Xtend2 Language Specification

Sven Efftinge, Sebastian Zarnekow, et al.

February 14, 2011

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, with intelligent whitespace handling
- built-in support for dependency injection (based in Google Guice)
- injected extension methods
- multiple dispatch aka polymorphic method invocation
- Advanced type inference
- Uses Xbase expressions, which means :
    - No checked exceptions
    - Object-oriented
    - No statements - Everything is an expression
    - Closures
    - Type inference
    - Java Beans Properties are called using simple property access and assignment operators
    - 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 ˆ in case it conflicts with a keyword. Also you don't terminate a package declaration with a semicolon.

2.1.1 Syntax

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

2.1.2 Examples

```
package org.eclipse.xtext package my.^public.^package
```

2.2 Imports

The ordinary imports of type names are equivalent to the imports known from Java. Again one can escape any names conflicting with keywords using the ˆ and the import statemet is never terminated with a semicolon.

2.2.1 Importing Extension Methods

In addition one can add the keyword extension after the import (where the keyword static occurs in Java), which makes any static methods from a certain type available as an extension method (§2.6.2). The extension keyword only works with qualified imports (i.e. non-wildcard imports). If you only want to make some methods from the imported type available using the extension syntax, you can provide a list of function names. You cannot further specify type arguments.

Example:

```
import extension java.util.Collections.(sort, shuffle);
```

This will import four methods (sort(List<T>), sort(List<T>, Comparator<? super T>), shuffle(List<?>), shuffle(List<?>, Random)) available as extension methods (§2.6.2).

2.2.2 Syntax

```
ImportSection :
    (Import|Injection)∗;
Import :
    'import' 'extension'? QualifiedName (('.''∗')
                                        |'('ID (',' ID)')')?;
```

2.3 Dependency Injection

Xtend2 has built-in support for dependency injection, which means that declaring dependencies is a first class feature of Xtend2. It is done as part of the import section and looks like this:

```
inject MyService myService;
```

This will translate to Java as a field as well a setter method annotated with @Inject:

```
private MyService myService;

@Inject
public void setMyService(MyService myService) {
    this.myService = myService;
}
```

2.3.1 Name can be inferred

You could even leave the name out as by default it will be the lower case version of the simple name of the class. That is the following inject declaration would be equvalent with the former.

```
inject MyService;
```

2.3.2 Guice Keys

Xtend2 uses Guice as the dependency injection container, which in turn uses types as keys. Those types can be generified, which is why you can write the following in Xtend2.

```
inject HashMap<String,MyType> myCache;
```

Also Guice allows to use annotations to further distinct between different implementations of the same type. This is usually done by using the `com.google.inject.name.Named` annotation or a custom annotation which is itself annotated with `com.google.inject.BindingAnnotation`.

Xtend supports these two variants:

```
inject @CustomAnnotation my.Service customService;
inject @"NamedThingy" my.Service namedService;
```

Providers are declared like in plain Guice. That is you wrap your type into `com.google.inject.Provider`. Example:

```
inject @CustomAnnotation Provider<Service> customServiceProvider;
```

2.3.3 Injected Extension Methods

The `extension` keyword is also available for `inject` which is much nicer than importing static methods, because you don't bind your code to actual implementation, since the implementation can be easily changed through the global dependency injection configuration (i.e. Guice modules).

Example:

Imagine you had something like IterableExtensions which adds some higher-order functions to the java.util collection library (we will likely ship a library based on google guava).

```
class IterableExtensions {
   public <T> T find(Iterable<T> elements, Function<T,Boolean> func) {
      ...
   }
}
```

You declare the dependency like this:

```
inject extension my.common.IterableExtensions;
```

and can then use it as if those functions were members of any `java.lang.Iterable`:

```
myIterable.find(e|e.name=="Foo")
```

The nice thing with using injection as opposed to static methods is, that in case there is a bug in the extension or it is implemented inefficiently or you just need a different strategy, you can simply exchange the component with another implementation. You do this without modifying the library nor the client code. You'll only have to change the binding in your guice module. Also this gives you a general hook for any AOP-like thing you would want to do, or allows you to write against an SPI, where the concrete implementation can be provided by a third party.

```
ImportSection :
    (Import|Injection)*;
Import :
    ...
Injection :
    'inject' 'extension'?
    ('@'STRING | '@'TypeReference)?
    TypeReference ('('ID (',' ID)*')')?;
```

2.4 Class Declaration

The class declaration reuses a lot of Java's syntax but still 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. Java's default "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.

2.4.1 Inheritance

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

2.4.2 Generics

Full Java Generics with the exact same syntax and semantics are supported.

2.4.3 Abstract Classes

A class is automatically abstract if one of its functions is abstract. This is the case if the function's expression (body) is not defined.

2.4.4 Syntax

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

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 exceptions are dispatch methods, which compile to a single method. This is explained in the section about subsection 2.5.6.

