

## Introduction à Xtend

On passe maintenant à une nouvelle section consacrée à Xtend, un langage que vous avez déjà entraperçu dans les slides précédentes à travers des exemples de transformations et de génération de code.

Xtend, c'est quoi ?

Xtend est un langage généraliste qui a été conçu pour être simple à écrire, agréable à lire, et surtout : profondément intégré à l'écosystème Java et EMF.

Il reprend les principes des langages modernes (inférence de types, lambdas, expressions concises), mais il est surtout un produit direct des techniques de MDE :

- Il est défini via une grammaire Xtext,
- Il repose sur un métamodèle EMF,
- Il utilise des transformations modèles pour générer du code Java.

Pourquoi on l'étudie ici ? Parce qu'Xtend est un cas d'étude parfait

:

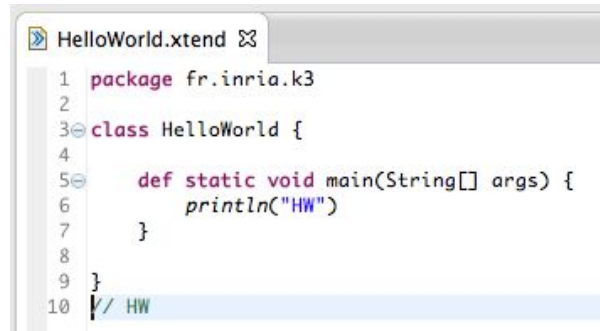
- Il illustre ce qu'on peut construire avec Xtext + EMF,
- Il montre comment DSL et langage généraliste peuvent se rencontrer,
- Et c'est un excellent langage pour manipuler des modèles, que ce soit pour des interprètes, des compilateurs, ou des générateurs.

Bref :

Xtend est à la fois un langage de programmation et un démonstrateur des technologies que vous venez d'apprendre.

On va donc s'y intéresser de plus près.

# Hello World



```
1 package fr.inria.k3
2
3 class HelloWorld {
4
5     def static void main(String[] args) {
6         println("HW")
7     }
8
9 }
10 // HW
```

Avant d'entrer dans les détails techniques, on commence comme toujours... par un Hello World.

Ce que vous voyez ici, c'est Eclipse avec un fichier `.xtend` ouvert.

Il s'agit d'un fichier Xtend classique, contenant une classe `HelloWorld` dans le package `fr.inria.k3`.

À l'intérieur, une méthode `main`, exactement comme en Java, mais écrite dans la syntaxe Xtend.

Quelques remarques :

- Le mot-clé `def` est utilisé à la place de `public static`, pour déclarer une méthode.
- L'appel à `println("HW")` est très naturel, sans besoin de préciser `System.out`, comme en Java.
- Xtend s'intègre directement dans Eclipse, avec auto-complétion, validation, navigation, etc.

Ce petit exemple montre déjà deux choses importantes :

1. Xtend ressemble à Java, mais avec une syntaxe plus légère.

1. Xtend est un langage compilé : ce fichier `.xtend` sera compilé automatiquement en un `.java` équivalent.

Ce Hello World va nous servir de point de départ pour explorer les fonctionnalités puissantes de Xtend :

- transformations de modèles,
- génération de code,
- et des mécanismes avancés comme les expressions de template ou les annotations actives.

## Semi-colon is optional (within class, methods, etc.)

```
1 package fr.inria.k3.fields
2
3 class MyFielder {
4
5     int count = 1
6     static boolean debug = false
7     var name = 'Foo' // type String is inferred
8     val UNIVERSAL_ANSWER = 42 // final field with inferred type int
9     // ...
10    public int count2 = 2 ;
11
12 }
13
```

On va faire un tour d'horizon de la syntaxe de xtext.

Tout d'abord, vous verrez qu'il y a pas mal de sucre dans la syntaxe the xtext, a commencé par le fait que les points-virgules sont optionnels.

# Package Declaration

```
HelloWorld.xtend ⌵
1 package fr.inria.k3
2
3 class HelloWorld {
4
5     def static void main(String[] args) {
6         println("HW")
7     }
8 }
9
10 // HW
```

Semi-colon ';' is optional

'^' for avoiding  
keyword conflicts

```
HelloWorld.xtend HelloWorld.java PackageExploder.xtend ⌵
1 package fr.inria.k3.^def
2
3 class PackageExploder {
4
5 }
```

Comme pour java, xtext est organisé en package, définit tout en haut d'un fichier.

