Xtend2 Language Specification

Sven Efftinge, et al.

November 26, 2010

Contents

1 Preface

This document specifies the language Xtend2. Xtend2 is a programming language implemented in Xtext, based on Xbase, and tightly integrated with Java. It's main purpose is to write compilers, interpreters and other things, where you need to traverse typed tree structures (read EMF models). It integrates and compiles to Java.

Conceptually and syntactically, Xtend2 is a subset of Java, with the following differences:

- First class support for template syntax
- extension methods
- multiple dispatch aka polymorphic method invocation
- built-in support for dependency injection
- inferred return types
- convenient graph navigation expressions
- Uses Xbase expressions, which means :
  - No checked exceptions
  - Pure object-oriented, i.e. no built-in types and no arrays
  - Everything is an expression, there are no statements
  - Closures
  - Type inference
  - Properties
  - Simple operator overloading
  - Powerful switch expressions

2 Language Concepts

On a first glance an Xtend2 file pretty much looks like a Java file. it starts with a package declaration followed by an import section, and after that comes the class definition. That class in fact is directly translated to a Java class in the corresponding Java package. Here is an example:

```
package com.acme;

import java.util.List;

class MyClass {
    String first(List<String> elements) {
        ...
    }
}
```

2.1 Package Declaration

Package declarations are like in Java, with the small difference, that an identifier can be escaped with a ^ in cae it conflicts with a keyword.

2.1.1 Syntax

```
PackageDeclaration : 'package' QualifiedName;
QualifiedName : ID ('.' ID);
```

2.2 Imports

Also the imports are equivalent to the imports known from Java. Again one can escape any names conflicting with keywords using the ^.

2.2.1 Syntax

```
ImportSection : (Import|StaticImport)*;
Import : 'import' 'static'? QualifiedName ('.''*')?;
```

2.3 Class Declaration

Also the class declaration reuses a lot of Java's syntax it is a bit different in some aspects. Firstly the default visibility of any class is public. It is possible to write it explicitly but if not specified it defaults to public. The package private visibility does not exist.

The abstract as well as the final modifiers are directly translated to Java, hence have the exact same meaning.

2.3.1 Inheritance

Also inheritance is directly reused from Java. Xtend2 allows single inheritance of Java classes as well as implementing multiple Java interfaces.

2.3.2 Generics

Full Java Generics are supported.

2.3.3 Syntax

```
ClassDeclaration :
    (Visibility|'abstract'|'final')* 'class' TypeParameters? ('extends' QualifiedName)? ('implements' QualifiedName (',' Qu
        Members*
    '}'
;
Visibility : 'public'|'protected'|'private';
```

2.4 Fields

An Xtend2 class can have state, just declare a field like you are used to in Java.

```
int numberOfCalls = 0;
```

It directly translates to Java. The default visibility for fields is private.

2.4.1 Syntax

```
FieldDef : ('public'|'protected'|'private'|'final')* TypeRef? ID'
    ( '=' Expression)? ';'
```

2.5 Functions

Xtend2 functions are declared within a class and are usually translated to a corresponding Java method with the exact same signature. The only exception is overloaded methods, which compile to a single method. This is explained in subsection 2.5.3.

The default visibility of a function is public, which can also be declared explicitly. The two other available visibilities are protected and private.

A function can be declared either final or abstract. If it is declared abstract also the class needs to be declared abstract and the function is not allowed to have an implementation.

An example of a function declaration

```
Boolean equalsIgnoreCase(String s1,String s2) :
    s1.toLowerCase() == s2.toLowerCase();
```

2.5.1 Inferred Return Types

If the return type of a function can be inferred it does not need to be declared. That is the function

```
Boolean equalsIgnoreCase(String s1,String s2) :
    s1.toLowerCase() == s2.toLowerCase();
```

could be declared like this:

```
equalsIgnoreCase(String s1,String s2) :
    s1.toLowerCase() == s2.toLowerCase();
```

This doesn't work for abstract function declarations as well as if the return type of a function depends on a recursive call of the same function. The compiler tells the user when it needs to be specified.

2.5.2 Generics

Full Java Generics are supported.

2.5.3 Method overloading and polymorphic dispatch

It is possible to overload methods, but overloaded methods are not simply transalted to corresponding Java methods. Instead for each set of methods where the name is equal and the number of arguments is equal the most common dominator signature is taken (or computed if neccessary) and only for that method a Java method is derived. Within the implementation the correct method is looked up at runtime. This is done by sorting the methods from most specific to least specific and generating an if-else cascade for the code.

