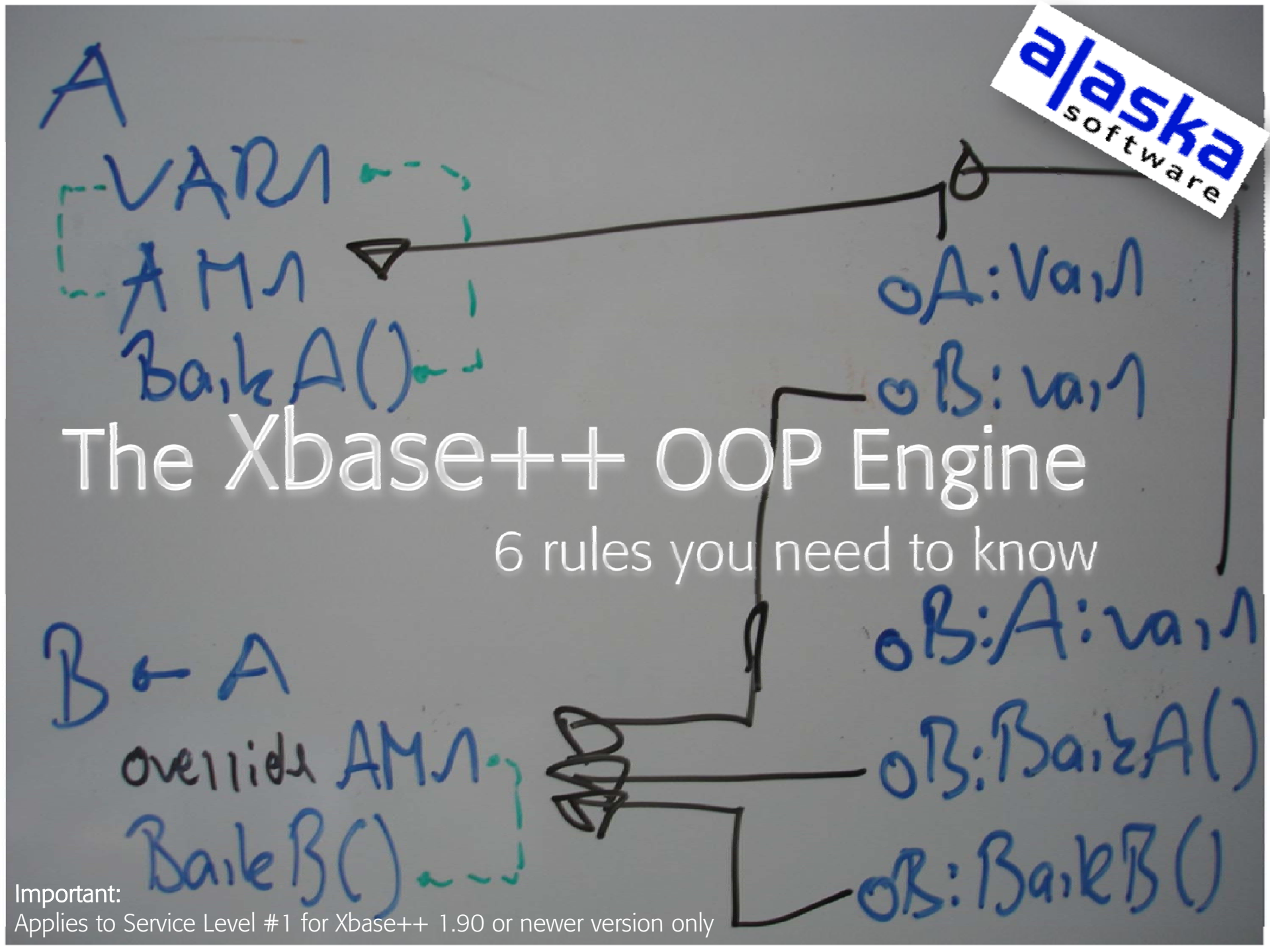


The Xbase++ OOP Engine

6 rules you need to know

Important:
Applies to Service Level #1 for Xbase++ 1.90 or newer version only

Rule No. 1

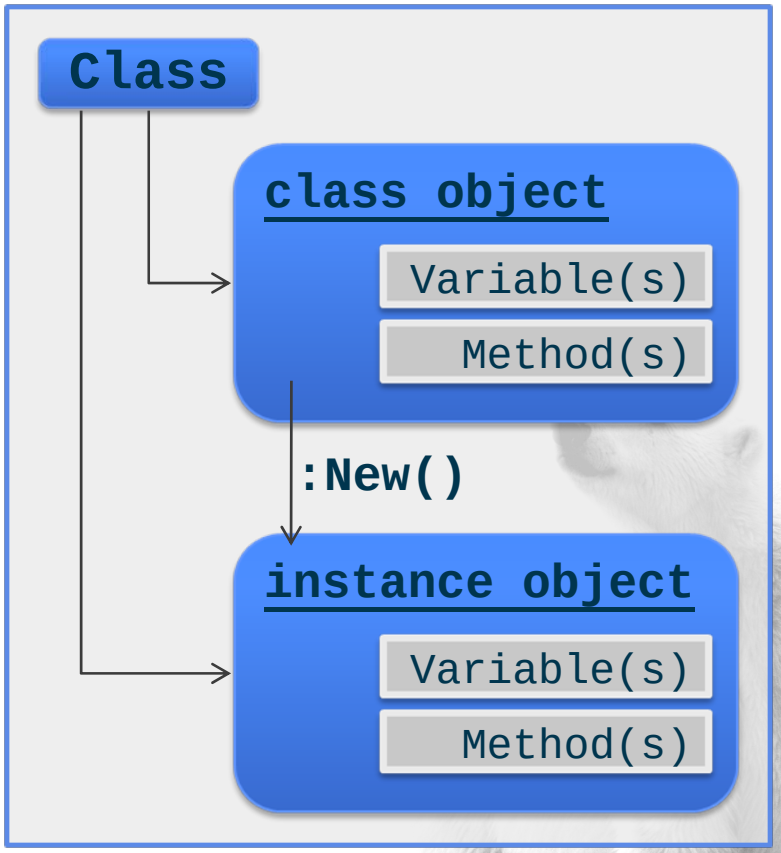
- Objects are everywhere!

„Class“ is an abstract term.

The Xbase++ runtime knows class objects and instance objects. Class objects are created at the first time the class function is executed. Subsequent executions of a class function then return the same class object.

To „free“ a class object, the function ClassDestroy() can be used. The Xbase++ runtime has a process-global hash table used to associate class objects with class names. ClassDestroy() simply removes the name<->class object association. However, a class object is not freed until the last instance object of the class is destroyed.

:New() is the default constructor to create instance objects. Instance objects are always created by class objects. Therefore, class objects are sometimes called „object