# Visual FoxPro vs JAVA

Francis FAURE

Pour les conférences AtoutFox à Lille 2014

# Votre intervenant

**Francis FAURE**

46 ans, marié, 3 enfants

Gérant d'une SSII « Design or Decline » depuis 1991
Gérant de l'ISP « Wan Again » depuis 1997
Formation : Ingénieur CNAM

Membre actif et du bureau AtoutFox depuis 2004, webmaster www.atoutfox.org

MVP Microsoft Visual FoxPro de 2006 – 2011

VFPX Award 2010

# Agenda

- **Objet :** Session de découverte du langage Java par un développeur Visual FoxPro. *(Sondage )*
  Puis, d'essayer de répondre à la question « Peut-on faire en Java ce que l'on fait en VFP ? »
- **Présentation :** Proposition d'une présentation où chaque diapositive contient une comparaison VFP/JAVA afin de relever les analogies & les différences.
- **Cette session n'est pas un cours Java !**
  Pourquoi ?
  - Elle s'adresse à des développeurs VFP ne connaissant pas Java (ni C#) mais curieux d'un survol simplifié, donner des pistes pour vos propres recherches.
  - Votre intervenant a un recul de 25 ans et une expertise sur VFP, mais découvre Java que depuis de seulement quelques mois...

Challenge : présentation en 1h15…

# Historique

Origine

1984 - Fox Software releases FoxBase

1989 - FoxPro 1.0 for DOS
1991 - FoxPro 2.0 for DOS

1992 – Rachat de Fox Software par MICROSOFT ($173M)
1993 - FoxPro 2.5

Versions Microsoft

1995 - Visual FoxPro 3 (TAZ)
1996 - Visual FoxPro 5 (ROADRUNNER)
1998 - Visual FoxPro 6 (THAOE)
2001 - Visual FoxPro 7 (SEDONA)
2003 - Visual FoxPro 8 (TOLEDO)
2004 - Visual FoxPro 9 (EUROPA)
2007 - Sedna and SP2 for VFP 9

2007 – Annonce du 13 mars pas de vfp 10

---

Origine

1991 – Langage « Oak », / Green **James Gosling** pour Sun

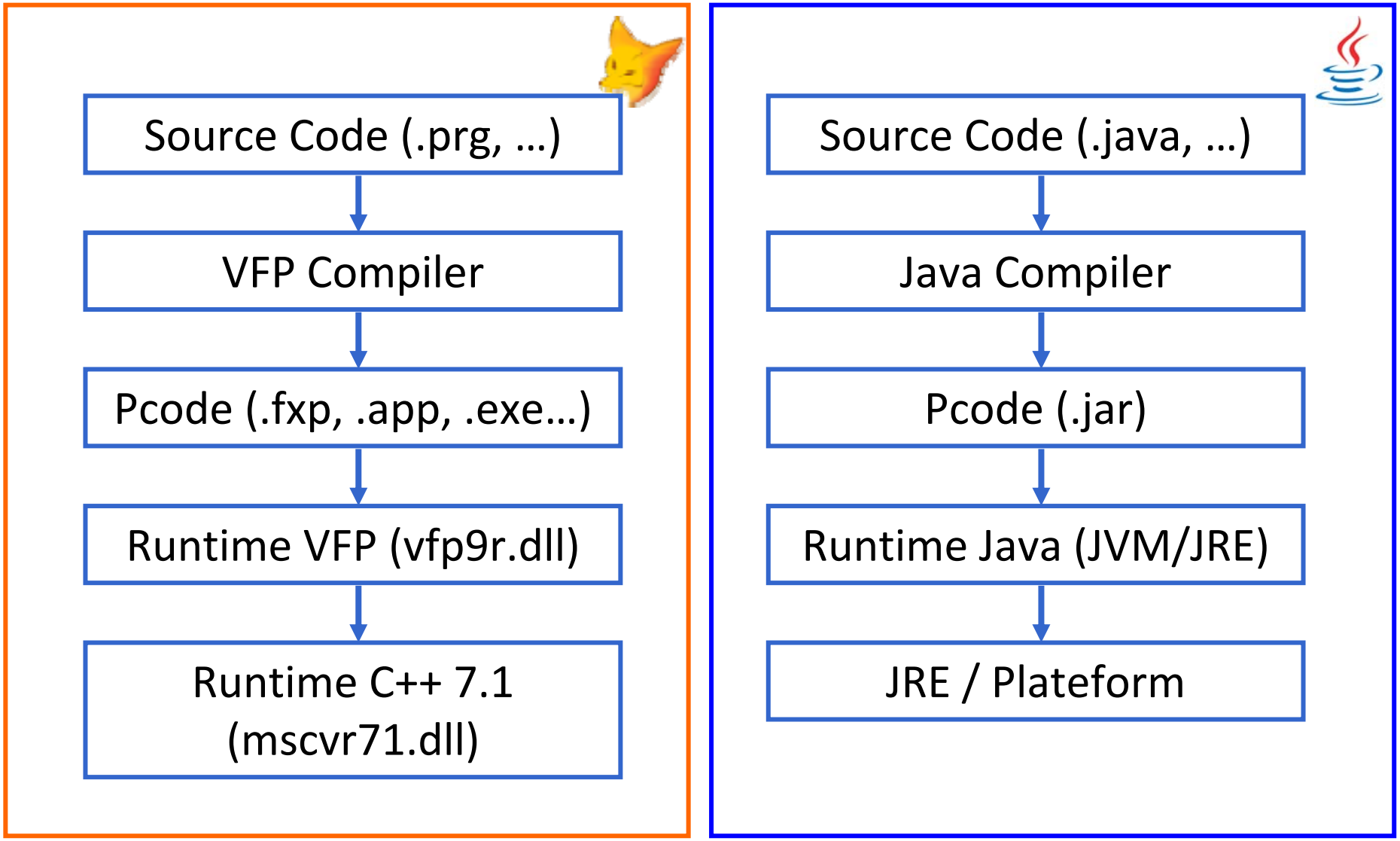
1995 – Version 1.0
1997 – Version 1.1
1998 – Version 1.2 (PLAYGROUND)
2000 – Version 1.3 (KESTREL)
2002 – Version 1.4 (MERLIN)
2004 – Version 5.0 (TIGER)
2006 – Version 6.0 (MUSTANG)

