be used are both public methods as well as public properties if they exist. You should memorize the syntax for the access to the elements of an object, because they are used this way in all current object oriented languages.

```
begin
  oAccount1.vAddToAccount(1000);     {Call of the procedure
                                      vAddToAccount of the object}

  Writeln(oAccount1.i32GetBalance);  {Output of the account balance}
end;
```

```
begin
end.
```

If another object is used within an object, then it is accessed using the following syntax:

Instance1.Instance2.MethodObject2;

Of course this can only work if Object2 is a public property of Object1.

Further nesting is obviously also possible. The reading direction is what is important: Object1 includes Object2 as an element and Object2 includes a method. In other words, a method of Object2 is called via Object1 with the help of this construct. At this point we also want to refer to the with statement, which was explained in the first part of the tutorial and which helps to reduce the amount of typing.

Exercise

- Create the Test1.pool file that is described above and test the function in the Aida commander. You can call the function i32GetBalance within a WriteIn statement in the vMain procedure to perform the test.
- Add the methods for setting and reading the account number of the class. Thoroughly test these methods. Please try also to access a property through the ObjectName.PropertyName syntax. What happens? Think about what this compiler message means.

Solution

Solution (file TestLib2.pli)

```
function  i32GetAccountNumber:Int32;            {Function to retrieve the
                                                  account number}
procedure vSetAccountNumber(i32ValuePar:Int32); {Procedure to set the
                                                  account number}
```

Solution (File TestLib2_toAccount.pli)

```
{Function to retrieve the account number }
function  TestLib_toAccount.i32GetAccountNumber:Int32;
begin
  i32GetAccountNumber := i32AccountNumber;
end;

{Procedure to set the account number }
procedure TestLib_toAccount.vSetAccountNumber(i32ValuePar:Int32);
begin
  i32AccountNumber := i32ValuePar;
end;
```

You can find the complete solution in the Test2.pool, TestLib2_toAccount.pli TestLib2.pli, and TestLib2.pool files.

4.3 Properties

All kinds of variables, pointers as well as records and other classes can be used as properties. If OOP is consequently used, all properties are located in the private area of the class, in other words they are declared `private`. If properties are declared in the public area, they can be accessed using the object name followed by a period.

Example

```
oAccount1.i32Balance := 100;  {Setting the public variable
                               i32Balance to 100}
```

4.4 Methods

In addition to the already known methods, this section will mainly focus on the constructor and destructor. Please remember that only those methods can be accessed from the outside an object, which are declared public. However, all public and private methods of an object can be accessed from within an object. The exact definition of the access was explained before in the chapters 4.2.1 and 5.1 on encapsulation. Please take your time and reread the chapters if necessary.

4.4.1 Functions

We had a closer look at functions in part one of the tutorial so we will limit this section to the syntax of function calls. Please refer to part one for further detail information.

Example

```
i32Help := i32GetBalance;           {Call of an object function in the actual
                                     object}


i32Help := oAccount1.i32GetBalance; {Call of a public object function from
                                     outside the object}
```

4.4.2 Procedures

The syntax for the internal and external procedure call differs from the function call in that the return value is missing.

Example

```
vAddToAccount(1000);                {Call of the procedure in the actual
                                     object}
oAccount1.vAddToAccount(1000);      {Call of the public procedure from
                                     outside the object}
```

Exercise

Add the methods for setting and reading the credit limit to the parent class.

Prototypes (in toAccount)

```
function  i32GetCreditLimit: Int32;             {Procedure to retrieve the
                                                 credit limit}
procedure vSetCreditLimit(i32CreditPar: Int32); {Procedure to set the
                                                 credit limit e.g., -1000}
```

Next, implement the missing withdrawal function. The function should check, with the help of the credit limit, the amount that can be withdrawn, charge it to the account, and return it as an `Int32` value. The credit limit is specified as a negative number. The return value is then used by the ATM to disburse the appropriate amount. Please test this function in your main program as well.

Prototype (in toAccount)

```
function i32RemoveFromAccount(i32ValuePar:Int32): Int32; {Function for
                                                          withdrawal}
```

Solution

Solution (TestLib3_toAccount.pli)

```
{Procedure to get the credit limit}
function TestLib_toAccount.i32GetCreditLimit: Int32;
begin
  i32GetCreditLimit := i32CreditLimit;
end;

{Procedure to set the credit limit}
```

```
procedure TestLib_toAccount.vSetCreditLimit(i32CreditPar:Int32);
begin
  i32CreditLimit := i32CreditPar;
end;

{Function for withdrawal}
function TestLib_toAccount.i32RemoveFromAccount(i32ValuePar:Int32): Int32;
var
  i32Help: Int32;                                          {Help variable}

begin
  {If the full amount can not be disbursed}
  if (i32Balance - i32ValuePar) < i32CreditLimit then
    i32Help := i32Balance - i32CreditLimit;                {Max. permissible
                                                            disbursement}
    Writeln("You can't withdraw more than ", i32Help," Euro.");
    i32Balance := i32Balance - i32Help;                    {Withdrawing the
                                                            amount}
    i32RemoveFromAccount := i32Help;

  else
    i32Balance := i32Balance - i32ValuePar;
    i32RemoveFromAccount := i32ValuePar;
  endif;
end;
```

We intentionally did not deal with the treatment of user errors. The sign for the withdrawal amount for example has to be checked also. Since the sign has to be positive, an `unsigned` value could be used as parameter.

4.4.3 Constructors

Important notice for users who switch from other programming languages:

In contrast to other object oriented languages such as C++ and Java, the constructor and the destructor are not automatically called during the declaration of an object. The user has to call them. Possible errors have to be prevented in the program. If you consider that a class can have several constructors and that dynamic instancing of objects can only be done within a program concept, then this is certainly the more general approach.

Constructors are special methods that are used to create new objects. As we already mentioned before, the constructor has to be called explicitly, in other words by the developer in order to create an object. Variables and pointers are initialized with a start value in the constructor, and dynamic memory is allocated for the data of the object if necessary.

Memory allocated using a constructor has to be deallocated using the destructor (see section 4.4.5). The deallocation has to occur no later than at the completion of the program. As with the constructor, an explicit call is required, usually in the vDone procedure.

Constructors are static methods, in other words they must not be declared as `virtual` (see also section 4.6.1), since the virtual methods table (VMT), which is responsible for the correct assignment of the methods, does not exist during the initialization of the object.

All classes in POOL are derived implicitly by the POOL toRoot base class. This base class includes the poInit base constructor. If a separate constructor is created, one of the inherited constructors has to be called as well (see also section 4.5.5 on inheritance). Usually this is the poInit base constructor, which is called using the key word `inherited`.

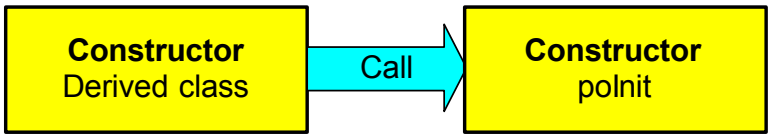

Figure 7: Call of the constructor poInit of the base class

The constructor returns a pointer to the new object. However, this return value can be ignored (use of a function as a procedure – see also part 1 of the tutorial).

In case of an error the nil pointer it returned. Usually, this happens only if there is not enough memory. The return value should most definitely be checked when calling the base class constructor. If the call of the constructor of the parent class fails, the destructor must not be called.

If you have comprehension difficulties at this point, please continue with the following example and reread this section later again.

We will take a closer look at the call of constructors using inheritance in section 4.5.5. In the following you can see the definition of the constructor from our account example.

Example (TestLib4_toAccount.pli)

```
constructor TestLib_toAccount.poInit;
begin
  if inherited poInit = nil then {Call of the inherited constructor of the
                                  base class}
    poInit := nil;               {return value = nil}
    return;                      {Termination, if the initialization
                                  has failed}
  endif;

  Writeln("Constructor is executed");
  {Initialization of the object properties}
  oSelf.i32Balance := 0;         {Assignment of the object property
                                  account balance}
  i32AccountNumber := 1;         {Assignment is equivalent to
                                  oSelf.i32AccountNumber := 1;}
  i32CreditLimit := 0;           {Assignment is equivalent to
                                  oSelf.i32CreditLimit := 0;}
end;
```

The parameter oSelf

Up until now we have always been assuming that an assignment to the variable i32AccountNumber of the object oAccount1 took place, when the procedure oAccount1.vSetAccountNumber(1) was called and the implementation contained the statement i32AccountNumber := 1.