Il existe un moyen d'échappé les mot-clés si on veut les utiliser, ici comme pour "def"

# Methods

**By default:  
visibility  
conditions set  
to public**

```
1 package fr.inria.k3.methods
2
3 class FooMethod {
4
5     def foo1() {
6         "A"
7     }
8
9     def foo2() {
10        6 + 3
11    }
12
13     def private foo3() {
14        6 + 3
15    }
16
17     def public foo4() {
18        foo3() * 8
19    }
20 }
21
22 class FooMethodUses {
23
24     def fooUse() {
25         new FooMethod().foo1
26         new FooMethod().foo3()
27
28         new FooMethod().foo4
29     }
30
31     // foo1 : String = FooMethod.foo1()
32     // foo2 : int = FooMethod.foo2()
33     // foo4 : int = FooMethod.foo4()
34 }
35 }
```

```
1 package fr.inria.k3
2
3 class HelloWorld {
4
5     def static void main(String[] args) {
6         println("HW")
7     }
8 }
9
10 // HW
```

On commence par une structure fondamentale : la déclaration de méthodes.

Xtend réutilise la grammaire de Java mais la simplifie considérablement. On introduit une méthode avec le mot-clé **def**, suivi :

- du nom de la méthode,
- de la liste des paramètres typés entre parenthèses,
- éventuellement du type de retour
- et d'un corps entre accolades.

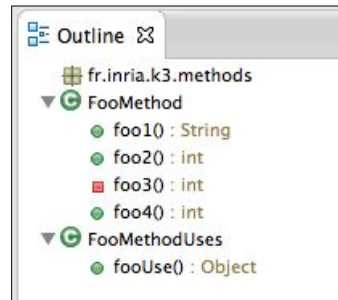
Le mot-clé **return** est souvent inutile : Xtend retourne automatiquement la dernière expression.

Les méthodes peuvent être déclarées dans des classes Xtend, ou directement dans des fichiers utilitaires.

# Methods

```
1 package fr.inria.k3.methods
2
3 class FooMethod {
4
5     def foo1() {
6         "A"
7     }
8
9     def foo2() {
10         6 + 3
11     }
12
13     def private foo3() {
14         6 + 3
15     }
16
17     def public foo4() {
18         foo3() * 8
19     }
20 }
21
22 class FooMethodUses {
23
24     def fooUse() {
25         new FooMethod().foo1
26         new FooMethod().foo3()
27
28         new FooMethod().foo4()
29     }
30
31     def fooUse2() {
32         new FooMethod().foo2
33         new FooMethod().foo4
34     }
35 }
```

## Type inference (return type)



202

hallita

33

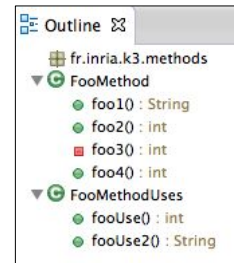
Le typage est obligatoire sur les paramètres, mais facultatif sur le retour si inféré.



# Method Calling

You can omit parentheses

```
33 def fooUse2() {  
34     3.toString  
35     "4".length  
36     new FooMethod().foo1  
37     new FooMethod().foo1()  
38     new FooMethod().foo1 + 3.toString  
39  
40 }
```



En Xtend, appeler une méthode fonctionne exactement comme en Java :

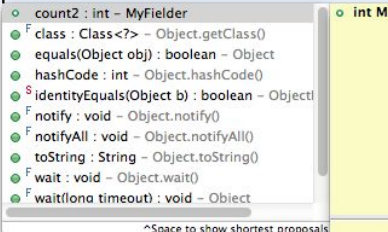
on écrit simplement `objet.methode(paramètres)`.

La différence, c'est que Xtend ajoute :

- des raccourcis syntaxiques,
- des expressions plus concises,
- et une prise en charge élégante des fonctions anonymes et lambda.

# Fields

```
1 package fr.inria.k3.fields
2
3 class MyFielder {
4
5     int count = 1
6     static boolean debug = false
7     var name = 'Foo' // type String is inferred
8     val UNIVERSAL_ANSWER = 42 // final field with inferred type int
9     // ...
10    public int count2 = 2 ;
11
12 }
13
14 class MyAccessor {
15
16     def foo() {
17         new MyFielder().
18     }
19 }
20
```



By default:  
visibility  
conditions set  
to private

2025-2026

Stephanie Chailita

35

Xtend permet de déclarer des champs d'instance de manière très simple, tout en étant interopérable avec Java.