Example: The following functions

```
foo(Number x) : 'it's some number';
foo(Integer x) : 'it's an int';
```

compile to the following Java method:

```
public String foo(Number x) {
    if (x== null || x instanceof Integer) {
        return "it's␣an␣int";
    } else {
        return "it's␣some␣number";
    }
}
```

In case there is no single most general signature, one is computed. Example:

```
foo(Number x, Integer y) : 'it's some number and an int';
foo(Integer x, Number x) : 'it's an int and a number';

public String foo(Number x) {
    if ((x== null || x instanceof Number) && (y== null ||y instanceof Integer)) {
        return "it's␣some␣number␣and␣an␣int";
    } else if ((x== null || x instanceof Integer) && (y== null || y instanceof Number)){
```

```
        return "it's an int and a number";
    } else {
        throw new UnsupportedOperationException("foo is not implemented for arguments "+x+" and "+y);
    }
}
```

As you can see a null reference is always a match. If you want to fetch null you can declare a parameter using the type java.lang.Void.

```
foo(Void x) : throw new NullPointerException("x");
foo(Number x) : 'it's some number';
foo(Integer x) : 'it's an int';
```

Which compiles to the following Java code:

```
public String foo(Number x) {
    if (x== null) {
        throw new NullPointerException("x")
    } else if (x== null || x instanceof Integer) {
        return "it's an int";
    } else {
        return "it's some number";
    }
}
```

So essential we put the polymorphic invocation logic into the declaration and not into the method call. The nice thing is that polymorphic overloaded functions can be transparently called from Java, and behave exactly the same as if they were called from Xtend2.

2.5.4 Syntax

Syntactically Xtend functions are much like Java methods, expect that there are no static methods, the return types are optional and of course the function body consists of one expression instead of a sequence of statements.

```
FunctionDef : ('public'|'protected'|'private'|'abstract'|'final')* TypeParameters? TypeRef? ID'('(ParameterDeclaration (','
    ('throws' TypeRef (',' TypeRef)*)?
    ( ':' Expression ';'
    | BlockExpression
    | ';')
```

2.6 Expressions (in addition Xbase)

Xtend2 adds a couple of expressions to the basic set of expressions provided by Xbase.

2.6.1 Rich Strings

Of course there is the template expression, which is used to write readable string concatenation, which is the main thing you do when writing a code generator. Xtend2 reuses the syntax known from the well known and widely used Xpand template language

(in fact Xtend2 is considered the successor to Xpand and Xtend). Let's have a look at an example of how a typical function with template expressions look like:

```
toClass(Entity this) :
    package packageName;

    placeImports

    public class name IF extendedType!=nullextends extendedTypeENDIF{
        FOREACH members
            member.toMember
        ENDFOREACH
    }
    ;
```

If you are familiar with Xpand, you'll notice that it is exactly the same syntax. The difference is, that the template syntax is actually an expression, which means it can occur everywhere where an expression is expected. For instance in conjunction the powerful switch expression from Xbase:

```
toMember(Member this) :
    switch(this) {
        Field :private type name ;;
        Method case isAbstract : abstract ...;
        Method: ..... ;
    };
```

IF in Rich Strings

There is a special IF to be used within rich strings which is identical in syntax and meaning to the old IF from Xpand. Note that you could also use the if expression, but since it has not an explicit terminal token, it is not as readable in that context.

FOREACH in Rich Strings

Also the FOREACH statement is available and can only be used in the context of a rich string. It also supports the SEPARATOR and ITERATOR declaration from Xpand.

Think about whether we really want to stick to the verbose syntax form previous Xpand for SEPARATOR and ITERATOR

Typing

The rich string is translated to an efficient string concatenation and the return type of a rich string is java.lang.CharStream which allows much room for efficient implementation.

2.6.2 Shortcut navigation through lists

In Xtend2 one usually navigates over graphs which contain a lot of collections. If you for instance need the qualified name of the interfaces a type is extending you could write the following using closures:

```
myType.interfaces.collect(e|e.qualifiedName)
```

However because this situation is so common Xtend2 provides a special sugared expression for that. So instead of the code above you can write:

```
myType.interfaces.*qualifiedName
```

Note that we decided to come up with an explicit operator (.*) because overloading the '.' operator as it is the case in Xpand, has caused a lot of surprises in the past. With an explicit operator the tooling as well as the user always can distinct whether you invoke a feature on the iterables elements or on the iterable itself.

The operator works on all members of java.lang.Iterable. the return type is always an java.lang.Iterable<T> where T is the return type of the called feature.

2.6.3 Extension Method Syntax

Static functions as well as any members annotated with @Extension can be called using the extension method syntax. This means that a function imported through a static import, like e.g. java.util.Collections.singleton(T) can be invoked using the member syntax.

Example:

```
"Foo".singleton
```

is the same as

```
singleton("Foo")
```

Note that extension methods never shadow a member of the current reference. That is if java.lang.String has a field singleton, a method singleton() or a method getSingleton(), those memebers would be referenced. That is done at compile time, so the tooling is able to tell you what you actually reference.

Static functions as well as extensions in the previous version of Xtend make clients not only depend on a certain signature but on the implementation as it is not possible to exchange the implementation of a static function. That's where the @Extension annotation in conjunction with dependency injection comes in.

If you have declared a field with @Extension than all members of that type become available for extension method syntax.

Example

Imagine the following Java interface

```
interface MyExtensions {
    String getComputedProperty(SomeType type);
}
```

With Java you would have to call this method on an instance of MyExtensions like so:

```
myExtensions.getComputedProperty(someType)
```

In Xtend2 you can have the instance injected like in Java:

```
@Inject @Extension MyExtensions extensions;
```

But instead of using the long expression known from Java (which would also work) you are able to use the extension method syntax:

```
someType.computedProperty
```

It will statically bind to the right method and even more important it doesn't bind to a specific implementation but just to the signature. The implementation can be provided through dependency injection.

List of External Links

Todo list