Erfahrungen mit Xtext zur Entwicklung domänenspezifischer Sprachen und geplante Erweiterungen in Xbase

Reiner Jung Robert von Massow

Christian-Albrechts-Universität zu Kiel
Institut für Informatik

10. Nov 2010

MENGES KoSSE
ZUKUNFTSprogramm Schleswig-Holstein
Investition in Ihre Zukunft
Xbase
GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Teil I

DSL Entwicklung in Xtext

Reiner Jung

Inhalt

Domänen

- Topologie des Stellwerks: Gleise, Signale usw.
- Programmierung von SPS
- Hardwareaufbau und Deployment

Agile Entwicklung

- Komplexe Anwendungsdomäne
- Akzeptanz bei den Anwendern
- Bessere Kommunikation mit den Domänenexperten

Warum Xtext?

Warum Xtext?

- Top-Down-Ansatz verträgt sich gut mit MDE

Warum Xtext?

- Top-Down-Ansatz verträgt sich gut mit MDE
- Agile Grammatik-Entwicklung

Warum Xtext?

- Top-Down-Ansatz verträgt sich gut mit MDE
- Agile Grammatik-Entwicklung
- Grammatik ist auch Metamodellbeschreibung

Warum Xtext?

- Top-Down-Ansatz verträgt sich gut mit MDE
- Agile Grammatik-Entwicklung
- Grammatik ist auch Metamodellbeschreibung
- Einbettung in eine Tool-Landschaft

Automatische Ecore-Modellerstellung

generate topoDSL "http://www.menges.de/topologie/TopoDSL"

Xtext Features

Automatische Ecore-Modellerstellung

generate topoDSL "http://www.menges.de/topologie/TopoDSL"

Kombination mit separat erstellen Ecore-Modellen

import "platform:/resource/Feldelemente.ecore" **as** feldelemente
import "http://www.eclipse.org/emf/2002/Ecore" **as** ecore

Automatische Ecore-Modellerstellung

generate topoDSL "http://www.menges.de/topologie/TopoDSL"

Kombination mit separat erstellen Ecore-Modellen

import "platform:/resource/Feldelemente.ecore" **as** feldelemente
import "http://www.eclipse.org/emf/2002/Ecore" **as** ecore

Abstrakte Regeln um Mehrfachvererbung auszudrücken

```
HauptsignalArt:
    Mehrabschnittssignal | Hauptsignal ;
VorsignalArt:
    Mehrabschnittssignal | Vorsignal ;
```

Xtext Features

Automatische Ecore-Modellerstellung

generate topoDSL "http://www.menges.de/topologie/TopoDSL"

Kombination mit separat erstellen Ecore-Modellen

import "platform:/resource/Feldelemente.ecore" **as** feldelemente
import "http://www.eclipse.org/emf/2002/Ecore" **as** ecore

Abstrakte Regeln um Mehrfachvererbung auszudrücken

HauptsignalArt:
    Mehrabschnittssignal | Hauptsignal ;
VorsignalArt:
    Mehrabschnittssignal | Vorsignal ;

Linking

SignalReference: 'for' signal=[HauptsignalArt] ;

1. Entwicklung eines konzeptionellen Modells (mit Domänenexperten)

1. Entwicklung eines konzeptionellen Modells (mit Domänenexperten)
2. Umsetzung in eine Grammatik und ggf. separates Domänenmodell

Agile DSL-Entwicklung

1. Entwicklung eines konzeptionellen Modells (mit Domänenexperten)
2. Umsetzung in eine Grammatik und ggf. separates Domänenmodell
3. Generierung der Entwicklungsumgebung

1. Entwicklung eines konzeptionellen Modells (mit Domänenexperten)
2. Umsetzung in eine Grammatik und ggf. separates Domänenmodell
3. Generierung der Entwicklungsumgebung
4. Test der Sprache durch die Domänenexperten

Initiale Bootstrappingphase

- Identifizierung der einzelnen Domänen
- Aspekte oder orthogonale Eigenschaften der Domänen identifizieren
- Vorgehensweise der Anwender ermitteln