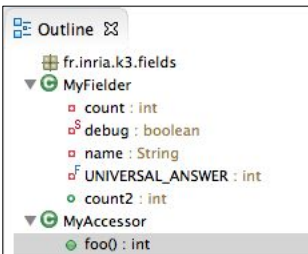
Le point important, c'est que Xtend génère automatiquement le code Java associé :

- les champs sont privés,
- les getters et setters sont générés selon les conventions JavaBeans,
- vous pouvez y accéder directement via `objet.champ` : c'est traduit correctement en Java.

Xtend infère le type si on l'omet, tant que l'initialisation est présente.

# Fields

```
1 package fr.inria.k3.fields
2
3 class MyFielder {
4
5     int count = 1
6     static boolean debug = false
7     var name = 'Foo' // type String is inferred
8     val UNIVERSAL_ANSWER = 42 // final field with inferred type int
9     // ...
10    public int count2 = 2 ;
11
12 }
13
```



primitive types of Java (int, boolean, etc) with autoboxing

var: type inference

val: constant, « final » in Java

2025-2026

hallita

36

On utilise :

- **val** pour un champ immuable — on ne pourra pas le modifier après initialisation.
- **var** pour un champ modifiable.

# Static Methods (::)

```
1 package fr.inria.k3.stat
2
3 import java.util.Collections
4
5 class FooStati {
6
7
8     static var colors = newArrayList(46, 76, 89, 53)
9
10
11     def static void main(String... args) {
12         println("B " + colors)
13         Collections::sort(colors)
14         println("A " + colors)
15
16         colors.add(45)
17         println("A " + colors)
18     }
19 }
20 }
```

```
B [46, 76, 89, 53]
A [46, 53, 76, 89]
A [46, 53, 76, 89, 45]
```

2025-2026

37

Xtend introduit une syntaxe très pratique pour travailler avec les méthodes statiques, grâce à l'opérateur `::`.

Avec `NomDeClasse::nomDeMéthode`, on peut :

- appeler directement une méthode statique,
- ou référencer cette méthode comme une fonction, par exemple pour la passer à une lambda.

Ici, `Collections::sort(colors)` est une référence de méthode utilisée comme callback dans une boucle.

Cette syntaxe permet une meilleure intégration avec les API fonctionnelles de Java, tout en rendant les appels plus concis et lisibles.

# Pairs

```
1 package fr.inria.k3.pairs
2
3 class FooSugar {
4
5     // syntactic sugar
6     val pair = "spain" -> "italy"
7     var k = pair.key
8     var v = pair.value
9
10    def foo() {
11        println("key=" + k + " value=" + v)
12    }
13
14 }
15
```

fr.inria.k3.pairs

- FooSugar
  - pair : Pair<String, String>
  - k : String
  - v : String
  - foo() : String

2025-2026

Stéphanie Challita

61

Xtend propose un opérateur très simple pour créer des paires clé-valeur, c'est l'opérateur `->`.

Par exemple :

```
val pair = "spain" -> "italy"
```

Ici, on construit un objet de type `Pair<String, String>`.

Ces paires sont particulièrement utiles :

- pour construire des structures associatives,
- pour mapper des correspondances entre modèles,
- ou comme valeurs de retour intermédiaires dans des transformations.

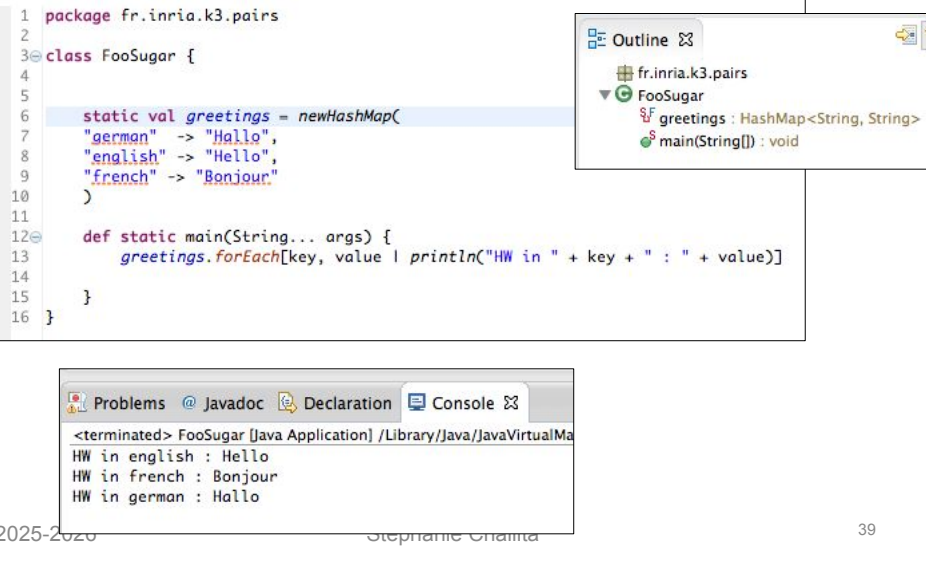
Une fois la paire créée, on peut accéder à ses éléments avec `.key` et `.value`, comme dans :

```
println("key=" + k + " value=" + v)
```