How does the vSetAccountNumber procedure know which instance to modify?

The solution to this problem is the oSelf parameter, which is passed implicitly by the calling object to the method and which refers to the actually calling object. If properties or methods of the calling object are used within the method, then the call is always done implicitly via oSelf.

The i32Balance := 0 statement is thus converted by the compiler into oSelf.i32Balance := 0. It is also imaginable (and you can even write it if you like) that any access to class elements is located in a with statement (see part 1 of the tutorial) with the oSelf parameter.

Example

```
with oSelf do
  i32Balance := 0;
  i32CreditLimit := 0;
  {etc.}
endwith;
```

The explicit use of oSelf is also permitted as was shown using the constructor example.

The oSelf parameter is usually needed whenever a pointer or a reference to the actual object is to be passed to another object. Please make sure you remember that oSelf is always a pointer to the actual object (o) itself (Self).

You will get to know another use of oSelf in section 4.4.6 on copy constructors.

Calling the constructor

The call of the constructor in the application program is placed after the declaration of the object. Memory can be reserved for all dynamic properties. Please compare this to the procedure of using dynamic pointers. Here too, the memory has to be initially allocated using New. No memory is reserved with the declaration, with the exemption of the static properties.

If you do not remember this relationship too well, we recommend that you repeat section 5.1.7 on dynamic memory management in the first part of the tutorial.

The success of the constructor call has to be checked every time. To do this, the validity of the object's address is checked using an if statement. If the address is not valid, a message is put out and the program is aborted with a return statement. Please remember that the constructor returns a pointer to the object, if the call was successful or `nil` in case of an error.

Example (Test4.pool)

```
if oAccount1.poInit = nil then      {Call of the constructor in vMain}
                                    {If the creation fails}
  Writeln(GetErrorMsg(GetError));   {An error message is put out}
  return;                           {Important: If return is called in
                                     vMain, this will lead to an abortion of
                                     the program.
                                     In practice an appropriate error
                                     handling is necessary depending
                                     on the application}
endif;
```

It is not necessary to implement a constructor for an object, which only consists of pre-allocated elements, in other words constants and not dynamically allocated variables. You already had the opportunity to take a closer look at the possibility of allocating memory and assigning a value to an object in section 4.2. There, a value was explicitly assigned to each variable of the object. It is also necessary to call the destructor for objects that do not have dynamically allocated memory.

Important

Memory that was allocated using a constructor always has to be deallocated using a destructor. The attempt of calling the constructor of a non-deallocated object again will inevitably lead to a runtime error. Please try this by calling oAccount1.poInit in the above example a second time. In order to avoid errors from occurring during this process, the memory would have to be deallocated using the destructor before the constructor is called the second time. This matter will be explained again in further detail in the destructor description.

4.4.4 Constructor with parameters

The initialization can be done with the desired values by passing parameters to the constructor. This works the same way as passing parameters to normal functions or procedures, like you know from the first part of the tutorial.

Example (see the next exercise also)

```
constructor
TestLib_toAccount.poInit(i32AccountNumberPar:Int32;i32BalancePar:Int32;
                         i32CreditLimitPar:Int32);

begin
  if inherited poInit = nil then {Call to the constructor of the base class
                                  failed?}
    poInit := nil; {Return value}
    return;                      {Termination, in case the initialisation
                                  failed}
  endif;
```

```
  Writeln("Constructor is executed");

  {Initialization of the object properties with the parameters}
  i32Balance := i32BalancePar;
  i32AccountNumber := i32AccountNumberPar;
  i32CreditLimit := i32CreditLimitPar;
end;
```

The sole difference between a constructor call with parameters and a constructor call without parameters is the specification of the initialization values.

Example

```
{Call of the constructor for the second object}
if oAccount1.poInit(2,1000,-5000) = nil then
  Writeln(GetErrorMsg(GetError));  {An error message is put out}
  return;                          {Important: If return is called in
                                    vMain, this will lead to an abortion of
                                    the program.
                                    In practice appropriate error
                                    handling is necessary depending on the
                                    application}
endif;
```

4.4.5 Destructors

A destructor is the counterpart to a constructor. A destructor is used to deallocate memory that is no longer needed. The destructor has to be explicitly called in POOL. It is either called when an object is no longer needed or before the program ends.

All objects have to be deleted explicitly by calling their destructors at the end of the program!

As with the constructor, there is at least the destructor of the toRoot base class. In contrast to the constructor, the vDone base class destructor is virtual, though. All reserved memory has to be freed in the destructor by using `Dispose`.

Example

```
Dispose(pHelp); {Deallocation of dynamic allocated memory}
```

We will take a closer look at the general structure of destructors using the account class example.

Example

```
destructor TestLib_toAccount.vDone;
begin
  Dispose (pTest);  {First deallocate dynamically allocated memory}
  inherited vDone;  {Second call the destructor of the base class}
end;
```

The destructor of the derived class must call the destructor of the parent class. This is done using the key word `inherited`. In general, the destructor of the derived class gets the same name (vDone) as the destructor of the toRoot base class.

In order to ensure the use of the correct destructor with inheritance, the destructor always has to be declared virtual. A non-virtual or static declaration is possible in principle, however, it generally does not make sense (since vDone is declared as virtual in toRoot, of course it would not be possible to call it vDone in this case).

Since the standard destructor is the first virtual method of all objects, vDone can also be called with non-initialized objects without causing any error. Thus, it becomes easier to free memory that is no longer needed (to be sure, the destructor is simply called for all objects at the end of the program). Statically created objects (see section 4.4.3 above on constructors) have to be included into the destructor calls as well.

The destructor calls are shown in the following example.

Example

```
oAccount1.vDone;  {Destructor call for the first object}
oAccount2.vDone;  {Destructor call for the second object}

oAccount2.vDone;  {Permissible, although the data was already deallocated}
```

Please notice that destructors do not have a return value.

Exercise

Add a destructor and a constructor with parameters to the account class. To do this, you can use the source code of the previous two sections. The declaration of the methods is once again done in the public area of the TestLib.pli file. The implementation of the method is done in the TestLib_toAccount.pli file.

Create two objects of the account class in the Test.pool file and call the appropriate constructor. Assign values to the objects using methods and then print them out in the commander. Deallocate the dynamic properties of the objects again before the program ends by calling the destructor.

Experiment also with multiple calls of the constructor and destructor.

Solution

Solution (Test5_2.pool)

```
{Test program}
module TEST;

import TestLib;

private

procedure vMain;
var
  {Declaration of the objects}
  oAccount1: TestLib_toAccount;
  oAccount2: TestLib_toAccount;
begin
  {Call of the constructor for the first object}
  if oAccount1.poInit(1,0,-1000) = nil then {In case of error}
    Writeln(GetErrorMsg(GetError));  {an error message is put out}
    return;                          {Important: If return is called in
                                      vMain, this will lead to an abortion of
                                       the program.
                                      In practice an appropriate error
                                       processing
                                      will become necessary depending
                                      on the application}

  endif;

  {Call of the constructor for the second object}
  if oAccount1.poInit(2,0,-5000) = nil then {In case of error}
    Writeln(GetErrorMsg(GetError));  {An error message is put out}
    return;                          {Important: If return is called in
                                      vMain, this will lead to an abortion of
                                       the program.
                                      In practice an appropriate error
                                       processing
                                      will become necessary depending
                                      on the application}
  endif;

  oAccount1.vAddToAccount(5000);  {Set the values for Account1}
  oAccount2.vAddToAccount(-1000); {Set the values for Account2}

  {Output of the object properties}
  Writeln("AccountNumber1: ",oAccount1.i32GetAccountNumber);
  Writeln("Balance1 : ",oAccount1.i32GetBalance);
  Writeln("CreditLimit1: ",oAccount1.i32GetCreditLimit);
  Writeln("AccountNumber2: ",oAccount2.i32GetAccountNumber);
  Writeln("Balance2 : ",oAccount2.i32GetBalance);
  Writeln("CreditLimit2: ",oAccount2.i32GetCreditLimit);

  {Call of the destructors}
  oAccount1.vDone;
  oAccount2.vDone;
end;
```

```
begin
end.
```

4.4.6 Copy Constructor

When copying an object we distinguish between a shallow copy and a deep copy.

Shallow Copy

The assignment of oAccount1 to oAccount2 is called a shallow copy. With a shallow copy all properties of oAccount1 are assigned to the corresponding properties of oAccount2. You can use assignments to initialize an object instead of calling the constructor on that object.