2006 : Open Source & Free

2009 – 2010 Rachat de Sun Microsystems par ORACLE

2011 - Version 7.0 (DOLPHIN)
2013->2014 – Version 8.0 (WOLF?)

# Dynamic


Source Code (.prg, …)
VFP Compiler
Pcode (.fxp, .app, .exe…)
Runtime VFP (vfp9r.dll)
Runtime C++ 7.1 (mscvr71.dll)
Source Code (.java, …)
Java Compiler
Pcode (.jar)
Runtime Java (JVM/JRE)
JRE / Plateform

# Editions

Visual FoxPro version 3 à 9 :

**un seule édition**

Historiquement :

De mémoire, pour Foxpro version **2.x**

il existait :

- Une version « standard »
  (pour développeur)
- Une version « Pro »
  (pour générer des exécutables)
- Un kit de « connectivité »
  (Librairie Connexion SGBDR)

**Java SE :**

Standard Edition

-> Applications / postes autonomes

**Java EE :**

Enterprise Edition

-> Applications Web

**Java ME :**

Mobile Edition

-> Application sur équipements (Téléphone, tablette, téléphone, autoradio, ….)

Même langage : mais classes différentes

Ne pas confondre le langage Java et le « Javascript » (Normalisé aussi à ECMA mais qui est un langage de script, non typé, et interprété dans les navigateurs)

# Plateformes

Visual FoxPro version 3 à 9 :

**Microsoft Windows 32 bits**

Historiquement :

Visual FoxPro version 3.0
- Microsoft Windows 32 bits (1ère appli 32 MS)
- Apple Mac OS (OS/9? 68000)

FoxPro version 2.5 et 2.6
- Microsoft DOS
- Microsoft Windows 16bits
- Mac OS (OS7-8? 68000)
- Unix

FoxBase
- Xenix SCO, Unix SCO, DOS

Mais pas « cross-plateforme »

Conçu nativement **Multiplateformes**.

OS de développement (JDK) :
- Windows 32 et 64 bits
- Linux 32 et 64 bits
- MAC OS X (64 bits - base FreeBSD)
- Solaris 32 et 64 bits
- Solaris SPARC 32 et 64 bits
- Linux ARM
- …

OS d'exécution : toutes les « Plateformes » disposant d'un runtime Java (JRE)… (Oracle parle de 3 Milliards d'équipements… en comptant téléphones et autres tablettes ?)