Ce mécanisme est très utilisé avec `map`, `groupBy`, ou `associate`,

et permet de structurer proprement des données simples, sans devoir créer une classe dédiée à chaque fois.

# Pairs



```
1 package fr.inria.k3.pairs
2
3 class FooSugar {
4
5
6     static val greetings = newHashMap(
7         "german" -> "Hallo",
8         "english" -> "Hello",
9         "french" -> "Bonjour"
10    )
11
12    def static main(String... args) {
13        greetings.foreach[key, value | println("HW in " + key + " : " + value)]
14    }
15 }
16 }
```

Outline

- fr.inria.k3.pairs
  - FooSugar
    - greetings : HashMap<String, String>
    - main(String[]) : void

Problems Javadoc Declaration Console

<terminated> FooSugar [Java Application] /Library/Java/JavaVirtualMa

```
HW in english : Hello
HW in french : Bonjour
HW in german : Hallo
```

2025-2026 Stephanie Chaintre

Ici, on voit un exemple complet et concret d'utilisation des **Pair** avec **foreach**.

On déclare une variable statique appelée **greetings**, qui contient une liste de paires langue → salutation.

On utilise ensuite **.foreach** pour parcourir cette liste, et on affiche chaque salutation avec sa langue associée.

Ce qu'on voit ici :

- L'opérateur **->** permet de construire les paires très facilement.
- **.foreach** accepte une lambda implicite, ici nommée **g**, qui représente chaque **Pair<String, String>**.
- On peut accéder aux champs **.key** et **.value** pour manipuler le contenu.

Ce genre de structure est très pratique en Xtend pour stocker des correspondances temporaires, configurer des générateurs de code, ou structurer de petits DSLs.

# Immutable data structure

```
1 package fr.inria.k3.stat
2
3 import java.util.Collections
4
5 class FooStati {
6
7
8     static var colors = #[46, 76, 89, 53] // newArrayList(46, 76, 89, 53)
9
10
11     def static void main(String... args) {
12         println("B " + colors)
13         Collections.sort(colors)
14         println("A " + colors)
15
16         colors.add(45)
17         println("A " + colors)
18
19     }
20 }
```

```
<terminated> FooStati [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_13.jdk/Contents/
B [46, 76, 89, 53]
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableList.set(Collections.java:1244)
    at java.util.Collections.sort(Collections.java:159)
    at fr.inria.k3.stat.FooStati.main(FooStati.java:15)
```

2025-2026

64

Xtend encourage l'utilisation de structures immuables, par défaut.  
Dans cet exemple, on définit une variable statique `colors`, une liste de nombres entiers, avec la syntaxe `#[ ]`.

Cette syntaxe crée une liste immuable :  
on ne pourra pas la modifier après sa création.

Dans `main`, on affiche la liste avec `println(colors)`,  
puis on appelle `colors.sort` — mais attention :  
Cela retourne une nouvelle liste triée,  
Sans modifier la liste originale.

On peut le constater car le deuxième `println(colors)` affiche  
toujours la version non triée.

Si on essaie d'appeler `colors.add(100)`,  
on obtient une erreur de compilation :

Cannot invoke add on List<Integer> because it is  
read-only.



- que les données ne changent pas de manière inattendue,
- que le code reste simple et prévisible,
- et que les structures sont plus sûres (notamment en multithreading).

# Constructor

Default visibility: public

```
1 package fr.inria.k3.classes
2
3 class FooConstructor {
4
5     var String l
6
7     new() {
8         this("F00")
9     }
10
11     new (String v) {
12         l = v
13     }
14
15
16     override toString() {
17         l
18     }
19
20     def static void main (String... args) {
21         println("1 " + new FooConstructor())
22         println("2 " + new FooConstructor("F00 2"))
23     }
24 }
```

Problems @ Javadoc Declaration Console

<terminated> FooConstructor [Java Application] /Library/Java/JavaVirt

1 F00  
2 F00 2

override keyword:  
mandatory

2025-2026

Stephanie Chailita

41

Xtend reprend la structure de Java, mais avec une syntaxe beaucoup plus simple.