factories“. You can create an unlimited number of instance objects of the same class. In the following, the term „object“ is used to describe instance objects.



Note: Class objects which reference code or data in a DLL lead to side effects, such as that the DLL is not unloadable until all objects and the class objects themselves have been destroyed. Until this precondition has been met, the DLL can not be unloaded.

Rule No. 2

- All Classes are dynamic

There is no difference between a class object created from a class function A() generated by the compiler and a class object created with CreateClass() by a Function A().

Runtime-classes can replace compile-time classes and vice versa. Any class can be replaced by another class with the same name. That‘s why classes are dynamic.

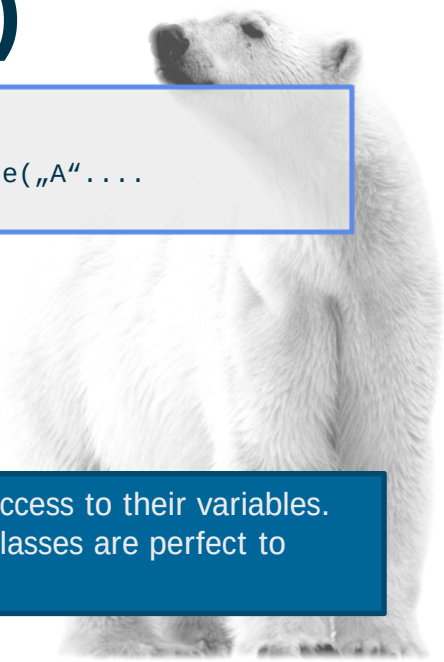
The FREEZE attribute in a CLASS declaration prohibits the implicit replacement of the class. FREEZE can be used to protect class implementations from being replaced implicitly.