# Licences

- Commerciale (ex)
- Core : Closed Source

- Sedna et les « outils » satellites de VFP + doc versés en « Shared Source » sur vfpx.codeplex.com

- Gratuite
- Open Source

# Langage

- Dérivé (inspiration) initiale de dBase Ashton-Tate
- Clone dBase,xBase

- Dérivé (inspiration) initiale de C/C++
- Normalisé à l'ECMA

# Concept de programmation

- Procédurale
  (Puis événementielle en version 2.x)

  +
- « Orienté » Objets

(depuis version 3)

(Compatibilité ascendante)

- Objets

(Tout est Objet)

# Périmètre

**Un Langage**
+
**Un IDE**
+
**Un gestionnaire de fichiers**
à ‘plat’ (DBC/DBF)
+
**Connectivité SGBDR**
(odbc)

Langage Orienté Données (Data Centric)

Communauté de plus en plus faible

**Un Langage**

Le reste sont des ‘briques’ complémentaires à sélectionner et assembler selon ses besoins propres :

IDE : Notepad (☺), Eclipse, NetBeans…
Vaste choix de Frameworks,
Contrôle de sources, versionning, IC, …

Connectivité SGBDR (jdbc)
Communauté importante

# Garbage Collector

- Le langage n'intègre pas de mécanisme d'allocation de mémoire (type malloc), ni de libération de la mémoire (type free).
- Le Runtime gère cela grâce à un mécanisme complexe appelé : « ramasse miettes » (G.C.)
- C'est une abstraction moderne de 'haut niveau' :
  - Gain de temps programmation
  - Sécurisant les oublis de libération de ressources mémoire
  - Abstraction : pas de 'pointeur' physique
  - …

- Identique -

VFP Précurseur ?

# Code Source

- Fichiers « textes » (.prg)

Mais aussi :

- En formulaire (.scx/.sct)
- En classe visuelle (.vcx/.vct)
- En rapport (.frx/.frt)
- En bdd (.dbc/.dct)
- En menu (.mnx/.mnt)
- En label (.lbx/.lbx)
- Etc…

Fichiers « textes » (.java)

.properties (texte)

# CaSe SeNsITiVe ?

VFP n'est pas sensible à la 'casse'.

Les '**Noms**' (variables, procédures, méthodes, champs,… peut être indifféremment écrit dans le code source en Majuscules comme en Minuscules (ou même mixé !) tout en désignant **la même** ressource. (case insensitive)

Concrètement :

```
iii=1
iii=III+1
IiI=iiI+2
```

Une seule variable "III« valant 4

JAVA **est sensible** à la 'casse'.

(**Tous les Noms** sont Case Sensitive) Concrètement :

```
iii=1;
iii=III+1;
```

(Erreur de compilation si III existe pas)

qui peut s'écrire : **iii++;** ☺

Concrètement :

```
iii=1;
```

Mais

```
III="Hello" ;
```

# Question de point-virgule ;

Le « ; » indique la continuité d'une commande sur la ligne suivante.

Exemple :

```
s=;
   "Hello"+;
   " "+;
   "World"
```

(Marquage robinet eau chaude)

Le « ; » indique la fin de la commande.

Exemple :

```
s=
   "Hello"+
   " "+
   "World";
```

(Classique)

# Commentaires

- **NOTE** Commentaire
- * Commentaire
- && Commentaire

- **Note** commentaire ;
  multi-lignes

```
// Commentaire

/*
  commentaire
  multi-lignes
*/

Javadoc
/**
 * @author Francis FAURE
 */
```

# Déclaration de Variables

En VFP un ‘type’ est renseigné dans la syntaxe ‘à droite’ (après) avec le token ‘**as**’

Exemple :

```
private i as Integer
i=1
```

En JAVA un ‘type’ est renseigné dans la syntaxe ‘à gauche’ (avant) sans token

Exemple :

```
Integer i ;
i=1 ;
```

A noter que l’on peut déclarer et affecter sur la même ligne . L’exemple précédent s’écrit aussi :

```
Integer i = 1 ; ☺
```

# Typage des Variables

**VFP n'est pas typé.**
Autre formulation :
**VFP est 'faiblement' typé**
car toutes les variables sont des « Variants » (Analogie VB/C++)

'Faiblement' = il n'est pas nécessaire de déclarer une variable pour l'utiliser.

VFP dispose d'une syntaxe de déclaration des variables : mais elle ne sert que dans l'IDE pour l'IntelliSense.

**Java est 'fortement' typé.**

Il faut déclarer les variables avec un type avant de pouvoir les utiliser.

Le type « Variant » n'existe pas. Rassurez-vous : tous les types de JAVA ont comme grand-père le type 'Object'.

D'une manière générale : en JAVA tout est typé. (on y reviendra)

# Portée des Variables

- Variable Publique

```
Public i as String
```

(portée globale à tout le programme, même déclarée dans une procédure...)

- Variable Privée

```
Private j as String
```

(portée au programme et à ses sous programmes)

- Variable Locale

```
Local s as String
```

(portée uniquement au programme / procédure / Méthode en cours)

Par défaut (omission) :

Private.

- Puisque tout est Objets… JAVA ne dispose de pas de ‘Variable publique’.