Example (for a shallow copy)

```
oAccount2 := oAccount1;  {Assignment of oAccount1 to pAccount2}
```

Using shallow copies is inadequate when the objects contain pointers. As explained before all properties are simply copied, thus the pointers in both objects point to the same memory. As a result, changing the property value in one object changes the value in the other object too. This is usually not the desired behavior.

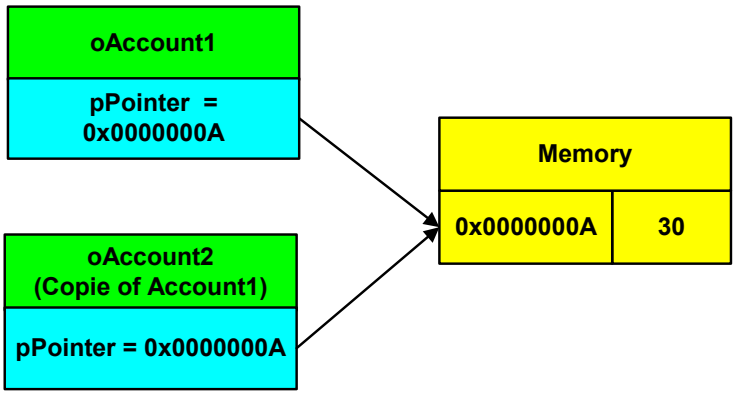

Figure 8: Shallow copy of an object

Another serious problem arises when the memory is freed. Assuming that the destructor of oAccount1 is called first and thus frees the memory. When the destructor of oAccount2 is called afterwards the memory has been freed already. This will inevitably lead to a runtime error.

Exercise

Please add a pointer to the account class, create an oAccount1 object and copy it into a second oAccount2 object. Add the allocation (New) and deallocation (Dispose) of the memory for the pointer to the constructor and the destructor. Test your program concerning this runtime error.

Write a method that can be used to manipulate the value through the pointer and one for the output.

Solution

Solution (TestLib6.pli - Extension of the properties)

```
private
pi32Test: ^Int32;
```

Solution (TestLib6_toAccount.pli - Extension of the constructor)

```
New(pi32Test);
```

Solution (TestLib6_toAccount.pli - Extension of the destructor)

```
Dispose(pi32Test);
```

Solution (TestLib6_toAccount.pli - New functions)

```
{Function to retrieve the value}
function  TestLib_toAccount.i32GetPointerValue:Int32;
begin
  i32GetPointerValue := pi32Test^;
end;

{Procedure to set the value}
procedure TestLib_toAccount.vSetPointerValue(i32ValuePar:Int32);
begin
  pi32Test^ := i32ValuePar;
end;
```

Solution (Test6.pool)

```
procedure vMain;
var
  i32LoopCounter: Int32;
  oAccount1:        TestLib_toAccount;
  oAccount2:        TestLib_toAccount;

begin
  {Calling the constructor for the first object}
  if oAccount1.poInit(1,500,-1000) = nil then {In case of an error}
    Writeln(GetErrorMsg(GetError)); {An error message is put out}
    return;                         {Important: If return is called in
                                     vMain, this will lead to an abortion of
                                     the program.
                                     In practice appropriate error
                                     handling is necessary
                                     depending on the application}
  endif;

  oAccount1.vSetPointerValue(10);   {Value = 10}
  oAccount2 := oAccount1;           {Shallow copy of oAccount1}
  Writeln("Value of account 1 is ",oAccount1.i32GetPointerValue);
  oAccount2.vSetPointerValue(20);   {Manipulation through the shallow copy}
  Writeln("Value of account 1 after modification through account 2 is ",
          oAccount1.i32GetPointerValue);
  oAccount1.vDone;                  {Deallocates the allocate memory the
                                     pointer points to}
  oAccount2.vDone;                  {Leads to an error, since the memory the
                                     pointer points to has already been
                                     deallocated}
end;
```

Warning!

The shallow copy is listed here for the sole purpose of describing a frequently made serious error and its implications. In practice you should not use this method to copy objects. In the next section we will introduce a better way to create a copy of an object.

Deep Copy

With the deep copy a new memory is explicitly allocated, assigned to the pointer, and the values of the memory are copied to the new memory.

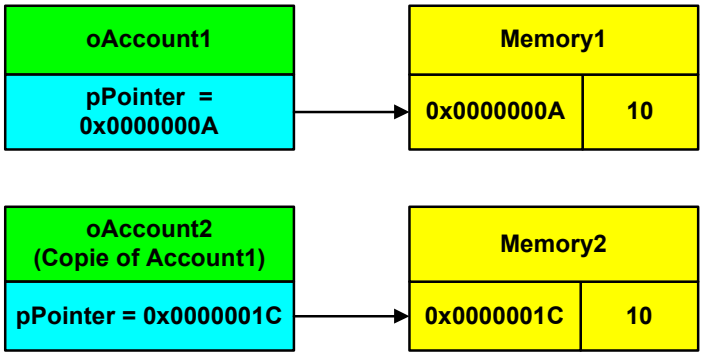

Figure 9: Deep copy of an object

In order to create a deep copy, a copy constructor has to be added to our account class (the standard constructor is preserved in this case, though). A reference to the object that is to be copied is passed to the copy constructor.

To do this you can use a feature of encapsulation that has not been mentioned yet. Access control is provided at class level instead of object level, that is two different objects of the same class can access each others private elements.

This seems to contradict the OOP principal of encapsulation though it does not. The developer of a class is still in control whether or not the objects can access private elements of other objects of the same class. The user of the class is still limited to its public elements.

By the way: This type of access can also be found in C++ and Java.

This allows the developer of a class to create a copy constructor that performs a deep copy.

The following example shows what a copy constructor looks like.

A reference to the object that is to be copied is passed to the copy constructor so that the data can be copied.

Example (TestLib7_toAccount.pli)

```
constructor TestLib_toAccount.poCopy(oAccount:TestLib_toAccount);
begin
  Writeln("Copy constructor is executed");
```

The first if statement is used to check, whether an object is assigned to itself.

Call: oAccount1.poCopy(oAccount1);

If this applies, a warning is put out and the copy obtains a reference to itself. Thus, oAccount1 continues to be oAccount1.

```
if @oSelf = @oAccount then            {If self-assignment}
  Writeln("Warning: self-assignment");
  poCopy := @oAccount;                {Reference to itself}
  return;                             {Back to the main program }
```

If no self-assignment occurred, a possibly already initialized object is first de-initialized using the destructor call.

```
else                                  {No self-allocation}
  oSelf.vDone;                        {De-initialization of a possibly
                                       already initialized object}
```

This is necessary, since the call of a constructor on an already initialized object leads to an error. In contrast to other methods it is allowed to call the destructor on non-initialized objects. More on destructors in the next section. This prevents errors in using destructors.

Alternatively you could check whether the object has been initialized before. In this case, you could notify the user and abort the copying process. Which method to use depends on the particular needs of the application.

Please notice that the object, to which the data that is to be copied is assigned, will lose any previous data.

The **data** from the object that is to be copied is passed to the standard constructor of the new object in the following code lines. At this point we use the option to access private elements of another object of the same class, in other words the concept of encapsulation on class level.

```
  {Call of the standard constructor of the new object}
  if oSelf.poInit(oAccount.i32AccountNumber,oAccount.i32Balance,
                  oAccount.i32CreditLimit) = nil then
                                      {Prevention of errors}
    Writeln("ERROR");
    poCopy := nil;                    {The result is the nil-pointer}
    return;                           {Exit the copy constructor}
  endif;
```

Possible errors are checked within the constructor.

After the object has been initialized, the content of the dynamically allocated memory is copied.

```
  pi32Test^  := oAccount.pi32Test^;          {Assignment of a value}
```

```
  endif;
end;                                        {End of the copy constructor}
```

Exercise

Write a program that creates a new account based on the data of an old account using the copy constructor. Test the deep copy by assigning a value to the pointer of the new object and verify that the value of the old object did not change (in contrast to a shallow copy).

Test the copying process with a previous initialization of the new object. Besides, experiment with self-assignment.