Technische Methoden

- Aufteilen der Anwendungsdomäne in verschiedene Partitionen
  - \+ Erlaubt kompakte, fokusierte Sprachen
- Nutzung eines separaten Domänenmetamodells
  - \+ Erlaubt größere Freiheiten bei der Sprachentwicklung
  - \+ Bessere Unterstützung von verschiedenen Aufgabe des Domänenmetamodells
  - – Pflege eines separaten Domänenmetamodells und der Transformationen
- Mehrfachvererbung
  - \+ Abbildung orhtogonaler Konzepte
  - \+ Stufenweiser Aufbau von Sprach- und Domänenmetamodell-Eigenschaften

Partitionskopplung

- Referenzierung von anderen Sprachmodellen
- Referenzierung und Subclassing
  - \+ Erlaubt es Klassenattribute voreinander zu verstecken
  - – Kann aber zu statisch sein
- Kopplung über eine separate Verabredung außerhalb der Modelle

Zusammenfassung

+ Schnelle Sprachentwicklung
+ Geringer Lernaufwand erforderlich
+ Gute Toolunterstützung
– Recht unflexible Kopplungsmethode von fremden Metamodellen
– Definition der Semantik erfolgt noch händisch

Teil II

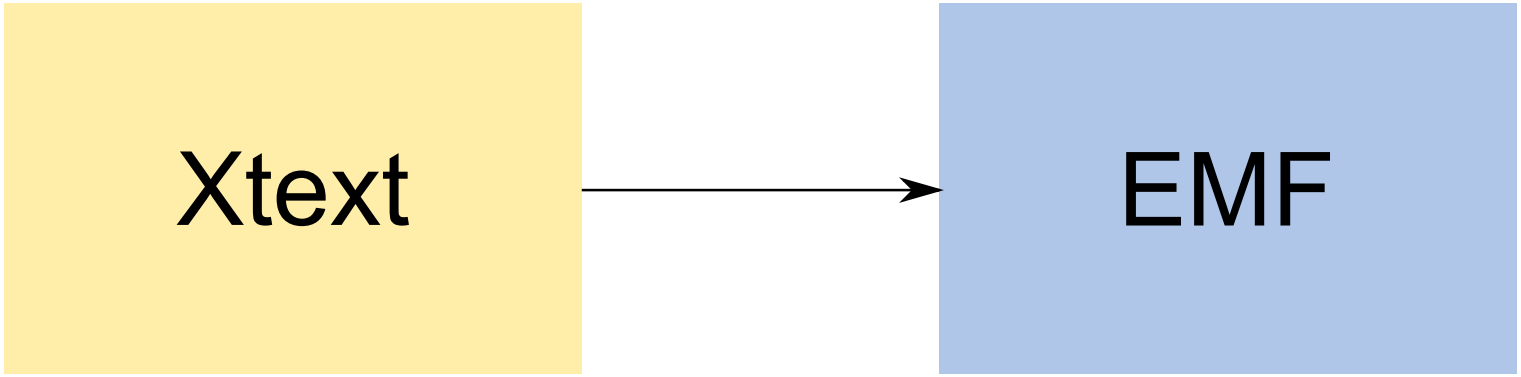
Ausblick auf Xbase

Robert von Massow

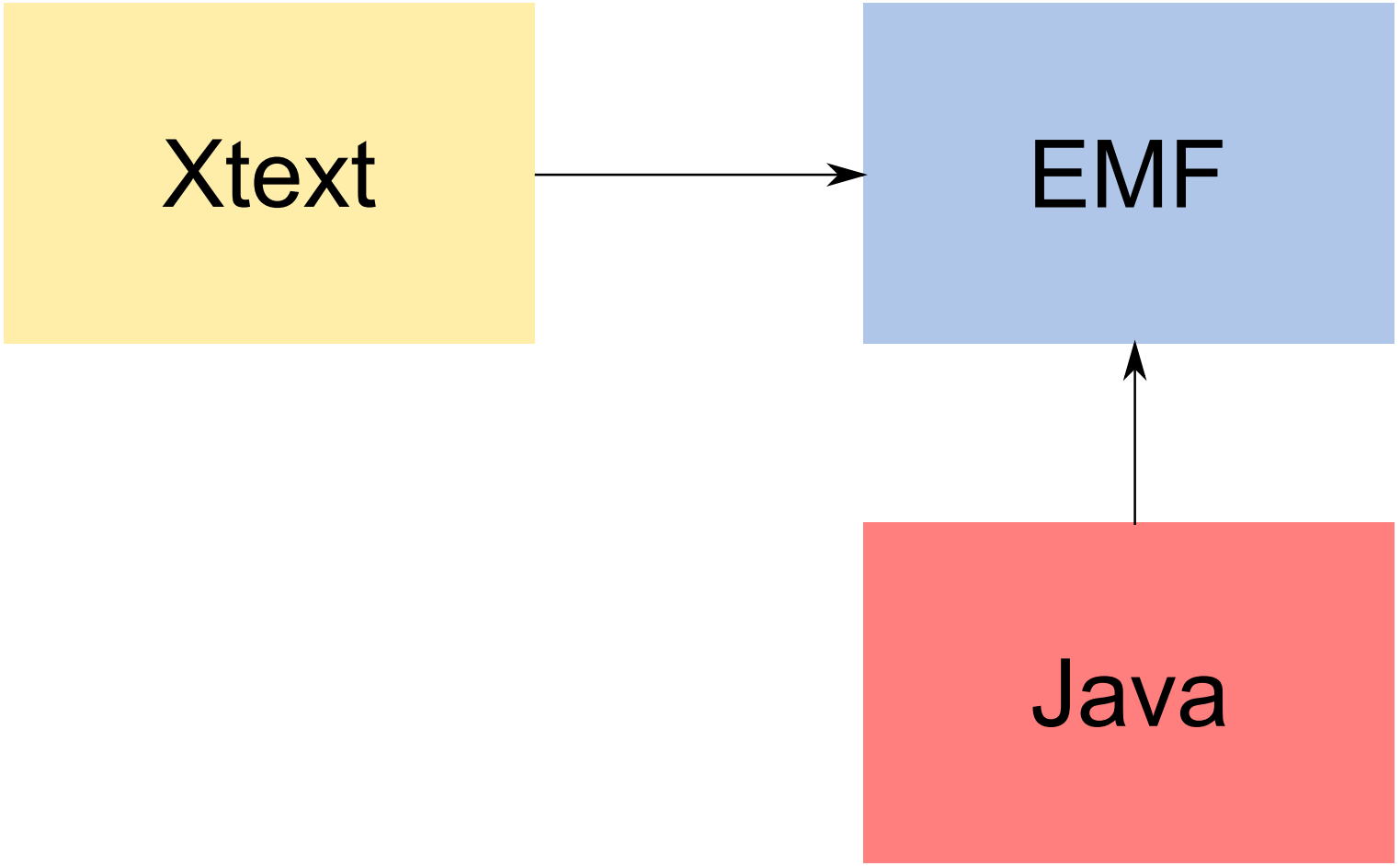
What is Xbase?

Xtext

What is Xbase?