Pour définir un constructeur, on utilise le mot-clé **new**, suivi des paramètres et du bloc d'initialisation.

Dans l'exemple :

- La classe **FooConstructor** a deux constructeurs: sans paramètre et avec 1 parametre
- Le constructeur affecte les valeurs reçues aux champs à l'aide de **this.nomChamp**.

Points clés à retenir :

- Visibilité par défaut : publique. On ne met donc pas **public** devant le constructeur ou la classe.
- Le mot-clé **override** est obligatoire lorsqu'on redéfinit une méthode héritée comme **toString**.

Lorsqu'on appelle **println(foo)**, Xtend utilise automatiquement **toString()**, qu'on a redéfini ici de façon plus expressive.

# Cast and Type

```
1 package fr.inria.k3.types
2
3 class FooTypes {
4
5     static val Object obj = "a string"
6     // static val String s = obj
7     static val String s = obj as String // cast
8
9     def static void main(String... args) {
10         println(typeof(String) + "") // String.class
11         println("\t" + s + "\n")
12     }
13 }
14 }
```

Ici, on illustre deux fonctionnalités typiques autour des types en Xtend :

le cast explicite et l'introspection de type.

On part d'un objet `obj` de type `Object`, initialisé avec une chaîne. Ensuite, on effectue un cast explicite avec `as String`. C'est l'équivalent de `(String)obj` en Java, mais avec une syntaxe plus lisible.

Dans `main`, on utilise `typeof(String)` pour obtenir une représentation du type `String`.

Cela permet d'interroger ou de générer dynamiquement du code en fonction de types.

Xtend reste fortement typé, mais autorise ce type de manipulation lorsqu'on a besoin de flexibilité, notamment pour :

- interagir avec du code Java externe,
- manipuler des modèles hétérogènes (ex : EMF),
- ou écrire des générateurs plus dynamiques.

# Extension Methods...

« ... allow to add new methods to existing types without modifying them. »

```
def removeVowels (String s){  
  s.replaceAll("[aeiouAEIOU]", "")  
}
```

We can call this method either like in Java:

```
removeVowels("Hello")
```

or as an extension method of String:

```
"Hello".removeVowels
```

**The first parameter of a method can either be passed in after opening the parentheses or before the method call**

2025-2026

Stéphanie Challita

68

Xtend propose une fonctionnalité très puissante et pratique : les extension methods.

Le principe est simple :

vous définissez une méthode "classique", comme la méthode "removeVowels".

Mais ensuite, vous pouvez l'appeler comme si elle appartenait à **String** directement

C'est ce qu'on appelle "promouvoir" le premier paramètre en récepteur :

la chaîne **"Hello"** devient l'objet sur lequel on appelle **.removeVowels**.

C'est extrêmement utile pour :

- enrichir les types standards (comme String, List, Map),
- écrire du code fluide, lisible, proche du langage naturel,
- éviter d'avoir des classes utilitaires pleines de méthodes statiques.

Et bien sûr, tout ça est fortement typé, compilé, et compatible avec

Java.

# Lambda Expression

No need to  
specify the  
type for e

```
1. textField.addActionListener([ e |  
2.     textField.text = "Something happened!"  
3. ])
```

You can even  
omit e

```
1. textField.addActionListener(  
2.     textField.text = "Something happened!"  
3. ])
```

Xtend propose une syntaxe très fluide et concise pour écrire des lambdas — c'est-à-dire des fonctions anonymes.

Dans l'exemple :

La variable `e` représente l'événement reçu, mais on n'a pas besoin de déclarer son type. Xtend l'infère automatiquement à partir du contexte (`ActionListener` → `ActionEvent`).

Encore mieux :

si vous n'utilisez même pas `e` dans le corps, vous pouvez l'omettre complètement :

Cela rend le code beaucoup plus expressif et lisible, notamment pour :

- les callbacks (`onClick`, `onChange`, etc.),
- les opérations sur collections (`map`, `filter`, `forEach`),
- ou tout usage fonctionnel dans les API Java modernes.

En résumé :

Xtend vous permet d'écrire du code fonctionnel propre, concis, et interopérable avec Java — sans sacrifier la clarté

# Lambda Expression

```
1. Collections.sort(someStrings) [ a, b |  
2.   a.length - b.length  
3. ]
```

```
Java 8: shapes.forEach(s -> { s.setColor(RED); });
```

```
Xtend: shapes.forEach[color = RED]
```

```
Java 8: shapes.stream()  
        .filter(s -> s.getColor() == BLUE)  
        .forEach(s -> { s.setColor(RED); });
```

```
Xtend: shapes.stream  
        .filter[color == BLUE]  
        .forEach[color = RED]
```

2025-2026

71

Ici, on voit à quel point Xtend excelle dans l'écriture fonctionnelle, avec une syntaxe plus concise et lisible que Java 8, tout en restant compatible.