The FINAL attribute prohibits subclassing of a class. The behaviour of FINAL classes can therefore no longer be changed – it is FINAL.

```
CLASS A
  EXPORTED:
  VAR Lastname
  INLINE METHOD Init()
    ::LastName := „hello“
  RETURN
  INLINE METHOD PrintA()
    ? ::LastName
  RETURN
ENDCLASS
```

A()=A()

```
FUNCTION A()
  LOCAL oCO
  oCO := ClassCreate(„A“....
RETURN(oCO)
```



Note: Compile-time classes can execute their methods faster and have in many cases faster access to their variables. In general, compile-time classes should be preferred over runtime classes. However, runtime classes are perfect to dynamically create types and behaviour.

Rule No. 3

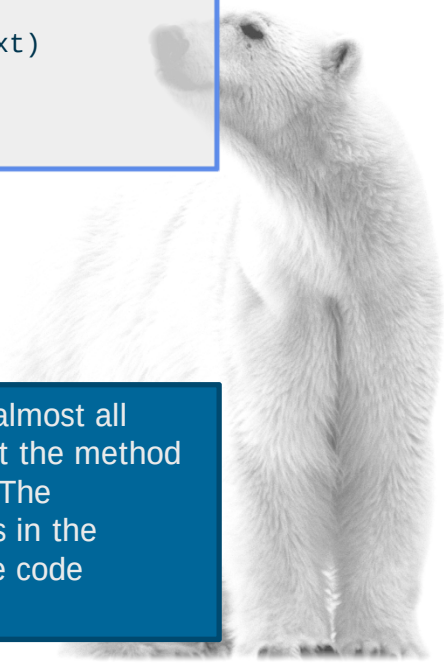
- METHODs are by default VIRTUAL OVERRIDE

Consider the code on the right side: Executing the method :Bark() on objects of Class A executes method :Print() in Class A. Executing the method :Bark() on objects of Class B executes the method :Print() implemented in Class B.

This behaviour is called virtual override because each method implementation in a subclass overrides a possible implementation of its superclasses

```
CLASS A
  EXPORTED:
  INLINE METHOD Bark()
    ::Print( „Wuff Wuff“ )
  RETURN
  INLINE METHOD Print(cText)
    ? cText
  RETURN
ENDCLASS

CLASS B FROM A
  EXPORTED:
  INLINE METHOD Print(cText)
    ? „I say:“,cText
  RETURN
ENDCLASS
```



Having all methods virtual by default leads to a very natural behaviour. As a matter of fact, in almost all cases when a method is redefined we want the method of the subclass to be executed and not the method of the superclass. Other languages such as C++, C# require to plan ahead for this behaviour. The implementor of the superclasses has to add the virtual keyword to his/her method declarations in the baseclass. This makes customizing of behaviour of class frameworks without having the source code sometimes complicated or even impossible.

Rule No. 4

- Overriding of METHODs is always a replacement

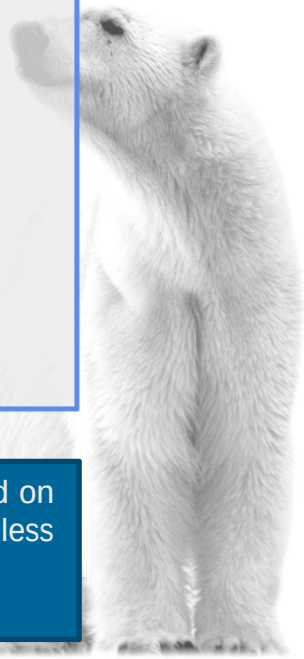
A method replacement completely overwrites the method of the baseclasses. That is, the code in the baseclasses is never executed when instances of subclasses are manipulated. The Class B and the method :Print() is an example of method replacement.