Solution

Solution (Test7.pool)

```
procedure vMain;
var
  {Declaration of the objects}
  oAccount1: TestLib_toAccount;
  oAccount2: TestLib_toAccount;

begin
  {Call of the standard constructor for the first object}
  if oAccount1.poInit(1,500,-1000) = nil then {In case of error}
    Writeln(GetErrorMsg(GetError));{Output of an error message}
    return;                       {Important: If return is called in
                                   vMain, this will lead to an abortion of the
                                    program.
                                   In practice an appropriate error handling
                                   is necessary depending
                                   on the application}
  endif;

  oAccount1.vSetPointerValue(10);{Value = 10}

  {Actually unnecessary initialization of the second object}
  if oAccount2.poInit(2,600,-2000) = nil then
    Writeln(GetErrorMsg(GetError));{Output of an error message}
    return;                       {Important: If return is called in
                                   vMain, this will lead to an abortion of the
                                    program.
                                   In practice an appropriate error handling
                                   is necessary depending
                                   on the application}
  endif;


  oAccount2.poCopy(oAccount1);  {Call of the copy constructor}
  Writeln("Value to which account 1 points: ",oAccount1.i32GetPointerValue);
  oAccount2.vSetPointerValue(20); {Assignment of a value pointer Account2}
```

```
  Writeln("Value to which account 1 points: ",oAccount1.i32GetPointerValue);
  Writeln("Value to which account 2 points: ",oAccount2.i32GetPointerValue);

  {Test of the copy}
  Writeln("AccountNumber Account 2: ",oAccount2.i32GetAccountNumber);
  Writeln("Balance Account 2 : ",oAccount2.i32GetBalance);
  Writeln("CreditLimit Account 2: ",oAccount2.i32GetCreditLimit);

  {Call of the destructors}
  oAccount1.vDone;
  oAccount2.vDone;
end;

begin
end.
```

4.5 Inheritance

This section will explain the special features of inheritance in POOL. Differences to the two major object oriented languages Java and C++ will be pointed out for users switching from these programming languages. Beginners should definitely have understood the basic concept of inheritance that was described in section 3.2.5 before continuing with the following section.

4.5.1 Parent Class

If a class is derived from another class, then the parent class has to be specified as well:

Example

```
{Definition of the checking account class (parent class account)}
type
  TestLib_toCheckingAccount = object(TestLib_toAccount)
```

The new class can also serve as a parent class for other classes. If a completely new class is created, it automatically inherits from the POOL's base class toRoot:

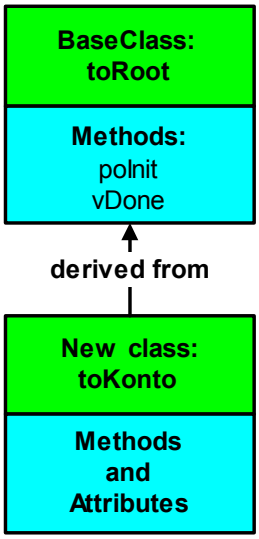

Figure 10: Deriving a class from the toRoot base class

4.5.2 Access rights

An important issue of inheritance is the access rights to private elements of the base class. If the object is located in the same library or in the same module, a direct access to private elements of the parent object is possible from the derived object. This makes it easier to implement new classes. Since you are responsible for the procedures within your own libraries, this point does not contradict the principle of encapsulation and since the source text of the library is usually not passed on, a user can not have this privilege. In C++ and Java there is a key word for the access to properties of a parent class from a derived class; the key word is `protected`.

If the inheriting object is located in another library or in another module, then the access to private elements of the parent class is not possible. This point is particularly important, because it allows to inherit from classes that were created by others, without compromising encapsulation. If it were possible to access the private elements of a parent class in this case as well, it would be easy to by-pass encapsulation simply by inheriting. The access to private elements is only possible through the provided functions.

This helps the creator of a library to specify exactly, how the elements of the parent class are accessed from the derived class.

This concept is also generally used in C++ and Java. The key word for all three languages is `private`.

Properties marked with the key word `public` can of course be accessed from all derived classes.

In order to memorize these important relationships, you should definitely do the next two exercises. They are intentionally kept simple, so that you can concentrate on the essential relationships.

Exercise 1

Derive the savings account class from the account class. The savings account class has an additional feature, the remaining time. The remaining time indicates for how much longer the account will be closed for withdrawals. Normally, you would rather use a release date in this case. However, we want to keep the example as simple as possible. Imagine the remaining time were decremented each day.

The declaration of the savings account class is done in the same file as the declaration of the class account.

Create the methods that are used to set and get the remaining time in a new file (TestLib_toSavingsAccount). Please don't forget that you have to import the file in TestLib.pool. Test the access to private properties of the parent class by accessing the account number using the i32GetPrivate function. Next, remove this access, which really doesn't make much sense.

You will learn about an appropriate use of the access to private elements of the parent class in the next section.

Solution 1

Solution (TestLib8.pli - Declaration of the derived SavingsAccount class)

```
type
  TestLib_tpoSavingsAccount = ^TestLib_toSavingsAccount;
  TestLib_toSavingsAccount  = object(TestLib_toAccount)

    private                                  {New elements of the class}
    i32RemainingTime: Int32;

    public                                   {New methods of the class}

    function i32GetRemainingTime:Int32;      {Procedure to retrieve the
                                              remaining time}
   {Procedure to set the remaining time}
    procedure vSetRemainingTime(i32RemainingTimePar:Int32);

    function i32GetPrivate:Int32;            {Access to private properties of
                                              the parent class}
  end;
```

Solution (TestLib8_toSavingsAccount- Definition of the functions)

```
function  TestLib_toSavingsAccount.i32GetRemainingTime:Int32;
begin
  i32GetRemainingTime := i32RemainingTime;
end;
```

```
procedure
TestLib_toSavingsAccount.vSetRemainingTime(i32RemainingTimePar:Int32);
begin
  i32RemainingTime := i32RemainingTimePar;
end;
```

Solution (TestLib8.pool)

```
private                              {Private implementation of the methods}

{$i TestLib_toAccount.pli}           {Import the file containing the functions
                                      of the account class}
{$i TestLib_toSavingsAccount.pli}   {Import the file containing the functions
                                      of the SavingsAccount account class}
```

Test of the access to private elements:

Solution (TestLib8_toSavingsAccount.pli - Definition of the function)

```
function  TestLib_toSavingsAccount.i32GetPrivate:Int32;
begin
  i32GetPrivate := i32AccountNumber; {Access to private properties of the
                                      parent class}
end;
```

Solution (Test8.pool - Call of the function)

```
Writeln(oSavingsAccount1.i32GetPrivate);
```

Exercise 2

Create a completely new library with the name TestLibB (you will need the TestLibB.pli, TestLibB_toCheckingAccount.pli and TestLibB.pool files). Derive a checking account class in the TestLibB.pli file from the account class and add the i32Password property. Define functions to set and read the password in the TestLibB_toCheckingAccount.pli file (i32GetPassword, vSetPassword(i32ValuePar:`Int32`)). Add the .pli file in TestLibB.pool. Import the resulting TestLibB.pi file into the Test8.pool file. Please don't forget to import the library TestLib.pi to provide the the implementation of the account class for inheritance.

Test the access to private elements of the parent class as described in exercise 1. Why is it not possible?

Solution 2

Solution (TestLibB.pli)

```
type
  {Declaration of the derived checking account class}
  TestLibB_tpoCheckingAccount  = ^TestLibB_toCheckingAccount;
  TestLibB_toCheckingAccount   = object(TestLib_toAccount)

    private                                {Elements of the class}
    i32Password: Int32;

    public                                 {Methods of the class}

    function  i32GetPassword:Int32;              {Procedure to retrieve
                                                  the password}
    procedure vSetPassword(i32ValuePar:Int32);   {Procedure to set the
                                                  password}
    function i32GetPrivate:Int32;                {Access to private
                                                  properties of the
                                                  parent class}
  end;
```

Solution (TestLibB_toCheckingAccount.pli)

```
function TestLibB_toCheckingAccount.i32GetPassword:Int32;
begin
  i32GetPassword := i32Password;
end;

procedure TestLibB_toCheckingAccount.vSetPassword(i32ValuePar:Int32);
begin
  i32Password := i32ValuePar;
end;

{Test the access to private elements of the parent class}
function TestLibB_toCheckingAccount.i32GetPrivate:Int32;
begin
  i32GetPrivate := i32AccountNumber;   {Access to private properties of the
                                        parent class}
end;
```

Solution (TestLibB.pool)

```
{Interface definition of the library}
```

```
module TESTLIBB;
import TestLib;                        {For inheritance}

public {Public methods}
{$i TestLibB.pli}                      {Import the file containing the classes}

private                                {Private implementation of the methods}
{$i TestLibB_toCheckingAccount.pli}{Import the file containing the
                                    functions of the
                                    checking account class}
begin
end.
```

Comment

Since you are the developer of the account classes, you would normally add all classes in the library TestLib.