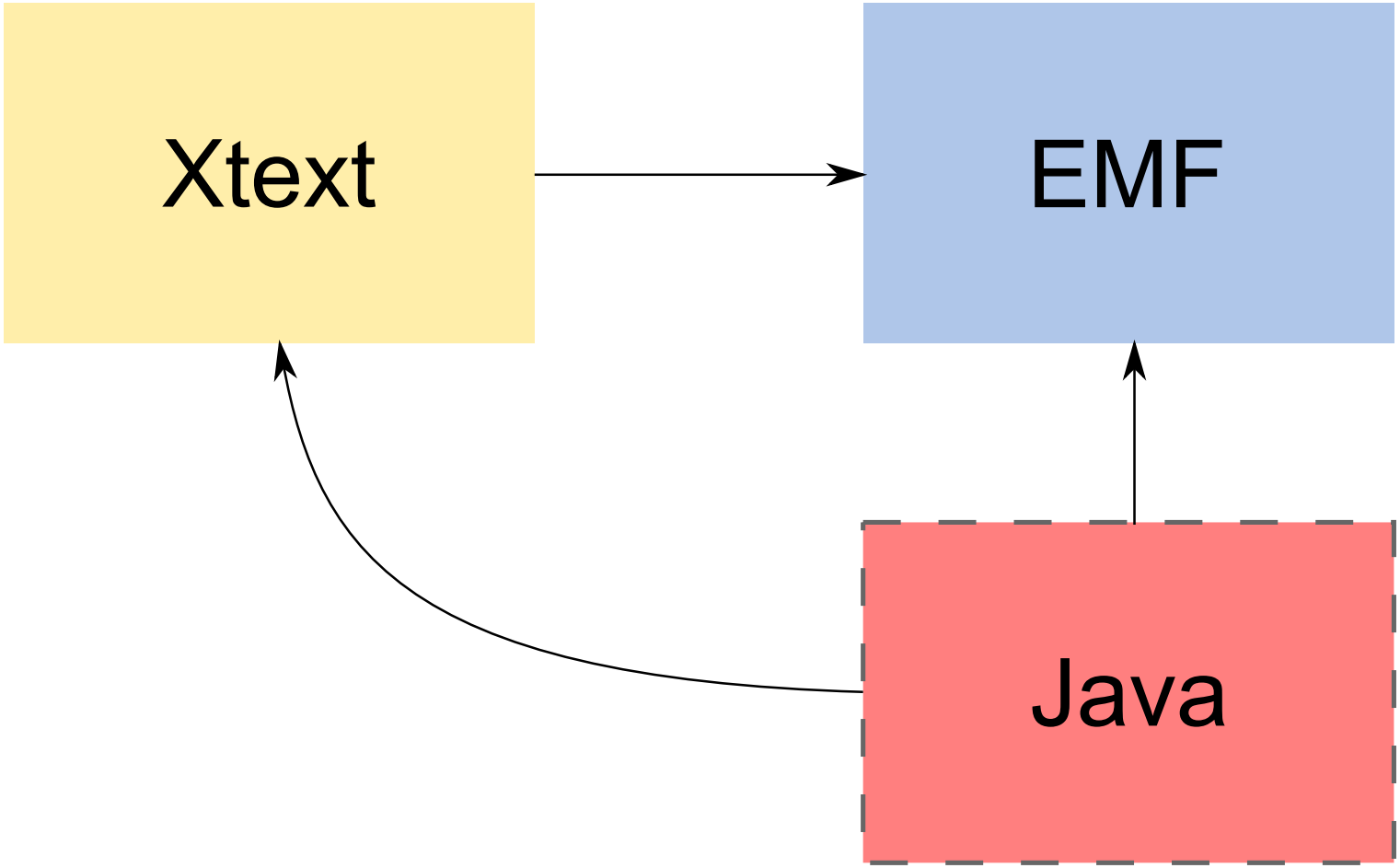


What is Xbase?

