

# **Xtend User Guide**

December 7, 2011

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Getting Started . . . . .	4
1.1.1	Hello World . . . . .	5
1.1.2	The Runtime Library . . . . .	5
1.1.3	The Xtend Tutorial . . . . .	5
<b>2</b>	<b>Types</b>	<b>7</b>
2.1	Conformance and Conversion . . . . .	7
<b>3</b>	<b>Classes and Members</b>	<b>8</b>
3.1	Package Declaration . . . . .	8
3.2	Imports . . . . .	8
3.3	Class Declaration . . . . .	9
3.4	Constructors . . . . .	9
3.5	Fields . . . . .	10
3.6	Methods . . . . .	10
3.6.1	Overriding Methods . . . . .	11
3.6.2	Declared Exceptions . . . . .	11
3.6.3	Inferred Return Types . . . . .	11
3.6.4	Dispatch Methods . . . . .	12
3.7	Annotations . . . . .	15
3.8	Extension Methods . . . . .	15
3.8.1	Local Extension Methods . . . . .	15
3.8.2	Library Extensions . . . . .	16
3.8.3	Extension Imports . . . . .	16
3.8.4	Extension Fields . . . . .	16
<b>4</b>	<b>Expressions</b>	<b>18</b>
4.1	Literals . . . . .	18
4.1.1	String Literals . . . . .	18
4.1.2	Integer Literals . . . . .	19
4.1.3	Boolean Literals . . . . .	19
4.1.4	Null Literal . . . . .	19
4.1.5	Type Literals . . . . .	19
4.1.6	Function Types . . . . .	19
4.2	Type Casts . . . . .	19

4.3	Infix Operators and Operator Overloading . . . . .	20
4.3.1	Short-Circuit Boolean Operators . . . . .	21
4.3.2	Examples . . . . .	21
4.3.3	Assignments . . . . .	21
4.4	Variable Declarations . . . . .	22
4.4.1	Typing . . . . .	22
4.5	Feature Calls . . . . .	22
4.5.1	Property Access . . . . .	23
4.5.2	Implicit Variables <i>this</i> and <i>it</i> . . . . .	23
4.5.3	Null-Safe Feature Call . . . . .	23
4.6	Constructor Call . . . . .	23
4.7	Closures . . . . .	24
4.7.1	Typing . . . . .	24
4.7.2	Function Mapping . . . . .	24
4.7.3	Implicit Parameter <i>it</i> . . . . .	25
4.7.4	Exceptions in Closures . . . . .	25
4.7.5	Builder Syntax . . . . .	26
4.8	If Expression . . . . .	26
4.9	Switch Expression . . . . .	27
4.9.1	Type guards . . . . .	27
4.10	Blocks . . . . .	28
4.11	For Loop . . . . .	28
4.12	While Loop . . . . .	29
4.13	Do-While Loop . . . . .	29
4.14	Return Expression . . . . .	30
4.15	Throwing Exceptions . . . . .	30
4.16	Try, Catch, Finally . . . . .	30
4.17	Template Expressions . . . . .	30
4.17.1	Conditions in Templates . . . . .	31
4.17.2	Loops in Templates . . . . .	31
4.17.3	Typing . . . . .	31
4.17.4	White Space Handling . . . . .	31

# 1 Introduction

Xtend is a statically-typed programming language which is tightly integrated with and runs on the Java Virtual Machine. It has its roots in the Java programming language but improves on many concepts:

- *Extension methods* - enhance closed types with new functionality
- *Multiple dispatch* a.k.a. polymorphic method invocation
- *Closures* - concise syntax for anonymous function literals
- *Operator overloading* - make your libraries even more expressive
- *Powerful switch expressions* - type based switching with implicit casts
- *Template expressions* - with intelligent white space handling
- *No statements* - everything is an expression
- *Property access syntax* - shorthands for getter and setter access
- *Advanced type inference* - you rarely need to write down type signatures
- *Full support for Java Generics* - including all conformance and conversion rules
- *Translates to Java* not bytecode - understand what is going on and use your code for platforms such as Android or GWT

It is not aiming at replacing Java all together. Therefore its library is a thin layer on top of the Java Development Kit (JDK) and interacts with Java exactly the same as it interacts with Xtend code. Java can call Xtend methods, too, in a completely transparent way. And of course, Xtend provides a modern Eclipse-based IDE closely integrated with Eclipse's Java Development Tools (JDT).