(Nous verrons plus loin comment disposer d’un mécanisme équivalent...)

- Une variable est donc ‘local’ (au sens VFP )

```
String s;
```

- Par contre nous retrouverons les portées de ‘Variables’ dans les propriétés.

# Portée des propriétés

- Propriété Publique

`prop1 = ""`

(La propriété est exposée et peut-être utilisée par les programmes instanciant un objet de cette classe)

- Propriété ‘Privée’

**`PROTECTED`** `prop2 = ""`

(portée à la classe et à ses sous-classes)

- Propriété ‘Locale’

**`HIDDEN`** `prop3 = ""`

(portée uniquement à la classe)

Par défaut : Publique.

On retrouve cette déclaration pour les méthodes

- Propriétés Publiques

**`public`** `String prop1 = ""`

(La propriété est exposée et peut-être utilisée par les programmes instanciant un objet de cette classe)

- Propriétés ‘Privées’

**`protected`** `String prop2 = ""`

(portée à la classe et aux classes filles, mais aussi aux classes du même **espace de noms** -> Explications suivent)

- Propriétés ‘Locales’

**`private`** `String prop3 = ""`

(portée uniquement à la classe)

Par défaut : Protected.

On retrouve cette déclaration pour les méthodes

# Espace de noms

- Il n'y a pas 'd'espace de noms' dans VFP.

Autre formulation :

- Il y a qu'un seul 'espace de noms'…

En fait :
on peut faire une analogie 'lointaine' avec la commande :

```
set procedure to
```

Illustration :

```
set procedure to proc1
o=createobject("classe")
set procedure to proc2
o=createobject("classe")
```

- JAVA dispose d'une déclaration forcement dans un espace de noms que l'on appelle:

**package**

Le nom étant unique dans l'espace de noms: Il n'y a donc pas de confusion possible entre deux classes du même nom dans 2 package différents (que l'on `'import'`).

(Similaire au NAMESPACE)

# Type 1/3 : Types ‘Primitifs’

## Pas de Type

(ou un seul type ‘Variant’ cf ci-avant)

En fait en Interne VFP gère des types pour la génération de son PCode (ByteCode) c’est logique car le runtime est en C++

boolean (1) :  un token 0x61 .T. / 0x2D .F.

byte (1) :  attention non signé (0 / 255)
short(2) : signé
int (4) : signé
Double (8) : format IEEE 754 « Double-precision floating-point format »
- Sign Bit : 1 bit
- Exponent : 11 bits
- Significand precision : 53

Plus d’info sur :
http://en.wikipedia.org/wiki/Double_precision

## Types « Primitifs »

(Tout est objet… enfin presque…)

boolean (1) : true / false
byte (1) : attention il est signé -128 / +127
short (2) : signé -32 768 / +32 767
int (4) : -2*10^9 / 2*10^9
long (8) : -9*10^18 / 9*10^18
float (4) : virgule flottante (réel simple)
double (8) : virgule flottante (réel double)
char (2) : Caractère Unicode

Nota : Ils n’ont pas de majuscules.

(Dans le roadmap de JAVA version 10 les types primitifs seraient supprimés…)

# Type 2/3 : **Les enveloppeurs (Wrappers)**

Pas d'équivalent.

Mais on pourrait créer des classes sur le même principe.

## Wrappers

(Tout devient réellement objets…)

| Enveloppeur | Type primitif |
|---|---|
| Character | char |
| Byte | byte |
| Short | short |
| Integer | int |
| Long | long |
| Float | float |
| Double | double |
| Boolean | boolean |

Exemple :

```
Integer i=1;
i.toString();
```

# Type 3/3 : Choix de Classes

- VFP : 19 classes ‘visuelles’ et une quantité limitée non visuelles.

- Effectivement cela couvre presque tout des besoins des (‘anciennes’) applications.
  Les solutions type treeview / calendrier / gantt / graphs … passent par OLE/ActiveX tiers

- En JAVA
  le nombre de classes est impressionnant et nécessite du temps d’exploration (3977 classes en version Java 7).
- Les classes Java sont en Java.

# C’est Classe

## Définition d’une classe:

```
define class MyC1 as custom
enddefine
```

## Instanciation d’un objet sur une classe :

```
local o as MyC1
o=createobject("MyC1")
```

## Définition :

```
public class MyC1 {
   }
```

/!\ Une classe = un fichier du même nom !