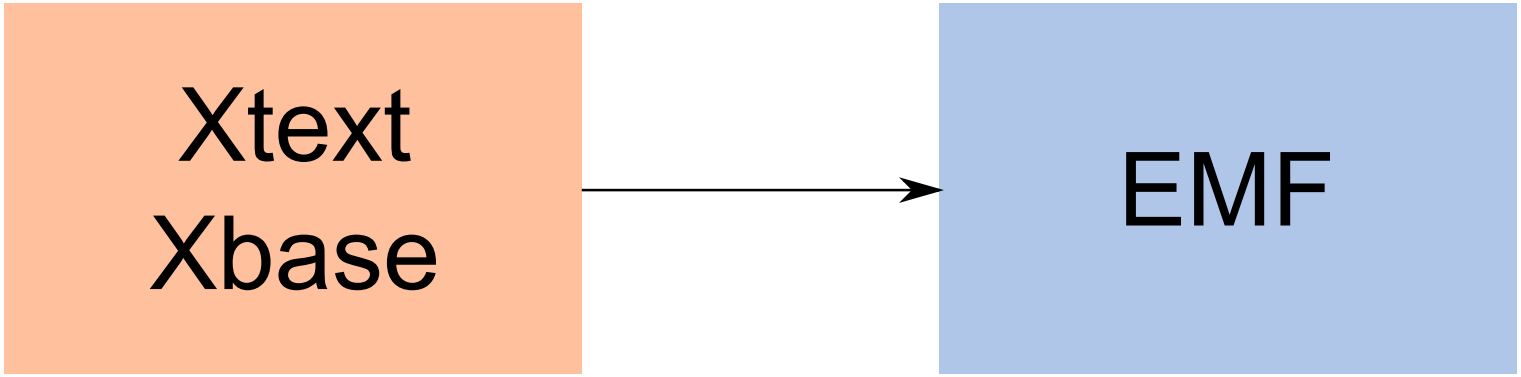


What is Xbase?

Christian-Albrechts-Universität zu Kiel

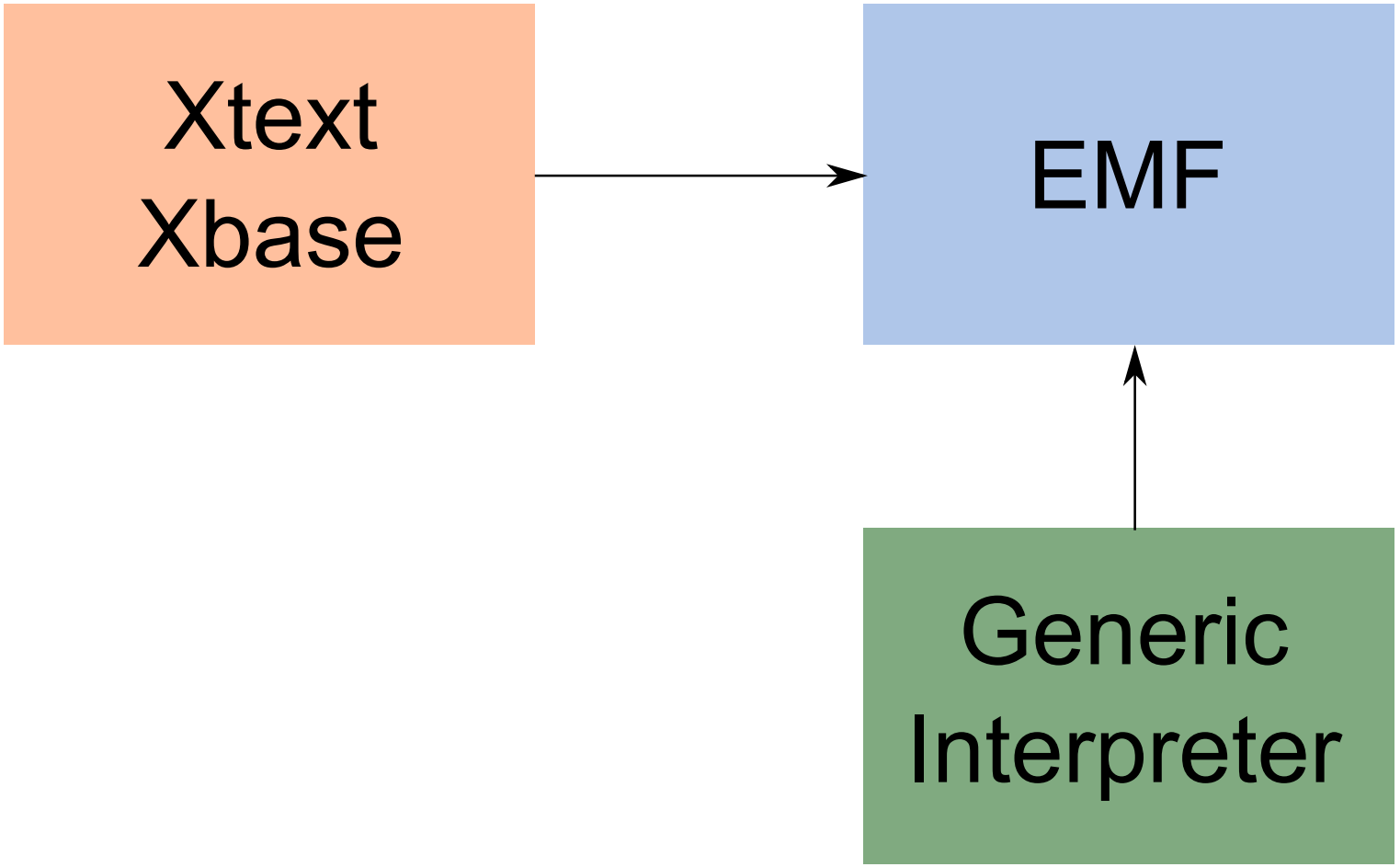


What is Xbase?