## 1.1 Getting Started

Xtend requires Eclipse 3.5 or higher and a Java SDK version 5 or above. The easiest way to install the SDK is via Eclipse Marketplace. There even is a client integrated into Eclipse. You'll find it in the Help-menu.

### 1.1.1 Hello World

Let us start with a simple "Hello World" example. In Xtend, that reads as

```
class HelloWorld {  
    def static void main(String[] args) {  
        println("Hello World")  
    }  
}
```

The Xtend code resembles Java a lot. You can already see how the syntactic noise is reduced: No semicolons, no return types etc.

An Xtend class resides in a plain Java project. As soon as the SDK is installed, Eclipse will automatically translate it to Java code. You'll find it in a source folder *xtend-gen*. The hello world example is translated to the following Java code:

```
import org.eclipse.xtext.xbase.lib.InputOutput;  
  
public class HelloWorld {  
    public static void main(final String[] args) {  
        InputOutput.<String>println("Hello World");  
    }  
}
```

### 1.1.2 The Runtime Library

The only surprising fact in the generated Java code may be the library class `InputOutput`. Many features of Xtend are not built into the language itself but provided via the library *org.eclipse.xtend2.lib*. The library is available from a Maven repository (check the website for details) and provides a lot of useful functionality annotating existing types from Java's SDK.

It provides means to create collections in a readable way:

```
val myList = newArrayList(1, 2, 3)  
val mySet = newHashSet(4, 5, 6)  
val myMap = newHashMap(1 -> 'one', 2 -> 'two', 3 -> 'three')
```

It also extends the collection types with a lot of very useful functions. One example is the ubiquitous map function:

```
val listOfNames = myList.map( e | myMap.get(e) )
```

Operators to concat collections or do arithmetics with types like `java.lang.BigDecimal` are also available.

### 1.1.3 The Xtend Tutorial

The best way to get acquainted with the language is to materialize the *Xtend Tutorial* example project in your workspace. You will find it in the *New > Project..* wizard dialog.

The project contains a couple of sample Xtend files which show the different language concepts in action. Looking into the *xtend-gen* folder which holds the compiled Java code will help you understand the concepts better.

## 2 Types

Xtend completely supports Java's type system: The primitive types as **int** or **boolean** are available as well as all classes and interfaces that reside on the classpath.

Java Generics are fully supported, such that you can define type parameters and type arguments in just the same way as in Java.

As Xtend classes compile to Java classes, you can integrate both Java classes in Xtend and Xtend classes in Java. Note that Xtend does not have an own syntax for interfaces, as the Java syntax is already very concise.

### 2.1 Conformance and Conversion

Conformance is used in order to find out whether some expression can be used in a certain situation. For instance when assigning a value to a variable, the type of the right hand expression needs to conform to the type of the variable.

As Xbase implements the unchanged type system of Java it also fully supports the conformance rules defined in The Java Language Specification.

## 3 Classes and Members

At a first glance an Xtend file pretty much looks like a Java file. It starts with a package declaration followed by an import section and a class definition. The class in fact is directly translated to a Java class in the corresponding Java package. As in Java, a class can have constructors, fields and methods.

Here is an example:

```
package com.acme

import java.util.List

class MyClass {
    String name

    new(String name) {
        this.name = name
    }

    def String first(List<String> elements) {
        elements.get(0)
    }
}
```

### 3.1 Package Declaration

Package declarations are mostly like in Java. There are two the small differences:

- An identifier can be escaped with a ^ character in case it conflicts with a keyword.
- There is no terminating semicolon.

```
package com.acme
```

### 3.2 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 a ^. In contrast to Java, the import statement is never terminated with a semicolon. Xtend also features static imports but allows only a wildcard \* at the end, i.e. you cannot import single



members using a static import. Non-static wildcard imports are deprecated for the benefit of better tooling.

As in Java all classes from the `java.lang` package are implicitly imported.

```
import java.math.BigDecimal
import static java.util.Collections.*
```

Static methods of helper classes can also be imported as *extensions*. See the section on extension methods (§3.8) for details.