Premier exemple : tri personnalisé avec **sort**.

En Java, cela prendrait plusieurs lignes avec un **Comparator**.

En Xtend : La lambda remplace directement l'objet fonctionnel attendu. Ensuite, comparons deux syntaxes **forEach** :

Ensuite, comparons deux syntaxes **forEach** :

En Java, on passe une lambda complète avec **s -> ....**

En Xtend, c'est : Le paramètre **s** est implicite, et la syntaxe est ultra-lisible.

Même chose pour une chaîne d'opérations fonctionnelles :

Ici, chaque opération est expressive, sans répétition, sans verbosité.

En résumé :

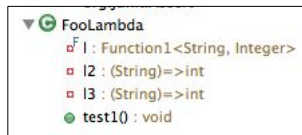
- Xtend rend la programmation fonctionnelle fluide,
- Et permet d'exprimer des transformations complexes avec

- très peu de code.



# Lambda Expression

```
class FooLambda {  
    val l = [String s | s.length]  
    var (String)=>int l2 = [it.length] // it is a keyword for referring to the first parameter  
    var (String)=>int l3 = [length] // we can even omit it or the first parameter  
  
    @Test  
    def test1() {  
        assertEquals(l.apply("RRRR"), l2.apply("PPPP"))  
        assertEquals(l2.apply("RRRR"), l3.apply("PPPP"))  
    }  
}
```



2025-2026

Stéphanie Challita

46

Ici, on résume de manière très claire **la souplesse de Xtend pour définir des lambdas**, avec trois styles équivalents.

Ligne 1 : syntaxe Java-like, très explicite. On précise le type et on donne un nom de paramètre :

Ligne 2 : plus idiomatique en Xtend. On utilise **it**, le paramètre implicite

Ligne 3 : encore plus concis ! On **omet complètement le paramètre**, quand c'est possible :

Dans tous les cas, ces trois lambdas sont **de type Function1<String, Integer>**, comme on le voit dans la vue de droite.

Le test suivant vérifie que les trois notations sont **équivalentes en comportement** :

À retenir :

- Xtend **autorise plusieurs niveaux de concision**
- On peut écrire des lambdas lisibles, explicites, ou extrêmement compactes
- Très utile pour la **programmation fonctionnelle** dans des contextes Java-like

# Templates

```
1 package fr.inria.k3.templates
2
3 import org.junit.Test
4 import static org.junit.Assert.*
5
6 class FooTempl {
7
8
9     def someHTML(String content) '''<html><body>«content»</body></html>'''
10
11
12     @Test
13     def test1() {
14         assertEquals("<html><body>HW</body></html>", someHTML('HW').toString)
15     }
16
17 }
```

Xtend propose un mécanisme très puissant et intuitif : les template expressions

Dans cet exemple :

On utilise des triple quotes `'''` pour délimiter une chaîne multilignes (comme en Python).

À l'intérieur, on peut directement insérer des variables ou expressions Xtend via les guillemets doubles `"..."`.

Ici, on insère la variable `content` dynamiquement dans une structure HTML.

Le test suivant montre comment on peut générer dynamiquement du HTML.

Cela permet de faire du Model-to-Text très simplement, sans outil supplémentaire.

À retenir :

- Le moteur de template de Xtend est intégré au langage
- Il est type-safe : si on insère une variable mal typée, la

- compilation échoue
- C'est extrêmement pratique pour générer des fichiers (HTML, Java, SQL, etc.)

## Templates (2)

```
@Test
def test2() {
  // loading
  var pollS = loadPollSystem(URI.createURI("foo1.q"))

  // MODEL MANAGEMENT (ANALYSIS, TRANSFORMATION)
  var html = toPolls(pollS.polls)
  assertNotNull(html)

  // serializing (note: we could type check the HTML
  // with Xtext by specifying the grammar for instance)
  val fw = new FileWriter("foo1.html")
  fw.write(html.toString())
  fw.close()
}

def toPolls(List<Poll> polls) '''
  <html>
  <body>
    «FOR p : polls»
    «IF p.name != null»
    <h1><p.name></h1>
    «ENDIF»
    «FOR q : p.questions»
    <p>
      <h2><q.text></h2>
      <ul>
        «FOR o : q.options»
        <li><o.text></li>
        «ENDFOR»
      </ul>
    </p>
    «ENDFOR»
  </body>
</html>
'''
```

platform/resource/org.xtext.e

- questionnaire
- PollSystem
  - polls : Poll
- Poll
  - name : EString
  - questions : Question
- Question
  - text : EString
  - options : Option
- Option
  - id : EString
  - text : EString