However, as you can see, a user who uses your account class as parent class can not access the private elements of your class.

Exercise 3

Rewrite the i32GetPrivate function so that an access to the account number is possible again.

Solution 3

Solution (Definition of the function in TestLibB_toCheckingAccount.pli)

```
function TestLibB_toCheckingAccount.i32GetPrivate:Int32;
begin
  i32GetPrivate := i32GetAccountNumber; {Access to private properties of the
                                         parent class via
                                         method}
end;
```

As you can see, an access to the account number is only possible via the public method i32GetAccountNumber.

Exercise 4

In order to have to handle only one library in this tutorial, you should now add the checking account class to your TestLib.pli library. Remove the i32GetPrivate function again to do this. You can find the complete source code in the solution with index 10.

4.5.3 Overriding properties

It is not permissible to override properties of the parent class in a derived class.

4.5.4 Overriding methods

Up till now we have added new methods in the derived classes. Now we want to take a closer look at a case, in which an existing method is overridden.

With static methods it is possible to change the prototype, in other words the quantity and type of parameters. This is not possible with virtual methods. More on this topic is provided in section 4.6 on polymorphism.

We will once again use our i32RemoveFromAccount method as example.

A withdrawal should only be possible in the savings account class, if the waiting period has elapsed. To illustrate a possible extension of the parameter list, we will pass an additional boolean variable that allows premature withdrawal. In the section on polymorphism we will continue to improve this example.

Example (TestLib11_toSavingsAccount.pli)

```
{Overridden method of the parent class account}
function TestLib_toSavingsAccount.i32RemoveFromAccount(i32ValuePar:Int32;
                                                       boSpecial:Boolean):Int32;
var
  i32Help: Int32; {Help variable}

begin
  {If the waiting period has elapsed or a special debit entry
   exists.}
  if (i32RemainingTime <= 0) or boSpecial then
    if (i32Balance - i32ValuePar) < i32CreditLimit then
      i32Help                := i32Balance - i32CreditLimit;
      Writeln("Your can't withdraw more than ", i32Help," Euros.");
      i32Balance             := i32Balance - i32Help;
      i32RemoveFromAccount   := i32Help;
    else
      i32Balance             := i32Balance - i32ValuePar;
      i32RemoveFromAccount   := i32ValuePar;
    endif;
  else

    Writeln("Waiting period has not yet elapsed\nwithdrawal is possible in ",
             i32RemainingTime," days");
  endif
end;
```

Please don't forget to complete the method declaration in the savings account class.

Please try to declare the i32RemoveFromAccount method in the parent class as a virtual method (see section 4.6.1). What happens? Why?

In general you should try to work with virtual methods. The simple administration of all accounts in a linked list is no longer possible in the above example due to the use of static methods.

4.5.5 Overriding constructors

At this point we want to take a closer look at the relationships when overriding constructors. Please remember that constructors are always static. Thus, they can always be overridden with modified parameters. We will once again use our toAccount class as example. There we overrode the constructor poInit of the base class poRoot.

Example (see TestLib4)

```
TestLib_toAccount.poInit(i32AccountNumberPar:Int32;i32BalancePar:Int32;
                         i32CreditLimitPar);
```

In order to correctly initialize an object, the constructor of the parent class has to be called using the key word `inherited`.

Example

```
begin
  if inherited poInit = nil then {Call of the constructor of the base class
                                  failed?}
    poInit := nil;               {Return value}
    return;                      {Termination, if the initialization
                                  has failed}
  endif;
end;
```

If other classes are derived from the account class, they also have to include a constructor, which then calls the constructor poInit(.....) of the account class. This results in a chain of the constructor calls:

Figure 11: Chaining of the constructor calls

You can find an exercise on this topic in section 4.6 below on polymorphism.

Overriding destructors

As with constructors it is necessary to call the destructor of the parent class for destructors of derived classes. This call also occurs via the key word `inherited`.

Example (see TestLib4)

```
inherited vDone; {Call of the destructor of the parent class}
```

Here too chained calls will generally evolve:

Figure 12: Nesting of the destructor calls

Each of these destructors has the task of deallocating memory, which was allocated by the appropriate constructor of the class.

Since the destructors are declared as virtual methods it is guaranteed that the method of the class that was derived last is called first. This method will then call the destructor of the appropriate parent class (provided the call was implemented) and so on.

4.6 Polymorphism

To describe polymorphism we will once again use the example from the introduction in section 3.2.2. Please don't hesitate to reread this section to refresh your memory.

4.6.1 Virtual methods

As example for the declaration of a virtual or dynamic method we first want to look at the overridden method i32RemoveFromAccount of the savings account class. The method of the parent class has to be marked with the key word `virtual` to allow dynamic access to the method..

Example

```
{Virtual function of the parent class}
function i32RemoveFromAccount(i32ValuePar:Int32;boAdmin:Boolean):Int32;
virtual;
```

Since we are now working with virtual methods, the declaration of the overridden method has to match exactly the declaration in the parent class.

Overridden virtual methods and their derivations have to be declared as virtual again. A static method can not be overridden by a virtual method.

The constructor has to be called in order to initialize the VMT before virtual methods can be used.

Before we continue with the account example, we will first look at the access to virtual methods, which is not at all trivial, using a simple example.

A simple example:

Let us assume you wanted to model a band in classes. We create the musician class as parent class. This class has the methods greet and play.

The extensions to be implemented are the trumpeter and singer classes. To keep things simple, we will not use properties.

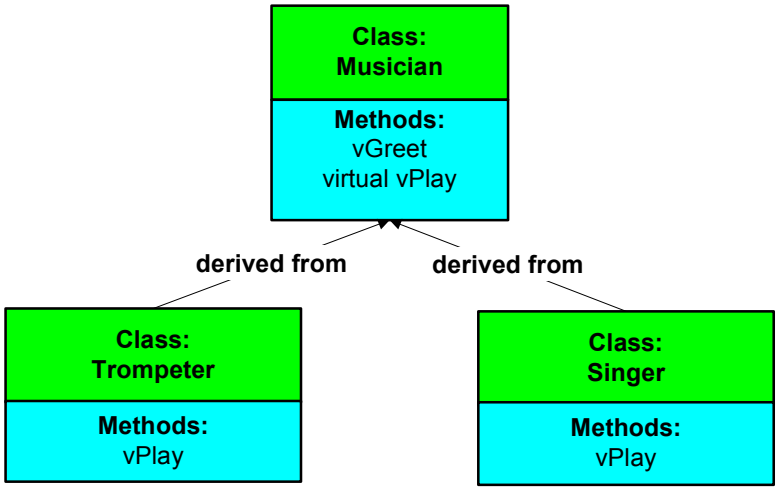

Figure 13: Band classes

All musicians should use the same greeting, but each one of them should play in his own way. All musicians should be administered in an array in order to play music. This setting of tasks is typical for the use of virtual functions.

To illustrate the difference between virtual and static methods, the method greet is overridden without using the key word `virtual` (we actually would not have to override it at all, since all should use the same greeting).

It would be best to create six files to test the methods (Musician.pool, LibMusician.pool, LibMusician_toMusician.pli, LibMusician_toSinger.pli, LibMusician_toTrumpeter.pli, and LibMusician.pli).

Please pay attention to the overridden method vPlay, which is declared virtual, in the following listing.