## Instanciation :

```
MyC1 o;
o = new MyC1();
```

S’écrit sur une seule ligne :

```
MyC1 o = new MyC1();
```

# Classe suite

```
define class MyC1 as custom
enddefine

define class MyC2 as MyC1

prop1 = "Hello World"

function method1(param1 as
string) as string
  this.prop1=UPPER(m.param1)
  return this.prop1
endfunc

enddefine

local o as MyC2
o=createobject("MyC2")
? o.method1("Hello Fox !")
```

```
public class MyC1 {}

public class MyC2 extends
MyC1{
String prop1 = "Hello World";

String method1(String param1)
 {
  this.prop1=
        param1.toUpperCase();
  return this.prop1;
  }
}
MyC2 o = new MyC2();
System.out.println(o.method1
("Hello Fox !"));
```

# Classe suite – this & paramètres

```
define class MyC as custom
prop1 = "Hello World"

function method1(prop1 as
   string) as string
  If type("m.prop1")=="C"
    this.prop1=m.prop1
  endif
  return this.prop1
endfunc
enddefine

local o as MyC
o=createobject("MyC")
? o.method1()
```

```
public class MyC {
String prop1 = "Hello World";
String method1(String prop1)
  {
  this.prop1=prop1;
  return this.prop1;
  }
String method1()
  {
  return this.prop1;
// return this.method1(this.prop1)
  }
}
MyC o = new MyC ();
System.out.println(
   o.method1());
```

# Procédures / Fonctions ?

- VFP dispose de deux déclarations :

**Procedure … EndProc**
**Function … EndFunc**

*(dans sa compilation le ByteCode généré par VFP est en fait le même pour les deux syntaxes.)*

- Une déclaration de « Function » qui ne contiendrait pas un « return » ne lève pas d'erreur. *(valeur par défaut retournée .T.)*
- Comme une déclaration de « Procedure » peut contenir un ou des « Return »…

- Une « Function » peut être appelée sans affectation.
(Token explicite "=" 0x86 implicite 0x99)
- Exemples :

```
z=UPPER("a")
=UPPER("a")
UPPER("a")
```

- Java ne dispose pas de déclaration de procédure ou de fonction au sens VFP. (Historique Fox Procédurale et puisque en Java tout est « objet »…)

*Mais, nous allons voir dans la diapo suivante que l'on peut faire la même chose !*

En analogie : il faut donc raisonner avec des méthodes (de classes).

- Une méthode qui retourne quelque chose doit être déclarée avec son type de retour. (équivalent d'une « Function » de VFP)

```
int ageDuCapitaine() …
```

Java impose alors une sortie « return » (du même type)

- Une méthode qui ne retourne rien doit être déclarée comme ne retournant rien avec le token « void ». (équivalent d'une « Procedure » de VFP)

```
void setAgeDuCapitaine(int a)..
```

- Comme VFP on peut appeler une méthode sans affectation.

# Static

De quoi s'agit-il ?

Le fait d'indiquer qu'une méthode est 'static' permet de l'invoquer sans créer d'instance de cette classe.

En VFP un objet doit forcement être instancié sur une classe (createobject(), addobject(), add object…) pour appeler une méthode.

L'équivalent de la déclaration d'une méthode 'static' en VFP n'existe donc pas.

En fait, en analogie, cela correspond à une Procédure / Fonction.

La déclaration 'static' d'une propriété n'existe pas en VFP.

En fait, en analogie cela serait une **variable publique**.

Exemple :

```
public class Cvfp
  {
  public static String
upper(String s)
    {
    return s.toUpperCase();
    }
  }
```

La déclaration 'static' d'une propriété indique que cette valeur de propriété est partagée par toutes les instances créées sur cette classe.

# Recommandation Setters / Getters
## « Accesseurs / Mutateurs »

Analogie en VFP :

On peut écrire de la même manière, voir utiliser :

_access()

_assign()

```
public class MyC {

private int prop = 0;

int getProp()
  {
  return this.prop;
  }

void setProp(int prop)
  {
  this.prop = prop;
  }
}
```

# Numeric Operators

| | |
|---|---|
| Addition | + |
| Subtraction | - |
| Multiplication | * |
| Division | / |
| Exponentiation | ** , ^ |
| Modulus | % *mod()* |
| Group | ( ) |

| | |
|---|---|
| Addition | + |
| Subtraction | - |
| Multiplication | * |
| Division | / |
| Exponentiation | **Math.pow(Double, Double)** |
| Modulus | **%** |
| Group | ( ) |

# Relational Operators

| | |
|---|---|
| Less than | < |
| Greater than | > |
| Not equal to | **<>, #, !=** |
| Less than or equal to | <= |
| Greater than or equal to | >= |
| Equal to | **=, ==** |
| Character string comparison | == |

| | |
|---|---|
| Less than | < |
| Greater than | > |
| Not equal to | **!=** |
| Less than or equal to | <= |
| Greater than or equal to | >= |
| Equal to | **==** |
| Character string comparison | **.equals(String)** |

# Character Operators

Concatenation +

Concatenation -

Comparison $

Concatenation +

x

Comparison

.contains(String)

# Logical Operators

Logical negative
NOT, ! , .NOT.