Refinement can be achieved as outlined in Class C. To do that, we must be able to execute the superclass‘ method implementation. SUPER is used to refer to the direct superclass implementation. The SUPER statement executes the superclass‘ implementation of the current method and passes along all its parameters. SUPER:Print() can be used as an alternative, and allows to specify the method name and the parameters passed.

```
CLASS A
  EXPORTED:
  INLINE METHOD Bark()
    ::Print( „Wuff Wuff“ )
  RETURN
  INLINE METHOD Print(cText)
    ? cText
  RETURN
ENDCLASS

CLASS B FROM A
  EXPORTED:
  INLINE METHOD Print(cText)
    ? „I say:“,cText
  RETURN
ENDCLASS

CLASS C FROM A
  EXPORTED:
  INLINE METHOD Print(cText)
    SUPER
    SUPER:Print( „I say:“+cText )
RETURN
ENDCLASS
```



Note: You can write SELF:A:Print(...) as an alternative to SUPER:Print() but this makes your code depend on the superclass names. Furthermore, using SELF:A:Print() increases maintenance costs and makes code less robust against changes. In general, the best approach is to use SUPER as this makes an implemention immune against class hierarchy, method name and parameter changes.

Rule No. 5

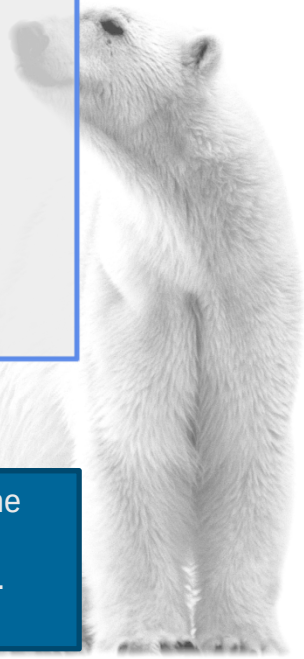
- VARIABLES are by default INTRODUCED

Consider the code on the right side: Executing the method :PrintA() on objects of Class B prints „Value from A“, while executing the method :PrintB() shows „Value from B“.

This behaviour is called hiding in OO terms. It simply means that redefining a variable in a subclass hides the variable of its superclass. The variable is therefore introduced in the subclass.

```
CLASS A
  EXPORTED:
  VAR Lastname
  INLINE METHOD Init()
    ::LastName := „Value from A“
  RETURN
  INLINE METHOD PrintA()
    ? ::LastName
  RETURN
ENDCLASS

CLASS B FROM A
 EXPORTED:
  VAR Lastname
  INLINE METHOD Init()
    ::A:Init()
    ::LastName := „Value from B“
  RETURN
  INLINE METHOD PrintB()
    ? ::LastName
  RETURN
ENDCLASS
```



Note: Encapsulation is a language feature that facilitates the bundling of state (data) with behaviour (the methods) operating **on that data**. Consequently, method implementations in different classes in a class hierarchy accessing a variable are automatically bound to the value related to their implementation level.

Rule No. 6

- ACCESS/ASSIGN methods are OVERRIDE, too

Consider the code on the right side: Accessing variable :Var1 on objects of Class B returns „FromB:value-a“. Accessing variable :Var1 on objects of Class A returns „value-a“. Accessing variable :A:Var1 on objects of Class B again results in „value-a“.

Introducing a new :Var1 at Class C simply hides the access/assign var of our baseclass. Therefore oC:Var1 is NIL while oC:A:Var1 is „value-a“.

```
CLASS A
  EXPORTED:
  VAR Var1
  INLINE ACCESS METHOD GetVar1() VAR Var1
  RETURN(::Var1)
  INLINE METHOD Init()
    ::Var1 := „value-a“
  RETURN
ENDCLASS

CLASS B FROM A
  EXPORTED:
  INLINE METHOD GetVar1()
  RETURN(„FromB:“+::Var1)
ENDCLASS

CLASS C FROM A
  EXPORTED:
  VAR Var1
ENDCLASS
```

When designing larger class frameworks, we strongly recommend to make all ACCESS/ASSIGN methods FINAL to ensure that class users do not accidently change behaviour. FINAL methods can still be replaced in subclass implementations by using the OVERRIDE attribute at the subclass.


alaska
software
Please visit us at
www.alaska-software.com
Sales DataBase
Shipper
ShipperKey
ShipperID
ShipperName
Product
ProductKey
ProductID
CategoryID
Price
ProductCategory
Employee
EmployeeKey
EmployeeID
ItemAll
Discount
Title
Address
City
Region
Company
ContactName
Action
WeekOfYear
Salary
Computed