**poll1**

**What is A ?**

- B
- C
- D

**poll2**

**What is D ?**

- E
- F

foo1.q.xt

```
PollSystem {
  Poll poll1 {
    Question A {
      "What is A ?"
      options
        b : "B"
        c : "C"
        d : "D"
    }
  }
  Poll poll2 {
    Question D {
      "What is D ?"
      options
        e : "E"
        f : "F"
    }
  }
}
```

2025-2026 Stéphanie Challita 48

Maintenant que vous avez vu la syntaxe de base des templates en Xtend, regardons un exemple plus complet et réaliste de Model-to-Text transformation, appliqué à notre système de sondage.

En haut à gauche, on retrouve notre fonction de test `test2()`.

Elle commence par charger un modèle, comme on l'a déjà vu avec `loadPollSystem`.

Ensuite, on transforme ce modèle en HTML avec la fonction `toPolls`, et on vérifie que le résultat n'est pas nul.

Puis, ce HTML est sérialisé dans un fichier avec un `FileWriter`.

Notez le commentaire important : ici, on écrit une chaîne de caractères HTML à la main, mais on pourrait aller plus loin.

Par exemple, on pourrait définir une grammaire HTML dans Xtext, ce qui nous permettrait de vérifier la validité de notre HTML à la compilation, comme on le ferait avec un modèle.

Dans la fonction `toPolls`, plus bas, vous voyez un exemple typique de template Xtend :

- On utilise des balises HTML standards,
- On intègre des blocs dynamiques grâce aux instructions **FOR**, **IF**, **END FOR**, etc.
- Et on injecte les données du modèle (le nom du poll, les questions, les options) directement dans le HTML.

Sur la droite, vous voyez :

- Le fichier **.q** d'entrée (le modèle),
- Et à droite, le résultat HTML rendu, avec une flèche qui illustre bien la transformation.

Ce slide résume très bien comment, en partant d'un modèle métier (ici, un système de sondage), on peut générer automatiquement un artefact exécutable ou visualisable, ici du HTML pour une interface web.

C'est un exemple concret de modèle opérationnalisé, avec un langage DSL spécifique et un générateur simple, maintenable, et entièrement traçable.

## Templates (3)

- You already experiment with web templating engines (JSP, Scala templates in Play!, Symfony templates, etc.)

```
<h1>Exemple de page JSP</h1>
<%-- Impression de variables --%>
<p>Au moment de l'exécution de ce script, nous sommes le <%= date %>.</p>
<p>Cette page a été affichée <%= nombreVisites %> fois !</p>
</body>
</html>
```

- Alternatives exist in the modeling world
  - Multiple predefined and customizable generators



- Xtend: seamless integration into a general purpose language

Ce slide vient relier le monde des DSLs et de la génération de code à des choses que vous avez probablement déjà expérimentées dans d'autres contextes.

Vous avez peut-être déjà utilisé des moteurs de templates pour le Web : comme JSP, les templates Scala dans Play!, ou encore Twig avec Symfony.

Le principe est toujours le même : on écrit du HTML dans lequel on injecte dynamiquement des données issues du code, souvent à partir d'un modèle ou d'un contrôleur.

Eh bien dans le monde de la modélisation et des DSLs, on a des approches similaires.

Il existe plusieurs outils de génération de code qui s'appuient sur des templates :

- Certains sont préconfigurés, mais on peut généralement les personnaliser.
- L'un des plus connus dans l'écosystème Eclipse, c'est Acceleo (vous voyez son logo ici), qui permet de faire de la

- génération de code conforme au standard OMG MOF2T.

Et enfin, comme vous l'avez déjà vu, Xtend offre une alternative très élégante et moderne :

- Vous pouvez écrire des templates directement dans le langage généraliste,
- Avec une intégration transparente à la logique métier,
- Et sans rupture entre le code, le modèle, et les artefacts générés.

En résumé, que vous veniez du monde du Web ou de l'ingénierie dirigée par les modèles, les templates sont une pièce centrale de la génération automatique, et vous avez des outils puissants pour les manipuler.