An example of a function declaration

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

2.5.1 Visibility

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

2.5.2 Overriding Functions

Like Java methods an Xtend2 function can be declared non overridable using the keyword final.

Also if a function overrides a method from the super class, the override keyword is mandatory.

Example:

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

2.5.3 Abstract Functions

A function is automatically considered abstract if its expression (body) is not defined.

Example: `Boolean myAbstractFunction(String s1,String s2);`

In such cases the class needs to be flagged abstract otherwise the compiler will complain.

2.5.4 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.

2.5.5 Generics

Full Java Generics with the exact same syntax and semantics are supported. Raw types, i.e. parameterized types without type parameters are treated as if each type parameter were a wild card.

That is List is the same as List<?>

2.5.6 Dispatch Functions

It is possible to overload functions like in Java, and the resolution is also like in Java based on the static type of the arguments and only dynamic polymorphic on the receiver. However Xtend2 supports the special notion of dispatch functions, which makes such overloaded functions polymorphicly dispatched.

Such methods have the keyword dispatch preceding the declaration.

If they are marked as dispatch functions, the compiler will use or infer the function with the most common argument types and generates Java method made up of if-else cascaded dispatching between the different dispatch functions based on the actual runtime types of the arguments. This has a lot of advantages and is a very convenient way to add functionality to heterogeneous data structures especially with deeper type hierarchies (i.e. EMF models and other typed domain models). It essentially eliminates the problem the visitor pattern is trying to solve.

This allows you to write a code generator by defining a function for all kinds of AST elements you want to process.

```
class MyCompiler {
    dispatch compile(CompilationUnit cu) :
        ... cu.types.compile() ...;
    dispatch compile(Interface interface) :
        ... cu.members.compile() ...;
    dispatch compile(Class cu) :
        ... cu.members.compile() ...;
    dispatch compile(Field cu) :
        ...;
    ...
}
```

How case methods are translated to Java

The polymorphic behavior of dispatch functions shall be transparently work no matter you call such a function from Java, Xtend2 or any other JVM language. Therefore the runtime dispatch is not done on the caller's side but on the declaration side. For each set of case methods where the name is equal and the number of arguments is equal, the most common denominator signature is taken or computed if there is no most common denominator. Only for that signature a Java method with the given name is generated, all the dispatch functions are generated to protected methods where the name is prefixed with an underscore.

Within the implementation of the dispatch method the correct case method is looked up at runtime using if else cascades. 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

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

compile to the following Java method:

```
public String foo(Number x) {
    if (x instanceof Integer) {
        return _foo((Integer)x);
    } else if (x instanceof Number) {
        return _foo((Number)x);
    }
    throw new IllegalArgumentException(
        "Couldn't␣dispatch.␣Argument␣was␣Number␣x␣:"+x);
}
```

```
protected String _foo(Integer x) {
    return "It's␣an␣int";
}

protected String _foo(Number x) {
    return "It's␣a␣number";
}
```

In case there is no single most general signature, one is computed and the different overloaded methods are matched in the order they are declared within the class file. Example:

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

public String foo(Number x, Number y) {
    if ((x instanceof Number)
        && (y instanceof Integer)) {
        return _foo((Number)x,(Integer)y);
    } else if ((x instanceof Integer)
        && (y instanceof Number)){
        return _foo((Integer)x,(Number)y);
    } else {
        throw new IllegalArgumentException(
            "Couldn't␣handle␣Number␣x:"+x+",␣Number␣y:"+y);
    }
}
```

As you can see a null reference is never a match. If you want to fetch null you can declare a parameter using the type java.lang.Void.

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

Which compiles to the following Java code:

```
public String foo(Number x) {
    if (x instanceof Integer) {
        return _foo((Integer)x);
    } else if (x instanceof Number){
        return _foo((Number)x);
    } else if (x == null) {
        return _foo((Void)null);
    } else {
        throw new IllegalArgumentException(
            "Couldn't␣handle␣Number␣x:"+x+",␣Number␣y:"+y);
    }
}
```

Overloading Functions from Super Types

Any Java methods from super types conforming to the compiled form of a dispatch method are also included in the dispatch. Conforming means they have the right number of arguments and start with an underscore.

Example:

Consider the following Java class :

```
public abstract class AbstractLabelProvider {
   protected String _label(Object o) {
      // some generic implementation
   }
}
```

and the following xtend class extends it like this:

```
class MyLabelProvider extends AbstractLabelProvider {
   dispatch label(Entity this) name
   dispatch label(Method this) name+"("+params.toString(",")+"):"+type
   dispatch label(Field this) name+type
}
```

The resulting dispatch method in the generated Java class 'MyLabelProvider' looks like this:

```
public String label(Object o) {
    if (o instanceof Field) {
        return _label((Field)o);
    } else if (o instanceof Method){
        return _foo((Method)o);
    } else if (o instanceof Entity){
        return _foo((Entity)o);
    } else if (o instanceof Object){
        return _foo((Object)o);
    } else {
        throw new IllegalArgumentException(
            "Couldn't handle Object o:"+o);
    }
}
```

2.5.7 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'|'final'|'case'|'override')*
    TypeParameters? TypeRef?
    ID'('(ParameterDeclaration (',' ParameterDeclaration)*)?')'
    ('throws' TypeRef (',' TypeRef)*)?
    Expression
```

;

2.6 Expressions (in addition Xbase)

Xtend2 adds some 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 looks like:

```
toClass(Entity this) :'''
    package «packageName»;

    «placeImports»

    public class «name» «IF extendedType!=null»extends «extendedType»«ENDIF»{
        «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:''' ..... ''';
    };
```

Conditions 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.

Loops 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.

Typing

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

Whitespace Handling

One of the key features of rich strings is the smart handling of whitespace in the template output. The whitespace is not written into the output data structure as is but preprocessed. This allows for readable templates as well as nicely formatted output. This can be achieved by applying three simple rules when the rich string is evaluated.

1. An evaluated rich string as part of another string will be prefixed with the current indentation of the caller before it is inserted into the result.
2. Indentation in the template that is relative to a control structure will not be propagated to the output string. A control structure is a FOREACH-loop or a condition (IF) as well as the opening and closing marks of the rich string itself. The indentation is considered to be relative to such a constrol structure if the previous line ends with a control structure followed by optional white space. The amount of whitespace is not taken into account but the delta to the other lines.
3. Lines that do not contain any static text which is not whitespace but do contain control structures or invocations of other templates which evaluate to an empty string, will not appear in the output.

The behavior is best described with a set of examples. The following table assumes a data structure of nested nodes.

```
class Template {
    print(Node this) '''
        node «name» {}
    '''
}
```

```
node NodeName{}
```

The indentation before node «name» will be skipped as it is relative to the opening mark of the rich string and thereby not considered to be relevant for the output but only for readability of the template itself.

```
class Template {
    print(Node this) '''
        node «name» {
            «IF hasChildren»
                «children*.print»
            «ENDIF»
        }
    '''
}
```

```
node Parent{
    node FirstChild {
    }
    node SecondChild {
        node Leaf {
        }
    }
}
```

As in the previous example, there is no indentation on the root level for the same reason. The first nesting level has only one indentation level in the output. This is derived from the indentation of the IF hasChildren condition in the template which is nested in the node. The additional nesting of the recursive invocation children∗.print is not visible in the output as it is relative the the surrounding control structure. The line with IF and ENDIF contain only control structures thus they are skipped in the output. Note the additional indentation of the node *Leaf* which happens due to the first rule: Indentation is propagated to called templates.

2.6.2 Extension Method Syntax

Static methods from imports and methods from injected objects where the declaration is preceded with the keyword extension (§2.2.1) can be called using the member syntax. This means that a method 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 members 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 keyword in conjunction with dependency injection comes in. If you have annotated a dependency with the extension keyword, its members 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;
```

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.

2.6.3 Create Expression

Xtend2 supports model to model transformations. The single most important problem a transformation language has to solve, is to get rid of the need to transform a net of objects in two phases. This is unfortunately necessary because you would end up with endless recursion if you write a mapping between objects but these objects are connected in a circular way, which is almost always the case.

Consider you want to transform the AST of the following Java interface to Java Class with an abstract method:

```
interface Foo {
    Foo newFoo();
}
```

Note how Foo contains a method and that method again refers to Foo. We have a circular dependency. Traditionally those problems are solved by doing the transformation in multiple phases: First transform the tree hierarchy, to make sure that all declarations are in place. Then establish the cross references. The problem with this approach is that you have to split up you code. The transformation of a method needs to be split up in creating the object and setting the values and containing elements and a code section which establishes the cross reference. This complicates the code and harms readability.

In Xtend2 there is a so called create expression (similar to the create extension from previous Xtend). The example from above could be implemented like this:

```
toAbstractClass(Interface intf) :
    create new Class() {
        name = intf.name;
        abstract = true;
        methods.addAll(intf.methods.toAbstractMethod);
    }
;
toAbstractMethod(Method m) :
    create new Method() {
        name = m.name;
        abstract = true;
        returnType = m.returnType.toAbstractClass; // recursive
    }
;
```

the create expression contains two expressions. The first expression right after the keyword is called the creator expression, the second expression is called the initializer. The execution semantics is that the result of the creator expression is cached and the key is comprised of all local variables and the create expression itself. This means that whenever the same create expression with the exact same local variables (equality) is executed again the expression will just return the cached value. After the value is cached it executes the initializer. The nice thing is, that you don't have to care about circular references since a call to the same create expression with the same arguments will return a reference to the not yet fully initialized element.

Todo list