Example (LibMusician.pli)

```
{Declaration of the musician class}
type
  LibMusician_tpoMusician = ^LibMusician_toMusician;
  LibMusician_toMusician  = object
    public                          {Methods of the class}

    procedure vPlay; virtual;       {Virtual method vPlay}
    procedure vGreet;               {Static method vGreet}
  end;


{Declaration of the trumpeter class}
type
  LibMusician_tpoTrumpeter = ^LibMusician_toTrumpeter;
  LibMusician_toTrumpeter  = object(LibMusician_toMusician)

    public                          {Methods of the class}
    procedure vPlay; virtual;       {Virtual method vPlay}
    procedure vGreet;               {Static method vGreet}
  end;


{Declaration of the singer class}
type
  LibMusician_tpoSinger = ^LibMusician_toSinger;
  LibMusician_toSinger  = object(LibMusician_toMusician)

    public                          {Methods of the class}
    procedure vPlay; virtual;       {Virtual method vPlay}
    procedure vGreet;               {Static method vGreet}
  end;
```

Example (LibMusician_toMusician.pli)

```
procedure LibMusician_toMusician.vPlay;
begin
  Writeln("Musician plays music.");
end;

procedure LibMusician_toMusician.vGreet;
begin
  Writeln("Musician greets Hello")
end;
```

Example (LibMusician_toTrumpeter.pli)

```
procedure LibMusician_toTrumpeter.vPlay;
begin
  Writeln("Trumpeter plays.");
end;

procedure LibMusician_toTrumpeter.vGreet;
begin
  Writeln("Trumpeter greets Good Evening");
end;
```

Example (LibMusician_toSinger.pli)

```
procedure LibMusician_toSinger.vPlay;
begin
  Writeln("Singer sings.");
end;

procedure LibMusician_toSinger.vGreet;
begin
  Writeln("Singer greets Hi");
end;
```

Example (LibMusician.pool)

```
{Interface definition of the library}
module LIBMUSICIAN;
public {Public methods}

{$i LibMusician.pli}              {Import the file containing the classes}

private                           {Private implementation of the methods}

{$i LibMusician_toMusician.pli}   {Import the file containing the
                                   functions for the musician class}
{$i LibMusician_toSinger.pli}     {Import the file containing the
                                   functions for the singer class}
{$i LibMusician_toTrumpeter.pli}  {Import the file containing the
                                   functions for the trumpeter class}

{Add additional classes at this point}

begin
end.
```

Since the classes have been defined now, we can take a closer look at how they are used in the main program.

Example (TestMusician.pool)

```
module MUSICIAN;

import LibMusician;                    {Import the library}
```

After the library has been imported, the variables of type musician, trumpeter, and singer are declared. Besides, we declare pointers of the appropriate type.

```
private

procedure vMain;
var
  i32LoopCounter: Int32;              {Counter for the loop}

  {Declaration of the objects}
  oMusician1:    LibMusician_toMusician;
  oTrumpeter1:   LibMusician_toTrumpeter;
  oSinger1:      LibMusician_toSinger;

  {Pointer to the objects}
  poMusician1:   LibMusician_tpoMusician;
  poTrumpeter1:  LibMusician_tpoTrumpeter;
  poSinger1:     LibMusician_tpoSinger;
```

Next, an array of pointers to musician objects is declared, the constructors are called to initialize the objects, and the address is assigned to the individual elements of the array. Besides, the pointers to the objects are initialized. The pointers are of the type of the appropriate object and not like the pointers in the array of the type musician!

```
  {Creating an arrays of pointers to musicians}
  apoMusician: array[0..2] of ^LibMusician_toMusician;

begin
  {Call of the standard constructor for oMusician1}
  if oMusician1.poInit = nil then
    Writeln(GetErrorMsg(GetError));    {An error message is put out}
    return;                            {Important: If return is called in
                                        vMain, this will lead to an abortion
                                        of the program.
                                        In practice an appropriate error
                                        handling is necessary depending
                                        on the application}
  endif;

  {Call of the standard constructor for oTrumpeter1}
  if oTrumpeter1.poInit = nil then
    Writeln(GetErrorMsg(GetError));    {An error message is put out}
    return;                            {Important: If return is called in
                                        vMain, this will lead to an abortion
```

```
                                        of the program.
                                        In practice an appropriate error
                                        handling is necessary depending
                                        on the application}
  endif;

  {Call of the standard constructor for oSinger1}
  if oSinger1.poInit = nil then
    Writeln(GetErrorMsg(GetError));    {An error message is put out}
    return;                            {Important: If return is called in
                                        vMain, this will lead to an abortion
                                        of the program.
                                        In practice an appropriate error
                                        handling is necessary depending
                                        on the application}
  endif;

  {Assign the musician}
  poMusician1    := @oMusician1;
  apoMusician[0] := @oMusician1;

  {Assign the trumpeter}
  poTrumpeter1  := @oTrumpeter1;
  apoMusician[1] := @oTrumpeter1;

  {Assign the singer}
  poSinger1      := @oSinger1;
  apoMusician[2] := @oSinger1;
```

You will see the difference between the static and dynamic binding of methods in the following function call via an array.

Since play is a virtual method, the call is done dynamically, and the method of the derived class is called.

Greet on the other hand is a static method, and this is why the method of the parent class is called (since we are using a pointer to a musician, in other words the parent class type). In order to be able to access the greeting method of the derived classes via a pointer of the parent class type, the greeting method would have to be declared as a virtual as well.

```
  apoMusician[0]^.vGreet;          {Musician greets - Hello}
  apoMusician[0]^.vPlay;           {Musician plays music}
  apoMusician[1]^.vGreet;          {Musician greets - Hello}
  apoMusician[1]^.vPlay;           {Trumpeter plays}
  apoMusician[2]^.vGreet;          {Musician greets - Hello}
  apoMusician[2]^.vPlay;           {Singer sings}
```

In order to be able to fully use the advantage of virtual functions, you should replace the above lines with a `for` loop.

```
  for i32LoopCounter := 0 to 2 do
    apoMusician[i32LoopCounter]^.vGreet;
    apoMusician[i32LoopCounter]^.vPlay;
  endfor;
```

If the vGreet method of the trumpeter or the singer is called directly via the object, then the methods of the derived class are used. You are already familiar with this kind of static access from section 4.5.4 above on inheritance.

```
  {Static access}
  oMusician1.vGreet;                {Musician greets Hello}
  oTrumpeter1.vGreet;               {Trumpeter greets Good Evening}
  oSinger1.vGreet;                  {Singer greets Hi}
```

The same result can be achieved with the pointers of the corresponding object type.

```
  {Access via pointers of the type of the object}
  poMusician1^.vGreet;               {Musician greets Hello}
  poTrumpeter1^.vGreet;             {Trumpeter greets Good Evening}
  poSinger1^.vGreet;                {Singer greets Hi}
```

Under no circumstances should you forget to call the destructors. Since we did not define separate dynamic data types, it is sufficient to call the destructor of the parent class. To make sure that the correct destructor is called every time, all destructors have to be declared virtual.

```
  {Call of the destructors}
  for i32LoopCounter := 0 to 2 do
    apoMusician[i32LoopCounter]^.vDone;
  endfor;
end;

begin
end.
```

Finally, you should try once again to assign the address of a musician to the pointer of the type singer.

```
poSinger1      := @oMusician1;          {Assignment is not possible}
```

Why can you allocate a singer to a musician object, but not a musician to a singer object? The explanation is quite simple. Each singer is also a musician, in other words he/she can also be called a musician, since he/she has all the properties and methods of a musician. However, not every musician is a singer, in other words you can't call each musician a singer, since he/she does not necessarily have all the qualities of a singer. You can see how the object oriented view (paradigm) continues to copy real life. Take your time in letting this relationship sink in. In polymorphism we basically always work with the lowest common denominator of all classes.

Example 2

Let us now take a look at a more complex application using our account example.

Let us assume that the account administrator is to collect a bank charge for each account. With the savings account the amount is to be either withdrawn from the checking account of the account holder or billed separately, if he/she doesn't have a checking account. To make a general distinction between the access of an account holder and the access through the account administrator possible, we add the boolean parameter boAdmin to the i32RemoveFromAccount method of the parent class. We use this parameter in the parent class in order to be able to bill the full bank charge even if the credit limit is exceeded (simulated through a text output). In the savings account class we use boAdmin to withdraw the amount either from the checking account of the account holder (if he has a checking account) or to send an invoice (once again simulated through a text output). Thus, there are no plans to directly withdraw the bank charge from the savings account.

To do this, a pointer of the **account type** has to be added to the savings account class.

A reference to a checking account is passed and assigned to this pointer (it would also be possible to pass an appropriate pointer as a value; however, passing a reference makes work easier for the user and looks better.

Additionally, you should add a method that can be used to retrieve the assigned checking account number and a method to remove the reference to the assigned checking account (set pointer to `nil`).

As you can see we are already using the dynamic binding concept by using a pointer of the parent class type instead of using a pointer of the derived class type. This would make it possible later on to exchange the checking account with another type of account without having to change the savings account method.

Extend the constructor of the savings account class, so that the pointer is initialized with `nil`. No adjustments have to be made in the checking account class, since it is working with the methods of the parent class. Here you can see a major advantage of inheritance. Changes in the parent class automatically cause changes in the derived classes. This makes it easier to eliminate errors subsequently.