Logical AND
AND, .AND.

Logical inclusive OR
OR, .OR.

Expression groups
( )

Logical negative
!

Logical AND
&&

Logical inclusive OR
||

Expression groups
( )

# CONSTANTES

Directive de pré compilation

**#DEFINE**

Exemple :

```
#define VERSION 1
```

Java ne dispose pas de « directive de pré compilation »

Par contre il propose le mot clé « final » rendant similaire la réponse au besoin :

```
final int VERSION = 1;
```

# IF

Nul doute, que dans cette audience, tout le monde sait que la syntaxe d'un test conditionnel s'exprime comme (doc VFP) :

```
IF lExpression [THEN]
  Commands
[ELSE
  Commands]
ENDIF
```

Remarque : dans le Pcode le « then » est purement ignoré…

Version Java :

```
if (lExpression)
  {
  Commands ;
  }
[else
  {
  Commands ;
  } ]
```

Accolades optionnelles.

*Nota : les [ ] est une notation optionnelle et non la syntaxe.*

# IIF (opérateur ternaire)

```
variable = IIF(condition,;
valeur_si_vrai, ;
valeur_si_faux)
```

```
variable = condition ?
valeur_si_vrai :
valeur_si_faux ;
```

Attention au respect des types.

# CASE

doc VFP :

```
DO CASE
CASE lExpression1
  [Commands]

[CASE lExpression2
  [Commands]] ...

[CASE lExpressionN
  [Commands]]

[OTHERWISE
  [Commands]]
ENDCASE
```

En Java, il n'y a pas l'équivalent du « DO CASE » de vfp.

La structure approchante est le « switch » mais attention c'est très différent.

```
switch (variable) {
 case valeur1 :
   Commands;
   break;
 case valeur2 :
   Commands;
   break;
 case valeurN... :
   Commands;
   break;
default:
 Commands;
 }
```

Les String sont utilisables.

Attention à l'oubli du « break » !

# Boucle FOR

```
FOR VarName = nInitialValue ;
 TO nFinalValue ;
  [STEP nIncrement] ;
    Commands
  [EXIT]
  [LOOP]
ENDFOR | NEXT
```

Exemple :

```
for i = 1 to 10
next
```

Les valeurs : finale et d'incrément sont évaluées une seule fois au démarrage de la boucle.

```
for (VarName ; condition ;
modification compteur) {
 Commands
  [break;]
  [continue;]
  }
```

Exemple :

```
for (int i=1 ; i<=10 ; i++) {
  }
```

# Boucle WHILE

```
DO WHILE lExpression
   Commands
   [LOOP]
   [EXIT]
ENDDO
```

```
while (lExpression) {
   Commands
   [continue;]
   [break;]
   }
```

Ou bien

```
do {
   Commands
   [continue;]
   [break;]
   } while (lExpression);
```

# Tableaux

```
DIMENSION tableau[3] as Integer
tableau[1]=10
tableau(2)=20
tableau(3)=30

* Ou
DECLARE tableau[3]
```

**/!\ One Based**

```
int tableau[] = {10, 20, 30};
// ou
int tableau[] = new int[3];
tableau[0] = 10;
tableau[1] = 20;
tableau[2] = 30;
// ou
int[] tableau3 = new int[3];
```

**/!\ 0 Based**

**Classes Collections : ArrayList, List, LinkedList, Vector.. Et Classe Map**

# Multitâches

VFP n'est pas Multitâches.
(pas de gestion de 'threads').

Voir librairie
« ParallelFox » sur vfpx

vfpx.codeplex.com/wikipage?title=ParallelFox

Java est nativement multitâches.

Un grand nombre de fonctionnalité concernant les threads (attente, réveil, synchro, etc…) sujet non détaillé dans ce survol car pouvant couvrir une session à lui seul.

# Classes « Abstraites »

VFP ne contient pas de définition de classes « Abstraites »

(Je n'ai pas trouvé d'équivalent)

C'est quoi une classe Abstraite ? C'est une classe sur lequel on ne peut instancier d'objet. Elle ne peut être utilisé que pour être héritée.

Exemple :

Une classe « Animal » pourrait-être définie pour être utilisée en définition de classes « Chat », « Chien », « Poule », etc…
En étant abstraite elle ne peut être instanciée mais permet de forcer la définition des méthodes (polymorphes) à surcharger.