# Xtend to Java



The screenshot shows an IDE with two tabs: 'HelloWorld.xtend' and 'HelloWorld.java'. The 'HelloWorld.xtend' tab is active, displaying the following code:

```
1 package fr.inria.k3;
2
3 import org.eclipse.xtext.xbase.lib.InputOutput;
4
5 @SuppressWarnings("all")
6 public class HelloWorld {
7     public static void main(final String[] args) {
8         InputOutput.<String>println("HW");
9     }
10 }
11
```

The 'HelloWorld.java' tab is also visible, showing the compiled Java code.

2025-2026

Stéphanie Challita

80

Ce slide illustre un point clé de la philosophie Xtend : Xtend n'est pas un langage indépendant, c'est un langage qui compile vers Java.

À gauche, vous voyez un fichier `HelloWorld.xtend`. C'est le code tel que vous, développeur, l'écrivez.

C'est plus concis, plus lisible, plus moderne que du Java classique.

Et à droite, vous voyez ce que le compilateur Xtend génère : un fichier `.java` entièrement standard.

Ici par exemple, l'appel à `println("HW")` est transformé en un appel explicite via `InputOutput.println(...)`, avec les bons types Java.

Ce qu'il faut bien comprendre ici, c'est que :

- Vous bénéficiez de toute l'outillage Java existant : compilation, débogage, exécution, intégration à Maven/Gradle, etc.
- Mais avec une syntaxe plus expressive.

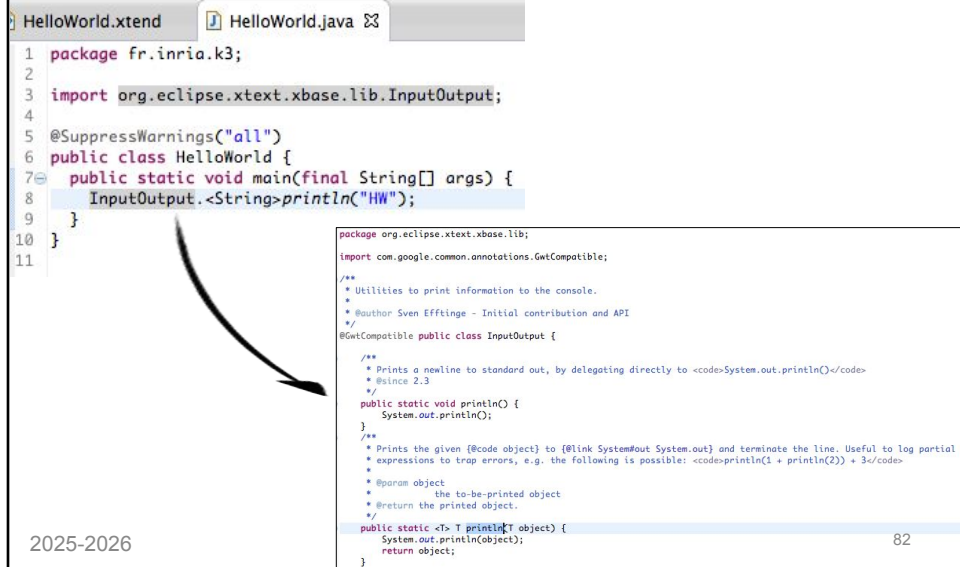


- Et surtout, cela s'intègre très bien avec les outils de modélisation, comme vous l'avez vu avec Xtext.

Autrement dit : Xtend vous donne la puissance de Java, avec la légèreté d'un langage moderne.

# Xtend to Java (2)

more after



Logiquement, on voit ici ce que fait réellement Xtend sous le capot.

On part de notre code très simple en Xtend — une instruction `println("HW")` dans une classe `HelloWorld`.

Et ce que produit Xtend à l'exécution, c'est du Java parfaitement valide, mais en s'appuyant sur une bibliothèque utilitaire : `org.eclipse.xtext.xbase.lib.InputOutput`.

Ici, `InputOutput.println(...)` est une méthode statique générique.

Elle redirige simplement vers `System.out.println(...)`, tout en retournant l'objet passé en paramètre.

Cela permet des appels chaînés, ou des utilisations dans des expressions plus complexes.

Ici, on illustre bien un point important :

Xtend repose sur une librairie d'exécution Java bien définie.

Et surtout :

- Le comportement est déterministe et maîtrisé,
- Et la lisibilité reste maximale côté Xtend.

On commence donc à comprendre le rôle de Xtend comme langage génératif, capable de produire du Java robuste, avec une syntaxe bien plus agréable à écrire.