Exercise 1

Extend the parent class and the savings account class according to the above description. Test the individual functions in the Test.pool main program.

Solution 1

Solution (TestLib12.pli - declaration of the savings account class)

```
{Declaration of the derived savings account class}
type
  TestLib_tpoSavingsAccount = ^TestLib_toSavingsAccount;
  TestLib_toSavingsAccount = object(TestLib_toAccount){Delaration of the
                                                        savings account class}

    private                                             {Elements of the class}
    i32RemainingTime:    Int32;
    i32StartDate:        Int32;
    poCheckingAccount :  ^TestLib_toAccount;            {Pointer to an account
                                                         class}

    public                                              {Methods of the class}
    {Overridden constructor of the parent class}
    constructor poInit(i32AccountNumberPar:Int32;i32BalancePar:Int32;
                       i32CreditLimitPar:Int32);

    {Function to retrieve the remaining time}
    function i32GetRemainingTime:Int32;

    {Procedure to set the remaining time}
    procedure vSetRemainingTime(i32RemainingTimePar:Int32);

    {Procedure to set the checking account}
    procedure vSetCheckingAccount(var oCheckingAccountPar:TestLib_toAccount);

    {Procedure to remove the checking account}
    procedure vRemoveCheckingAccount;

    {Function to get the checking account number}
    function i32GetCheckingAccountNumber:Int32;

    {Overridden method of the parent class; has to be declared virtual in
     this class}
    function i32RemoveFromAccount(i32ValuePar:Int32;boAdmin:Boolean):Int32;
      virtual;

  end;
```

Solution (TestLib12_toSavingsAccount.pli - Only definition of the new methods of the savings account class)

```
{Assigning a checking account}
procedure TestLib_toSavingsAccount.vSetCheckingAccount(
                  var oCheckingAccountPar:TestLib_toAccount);
  begin
    poCheckingAccount := @oCheckingAccountPar; {Pointer to the checking
                                                account, from which the
                                                charge is to be withdrawn}
```

```
  end;

{Removing the reference to the checking account}
procedure TestLib_toSavingsAccount.vRemoveCheckingAccount;
  begin
    poCheckingAccount := nil;                  {Set the pointer to the
                                                checking account to nil}
  end;

{Get the account number of the assigned checking account}
function TestLib_toSavingsAccount.i32GetCheckingAccountNumber:Int32;
  begin
    if poCheckingAccount = nil then            {If no checking account is
                                                assigned}
      i32GetCheckingAccountNumber := 0;        {Zero is assigned to the
                                                account number of the
                                                checking account}
    else {Checking account exists}
      {Return of the checking account number}
      i32GetCheckingAccountNumber := poCheckingAccount^.i32AccountNumber;
    endif
  end;
```

Important

Checking i32GetCheckingAccountNumber for the `nil` pointer prevents access to a `nil` pointer and a resulting runtime error.

Solution (TestLib12_toSavingsAccount.pli - Constructor of the savings account class)

```
constructor TestLib_toSavingsAccount.poInit(i32AccountNumberPar:Int32;
                                            i32BalancePar:Int32;
                                            i32CreditLimitPar:Int32);
  begin
    {Call of the constructor of the parent class failed?}
    if inherited poInit(i32AccountNumberPar,i32BalancePar,
                        i32CreditLimitPar)= nil then
      poInit := nil;                   {Return value}
      return;                          {Termination, if the initialization
                                        has failed}
    endif;

    Writeln("Constructor of the savings account class is executed");
    {Initialization of the additional properties}
    poCheckingAccount := nil;           {Initialization of the pointer}
  end;
```

Solution (TestLib12_toSavingsAccount.pli - Method i32RemoveFromAccount of the savings account class)

```
{Overridden method of the parent class account (declared as virtual)}
function TestLib_toSavingsAccount.i32RemoveFromAccount(i32ValuePar:Int32;
                                                       boAdmin:Boolean):Int32;
  var
    i32Help: Int32;                              {Help variable}

  begin
    if boAdmin = false then
      {No special withdrawal selected}
      if (i32RemainingTime <= 0) then
        {Account is not closed for withdrawal}
        if (i32Balance - i32ValuePar) < i32CreditLimit then
          {The requested amount is higher than permissible}
          i32Help := i32Balance - i32CreditLimit;
          Writeln("You can't withdraw more than ", i32Help," Euros.");
          i32Balance := i32Balance - i32Help;
          i32RemoveFromAccount := i32Help;    {Return the withdrawn amount}
        else
          {Credit limit is not exceeded}
          i32Balance := i32Balance - i32ValuePar;
          i32RemoveFromAccount := i32ValuePar;
        endif;

      else
        {Account is closed for withdrawal}
        Writeln("Waiting period has not yet elapsed\n",
                "withdrawal possible after ", i32RemainingTime," days");
      endif
    else
      {Special withdrawal selected}
      if poCheckingAccount = nil then
        {There is no checking account}
        Writeln("Invoice: ", i32ValuePar ," Euros is written");
      else
        {Checking account exists}
        {Withdrawal from the checking account}
        Writeln("Charge for the savings account ",
                "that was withdrawn from the checking account:",
                poCheckingAccount^.i32RemoveFromAccount(i32ValuePar,boAdmin),
                " Euros.");
      endif                                      {if poCheckingAccount = nil}

    endif;                                       {if boAdmin = 0}

  end;
```

Solution (in TestLib_toAccount.pli - Method i32RemoveFromAccount of the account class)

```
{Function for withdrawal}
function TestLib_toAccount.i32RemoveFromAccount(i32ValuePar:Int32;
```

```
                                                boAdmin:Boolean):Int32;
  var
    i32Help: Int32;                             {Help variable}

  begin

    i32Help := i32Balance - i32CreditLimit;     {max. amount that can be paid}

    {Amount too high and there is no special withdrawal}
    if ((i32Balance - i32ValuePar) < i32CreditLimit)
         and (boAdmin = false) then

      Writeln("You can't withdraw more than ", i32Help," Euros.");
      i32Balance := i32Balance - i32Help;
      i32RemoveFromAccount := i32Help;          {Withdrawn amount}

    {Amount too high and there is a special withdrawal}
    elseif ((i32Balance - i32ValuePar) < i32CreditLimit)
             and (boAdmin = true) then

      Writeln("Invoice ", i32ValuePar ," Euros is written");
      i32RemoveFromAccount := 0;                {Withdrawn amount}
    {Amount can be withdrawn}
    else
      i32Balance := i32Balance - i32ValuePar;
      i32RemoveFromAccount := i32ValuePar;
    endif
  end;
```

Please don't forget to adjust the declaration of i32RemoveFromAccount in the account class (file TestLib.pli).

Exercise 2

Create an array with three elements in the account parent class. Assign an object of the three classes to the elements. Withdraw the bank fee from all three accounts using a loop and observe which method is called in each case.

Solution 2

Solution (Test12.pool)

```
module TEST;

import TestLib12;

private
procedure vMain;
var
  i32LoopCounter: Int32;

  {Declaration of the objects}
  oAccount1:         TestLib_toAccount;
```

```
  oCheckingAccount1: TestLib_toCheckingAccount;
  oSavingsAccount1:  TestLib_toSavingsAccount;

  {Array with pointer of the account type}
  apoAccounts:       array[0..2] of ^TestLib_toAccount;

begin
  {Call of the constructor for the first object}
  if oAccount1.poInit(1,500,-1000) = nil then
    Writeln(GetErrorMsg(GetError));    {An error message is put out}
    return;                            {Important: If return is called in
                                        vMain, this will lead to an abortion
                                        of the program.
                                        In practice an appropriate error
                                        handling is necessary depending
                                        on the application}
  endif;

  {Call of the constructor for the second object}
  if oCheckingAccount1.poInit(2,1000,-1000) = nil then
    Writeln(GetErrorMsg(GetError));    {An error message is put out}
    return;                            {Important: If return is called in
                                        vMain, this will lead to an abortion
                                        of the program.
                                        In practice an appropriate error
                                        handling is necessary depending
                                        on the application}


  endif;

  {Call of the constructor for the third object}
  if oSavingsAccount1.poInit(4,2000,0) = nil then
    Writeln(GetErrorMsg(GetError));    {An error message is put out}
    return;                            {Important: If return is called in
                                        vMain, this will lead to an abortion
                                        of the program.
                                        In practice an appropriate error
                                        handling is necessary depending
                                        on the application}
  endif;

  {Assignment of addresses to the pointer variables}
  apoAccounts[0] := @oAccount1;
  apoAccounts[1] := @oCheckingAccount1;
  apoAccounts[2] := @oSavingsAccount1;

  {Withdrawing the charge for all accounts}
  for i32LoopCounter := 0 to 2 do
    Writeln(apoAccounts[i32LoopCounter]^.i32RemoveFromAccount(100,true));
  endfor;

  {Call of the destructors}
  for i32LoopCounter := 0 to 2 do
    apoAccounts[i32LoopCounter]^.vDone;
  endfor;
end;
```

```
begin
end.
```

As you can see it is possible with the help of polymorphism to administer all accounts in an array regardless of the exact type. In practice you would of course use a linked list instead of an array.