En java la déclaration se fait avec le token « `abstract` »

# Exceptions

En matière de gestion des erreurs : historiquement FoxPro ne proposait que :

`ON ERROR …` ☹

Depuis VFP 8,
la nouvelle fonctionnalité proposée comblait ce manque : ☺

```
TRY [ tryCommands ]
[ CATCH [ TO VarName ] [ WHEN lExpression ] [ catchCommands ] ]
[ THROW [ eUserExpression ] ]
[ EXIT ]
[ FINALLY [ finallyCommands ] ]
ENDTRY
```

En Java :

```
try
   {
   }
catch (ClassCastExpression e)
   {
   }
Finally
   {
   }
```

Le bloc « catch » peut être multiple selon le type d'erreur.
Depuis Java 7 on peut définir un « multi-catch » exemple : `C1 | C2 e`

L'équivalent « THROW » de VFP s'écrit « throw » en Java..

# Pratique

Nous devons voir suffisamment de théorie pour programmer notre premier cas concret :

- Hello World sur la console
- Classe Hello World en console
- Hello World Equivalent MessageBox

« transition »

- CvfpLike -> upper()
- VFP_MESSAGEBOX()

# Prototype de « Labo »

**PARTIE II :**

Objet :

- Essayer de répondre à la question :
« Peut-on faire en Java ce que l'on fait en VFP ? »
- Ma meilleur réponse est d'envisager, me semble t-il, l'exécution du Pcode de VFP en JAVA.
- Survol du Pcode VFP
- Démonstration

# VFP ByteCode

- Non documenté
- Ressources :

M. Robert Plagnard :
DVFP - session AtoutFox 2006
Membre du Bureau 2009-2010

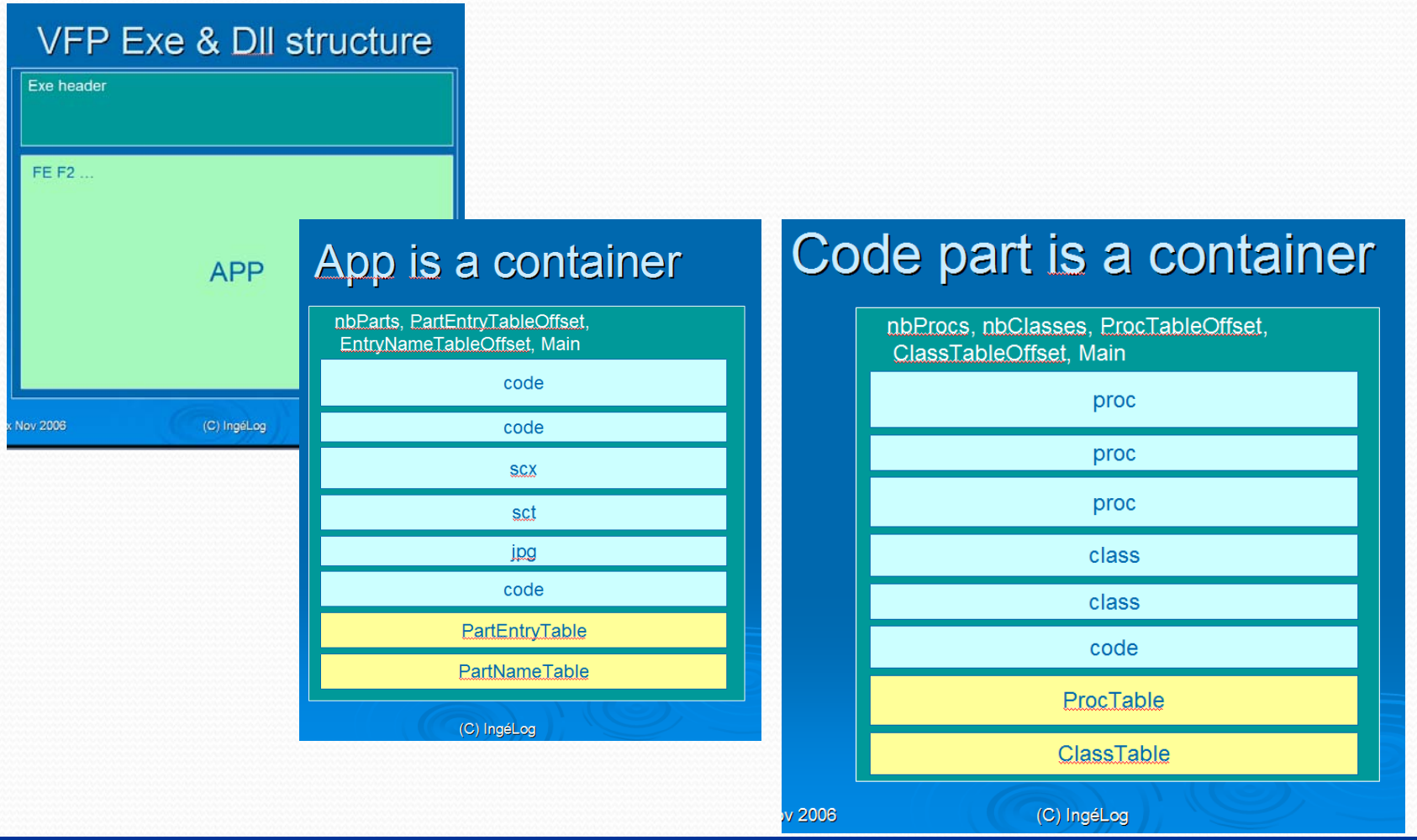
M. Christof Wollenhaupt (GUINEU)