What is Xbase?

Christian-Albrechts-Universität zu Kiel



What is Xbase?

Xbase is

- a simple generic expression language (partial programming language)
- embeddable in Xtext
- code generator (compiler) and interpreter

What is Xbase?

Xbase is

- a simple generic expression language (partial programming language)
- embeddable in Xtext
- code generator (compiler) and interpreter

Use Xbase if

- you want to specify computation semantics in your DSL
- want to calculate derived values
- you need complex types

Use Xbase as mix-in instead of terminals:

grammar org.example.MyGrammar **with** org.eclipse.xtext.xbase.Xbase

How to use Xbase

Use Xbase as mix-in instead of terminals:

grammar org.example.MyGrammar **with** org.eclipse.xtext.xbase.Xbase

Import needed meta models:

import "http://www.eclipse.org/xtext/xbase/Xbase"
import "http://www.eclipse.org/xtext/common/JavaVMTypes"

Use Xbase as mix-in instead of terminals:

grammar org.example.MyGrammar **with** org.eclipse.xtext.xbase.Xbase

Import needed meta models:

import "http://www.eclipse.org/xtext/xbase/Xbase"
import "http://www.eclipse.org/xtext/common/JavaVMTypes"

Now you can define rules like this:

```
Process:
      'process' name=ID '{'
'args':' args+=[Arg|FQN] (',' args+=[XLiteral|FQN])
            'do' ':'calc = XExpression
      '}'
;
```

How do Expressions look like


```
Process:
    'process' name=ID '{'
        'args'':' args+=[Arg|FQN] (',' ↩
            args+=[XLiteral|FQN])
        'do' ':'calc = XExpression
    '}'
;
```

How do Expressions look like


```
Process:
    'process' name=ID '{'
        'args'':' args+=[Arg|FQN] (',' ↩
            args+=[XLiteral|FQN])
        'do' ':'calc = XExpression
    '}'
;
```

```
process {
    args: s1, s2
    do: s1, s2 | if(s1 == s2) {s1;} else {↩
        s2;};
}
```

How do Expressions look like


```
Process:
    'process' name=ID '{'
        'args'':' args+=[Arg|FQN] (',' ↩
            args+=[XLiteral|FQN])
        'do' ':'calc = XExpression
    '}'
;
```

```
process {
    args: s1, s2
    do: s1, s2 | if(s1 == s2) {s1;} else {↩
        s2;};
}
```

The syntax is close to Java, but

- there are no statements in Xbase
- there are closures
- there are more powerful switch blocks
- there is type inference for variable declarations

Features of Xbase compared to Java

Java

Xbase

Features of Xbase compared to Java

Java

Java uses checked exceptions

Xbase

Xbase does not force to handle exceptions

Features of Xbase compared to Java

Java

Java uses checked exceptions

Java provides built-in types

Xbase

Xbase does not force to handle exceptions

Xbase is pure OO, no built-in types

Features of Xbase compared to Java

Java

Java uses checked exceptions

Java provides built-in types

Java does not allow operator overloading

Xbase

Xbase does not force to handle exceptions

Xbase is pure OO, no built-in types

Xbase allows for operator overloading

Features of Xbase compared to Java

Java	Xbase
Java uses checked exceptions	Xbase does not force to handle exceptions
Java provides built-in types	Xbase is pure OO, no built-in types
Java does not allow operator overloading	Xbase allows for operator overloading
Java needs explicit typing for variables	Xbase uses type inference

You want to know more?

Visit Sven Efftinge's blog (http://blog.efftinge.de/)