We have now reached the end of our account example. If you want to continue practicing, you can override the copy constructor of the account class in the savings account and checking account classes.

4.6.2 Virtual constructors

Since the virtual methods table (VMT) does not yet exists at the time the constructors are processed, there are no virtual constructors.

4.6.3 Virtual destructors

Theoretically, it is also possible to use static destructors. However, it will not be guaranteed that the correct destructor method is called in each case.

Therefore, destructors should always be declared as virtual. The vDone destructor of the poRoot base class is also declared virtual.

5 Important Differences between C++ and POOL

This section once again summarizes the major differences between POOL and C++ in a table. For detailed descriptions on the contexts please see the previous sections.

C++	POOL

Definition of a class with the key word `class:`
`class` Classname
{
}
	Definition of a class with the key word `object:`
	`type`
	Classname = `object`
	`end;`

A distinction is made between the terms class (type) and object (instance of the class).	A distinction is made between object (type) and instance (instance of the object).
Constructor and destructor are automatically called.	Constructor and destructor have to be called explicitly. Reason: There might be several constructors/destructors.
Automatic calling of constructors of the parent class.	The constructor of the parent class has to be explicitly called in the constructor of the derived class: `inherited`(poInit) Reason: There might be several constructors.
Automatic calling of destructors of the parent class.	The destructor of the parent class has to be explicitly called in the destructor of the derived class: `inherited`(vDone) Reason: There might be several destructors.
The constructor does not return anything.	The constructor returns a pointer to the object. `nil` is returned in case of error. The function result can be ignored if necessary.
The constructor and the class have the same name.	The constructors are declared with the key word `constructor` and can receive any name. The standard constructor of the base class toRoot is poInit.
The destructor and the class have the same name with a preceding tilde.	The destructors are declared with the key word `destructor` and can receive any name. The standard destructor of the base class toRoot is vDone.

No distinction is made between functions and procedures (in reality this happens via the imaginary return type `void`).	A distinction is made between functions (`function`) with return value and procedures (`procedure`) without return value.
Specification of the parent class (inheritance): `class` Classname: parentclass	Specification of the parent class (inheritance): `type` Classname= `object`(parentclass);
Multiple inheritance is possible.	Multiple inheritance is not possible.
Inheritance as `public`	Declaration in the public area of the object (`public`).
Inheritance as `protected`	Only possible, if the derived class is implemented in the same library as the parent class. In this case an access to the private elements of the parent class is possible. There is no separate key word for the access to private elements of derived objects from outside the library.
Inheritance as `private`	Declaration in the private area of the object (`private`).
Methods in a derived class can be overidden.	With static methods it is permissible to overide methods with a modified prototype.
Operators can be overloaded.	Overloading of operators is not possible.
Functions can be overloaded.	Overloading of methods is not possible.
Reference to the object itself: `this`	Reference to the object itself: `oSelf`
Address operator: &	Address operator @ (see also part 1 of the tutorial)
Passing parameters as a reference is not possible. Auxiliary system is used by passing a pointer as a value: `void` vTestFunc (`int &`iPar);	Transfer of parameters as a reference: `procedure` vTestProc(`var` iPar: `Int32`);
importing libraries is not possible. Only the input of prototypes via .h-Files. #include “OwnLibrary.h“	Importing libraries: import OwnLibrary;
Including source text: #include “OwnHeader.h“	Including source text: {$i OwnInclude.pli}

6 Summary

After having become familiar with OOP, you should put this concept to use by creating your own project and thus deepening your knowledge.

Please remember when working on your project that the correct classification into classes and interfaces is the most important step. You should take your time during this process, since an incorrect classification could jeopardize the success of the project.

Modelling classes and interfaces is often realized using UML (Unified Modelling Language*)* in larger projects. At this point we want to recommend again that you use this type of approach.

UML is a language that can be used to create software systems with the help of diagrams and graphical elements. You can find introductions into UML on the Internet. To find them use search engines such as www.google.com and use the terms UML Introduction.

Attachment

A1 Bibliography

Books:

C++ - For Dummies, Stephen R. Davis, International Thomson Publishing.
Very simple introduction into the object oriented programming in C++.

GoTo C++ Programming, Andre Willms, ADDISON-WESLEY.
Good and extensive description of object oriented programming in C++.

C/C++-Kompendium, Dirk Louis, Markt and Technik Verlag.
Very extensive programming description in C and C++.

Internet Addresses:
http://www.uni-regensburg.de/Fakultaeten/WiWi/niemeyer/edu/Winter00/tp2/
Very good introduction into object oriented programming in Pascal.
(Chapters 7 and 8).

http://www.br-online.de/alpha/it-kompaktkurs/cplus.html
Complete introduction into object oriented programming in C++.

http://www.br-online.de/alpha/it-kompaktkurs/java
Brief introduction into object oriented programming in Java.

http://www.sigs-datacom.de/sd/publications/os/1998/02/objectspektrum_UM_kompakt.htm
Introduction into UML.

A2 Index

A

Abstract data type....14
Abstraction level....6
Access control....15, 39
Access rights with inheritance....43
Accessing objects....26

B

Base class....18, 42
Base constructor....30

C

Chained destructor calls....50
Class declaration in POOL....24
Class instance....14
Class variable....14
Classes....14
Classes in POOL....24
Constructor call....25, 31
Constructor definition....30
Constructor with parameters....32
Constructors....18
Constructors in POOL....29
Copy constructor....36
Copying an object....36

D

Deallocate memory....33
Declaration the properties....21
Deep copy....38
Derived class....18
Destructor....18, 32
Destructor call....34
Destructor of the base class....33
Destructors in POOL....33
Dispose....33
Dynamic access....50
Dynamic Memory Management....31

E

Encapsulation....8
Encapsulation in POOL....21
External function call....28

F

Functions....17
Functions in POOL....27

G

Graphical User Interface....9
GUI....9

H

High Abstraction level....6

I

Import libraries....25
Include file....22
Inheritance....18
Inheritance - private....43
Inheritance - protected....43
Inheritance - public....43
Inheritance in POOL....42
inherited....30, 34, 50
Inherited constructors....30
Inherited destructors....50
Instance of a class....14
Instanciation process....14
Interface....8, 15
Interface in POOL....21
Internal function call....28

L

Low abstraction level....6

M

Methods....7, 14
Methods - Dynamic binding....55
Methods - Static binding....55
Methods of objects....17, 27
MFC....16
Model....6
Multilayer class hierarchy....18

N

nil pointer....30

O

Object library....22
Objects....14
Objects in POOL....24
OOP....13
oSelf....31
Overriding constructors....49
Overriding destructors....50
Overriding dynamic methods....48
Overriding methods....48
Overriding properties....48
Overriding static methods....48

P

Parent class....43
Parent element....43

poInit 30
Pointer to an object 25
Polymorphism 20
Polymorphism in POOL 50
private 21
Private area 15
Private methods 17
Private properties 21
Procedural (functional) programming 6
Procedures 18
Procedures in POOL 28
Properties 14
Properties in the private area 27
Properties in the public area 27
Properties of objects 17, 27
Protected 43
Prototypes 20
public 21
Public area 15
Public methods 17

R

Records 14
Reuse 13

S

Scope 23
Shallow Copy 36
Static binding 20, 55
Static declaration 34
Static destructors 63
Static methods 48
Structure 14

T

toRoot 33
type 24

U

UML 66

V

var 25
Variables 7
vDone 33, 63
Virtual constructors 63
Virtual destructor - vDone 63
Virtual destructors 63
Virtual methods 48
Virtual Methods 18
Virtual methods in POOL 50

W

Why OOP 13