Session Guineu Atoutfox 2007-2008

Publications :

http://www.foxpert.com/docs/howfoxproworks.en.htm

http://www.foxpert.com/docs/fxp.en.htm


VFP Exe & Dll structure
Exe header
FE F2 ...
APP
x Nov 2006
(C) IngéLog
App is a container
nbParts, PartEntryTableOffset, EntryNameTableOffset, Main
code
code
scx
sct
jpg
code
PartEntryTable
PartNameTable
(C) IngéLog
Code part is a container
nbProcs, nbClasses, ProcTableOffset, ClassTableOffset, Main
proc
proc
proc
class
class
code
ProcTable
ClassTable
v 2006
(C) IngéLog

# VFP PCode

## VFP source code :

```
MyVar = "World"
MESSAGEBOX("Hello " + MyVar, 4+48, "Title")
```

## ByteCode : Part Dump :

```
PART DUMP
00000000  00 00 00 00 25 00 00 00-00 00 00 00 00 00 00 00   ....%...........
00000010  6C 00 00 00 03 00 00 00-66 00 00 00 57 82 6A 44   l.......f...W.jD
00000020  38 00 00 00 FC 36 00 12-00 54 F7 00 00 10 FC D9   8....6...T......
00000030  05 00 57 6F 72 6C 64 FD-FE 21 00 99 FC 43 D9 06   ..World..!...C..
00000040  00 48 65 6C 6C 6F 20 F7-00 00 06 F8 03 34 D9 05   .Hello ......4..
00000050  00 54 69 74 6C 65 EA 78-FD FE 03 00 55 01 00 05   .Titleêx....U...
00000060  00 4D 59 56 41 52 23 01-11 02 31 00 00 00 00 00   .MYVAR#...1.....
00000070  00 00 00 00 00 00 00 00-00 00 00 00              ............
```

```
MyVar = "World"
MESSAGEBOX("Hello " + MyVar, 4+48, "Title")
```

```
PART DUMP
00000000  00 00 00 00 25 00 00 00-00 00 00 00 00 00 00 00   ....%...........
00000010  6C 00 00 00 03 00 00 00-66 00 00 00 57 82 6A 44   l.......f...W.jD
00000020  38 00 00 00 FC 36 00 12-00 54 F7 00 00 10 FC D9   8....6...T......
00000030  05 00 57 6F 72 6C 64 FD-FE 21 00 99 FC 43 D9 06   ..World..!...C..
00000040  00 48 65 6C 6C 6F 20 F7-00 00 06 F8 03 34 D9 05   .Hello ......4..
00000050  00 54 69 74 6C 65 EA 78-FD FE 03 00 55 01 00 05   .Titleêx....U...
00000060  00 4D 59 56 41 52 23 01-11 02 31 00 00 00 00 00   .MYVAR#...1.....
00000070  00 00 00 00 00 00 00 00-00 00 00 00              ............
```

0x54 Let (imp. Store)

:=

End proc

+ (plus)

Extended Function

MESSAGEBOX()

F7 00 00 F7 (VAR) + 2 Bytes = #VAR NTI

FC Start Expression ... FD End Expression

D9 D9 (String ) + 2 Bytes = Length

FE End Statement

99 0x99 Implicit Call
0x86 Explict Call (=)

43 Start Arg List

F8 03 34 F8 (Byte) + 2 Bytes width/value
0x34 = 52

# Démo

- test_0.fxp (helloworld)
- test_unit_tests.fxp
- test_gui.fxp
- Votre test ? (selon timing)

- Encodage RGB
- Dimension des fenêtres
- Style barre de titre
- …

# Objectif de session tenu ?

- **Objet :** Session de découverte du langage Java par un développeur Visual FoxPro.

  Puis, d'essayer de répondre à la question « Peut-on faire en Java ce que l'on fait en VFP ? »