### 3.3 Class Declaration

The class declaration reuses a lot of Java’s syntax but still is a bit different in some aspects: Java’s default “package private” visibility does not exist in Xtend. As an Xtend class is compiled to a top-level Java class and Java does not allow **private** or **protected** top-level classes any Xtend class is **public**. It is possible to write this explicitly.

Xtend’s approach to inheritance is conceptionally the same as in Java. Single inheritance of Java classes as well as implementing multiple Java interfaces is supported. Because Xtend classes are compiled to Java, Xtend classes can extend other Xtend classes, and even Java classes can inherit from Xtend classes.

The most simple class looks like this:

```
class MyClass {
}
```

A more advanced class declaration in Xtend:

```
class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess,
        Cloneable, java.io.Serializable {
    ...
}
```

### 3.4 Constructors

An Xtend class can define one or more constructors. Unlike Java, the keyword *new* is used to declare a constructor. Constructors can also delegate to other constructors using **this**(args...) in their first line.

```
class MyClass extends AnotherClass {
    new(String s) {
        super(s)
    }

    new() {
        this("default")
    }
}
```

The same rules with regard to inheritance apply as in Java, i.e. if the super class does not define a no-argument constructor, you have to explicitly call one using **super**(args...) as the first expression in the body of the constructor.

The default visibility of constructors is **public** but you can also specify **protected** or **private**.

## 3.5 Fields

An Xtend class can define fields, too. As for Java fields, you can optionally declare an initializer expression and use annotations on fields. Fields marked as **static** will be compiled to static Java fields.

```
class MyClass {  
    @Nullable String name  
    int count = 1  
    String string = new String()  
    static boolean debug = false  
    ...  
}
```

The default visibility is **private**. You can also declare it explicitly as being **public**, **protected**, or **private**.

A specialty of Xtend are fields that provide *extension methods* which are covered in their own section (§3.8).

## 3.6 Methods

Xtend methods are declared within a class and are translated to a corresponding Java method with exactly the same signature. The only exceptions are dispatch methods, which are explained here (§3.6.4).

```
def String first(List<String> elements) {  
    elements.get(0);  
}
```

The default visibility of a plain method is **public**. You can explicitly declare it as being **public**, **protected**, or **private**.

Xtend supports the **static** modifier for methods:

```
def static createInstance() {  
    new MyClass('foo')  
}
```

### 3.6.1 Overriding Methods

Methods can override other methods from the super class or implemented interface methods using the keyword **override**. If a method overrides a method from a super type, the **override** keyword is mandatory and replaces the keyword **def**. As in Java **final** methods cannot be overridden by subclasses.

Example:

```
override String first(List<String> elements) {  
    elements.get(0);  
}
```

### 3.6.2 Declared Exceptions

Xtend does not force you to catch checked exceptions. Instead, they are rethrown in a way the compiler does not complain about a missing throws clause, using the sneaky-throw technique introduced by Lombok. Nevertheless, you can still declare the exceptions thrown in a method's body using the same **throws** clause as in Java.

```
/*  
 * throws an Exception  
 */  
def void throwException() throws Exception {  
    throw new Exception()  
}  
  
/*  
 * throws an Exception without declaring it  
 */  
def void sneakyThrowException() {  
    throw new Exception()  
}
```

### 3.6.3 Inferred Return Types

If the return type of a method can be inferred from its body it does not have to be declared. That is the method

```
def String first(List<String> elements) {  
    elements.get(0);  
}
```

could be declared like this:

```
def first(List<String> elements) {  
    elements.get(0);  
}
```

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

### 3.6.4 Dispatch Methods

Generally, method binding works just like method binding in Java. Method calls are bound based on the static types of arguments. Sometimes this is not what you want. Especially in the context of extension methods (§3.8) you would like to have polymorphic behavior.

Dispatch methods make a set of overloaded methods polymorphic. That is the runtime types of all given arguments are used to decide which of the overloaded methods is being invoked. This essentially removes the need for the quite invasive visitor pattern.

A dispatch method is marked using the keyword **dispatch**.

```
def dispatch printType(Number x) {  
    "it's a number"  
}  
  
def dispatch printType(Integer x) {  
    "it's an int"  
}
```

For a set of visible dispatch methods in the current type hierarchy, the compiler infers a common signature using the common super type of all declared arguments and generates a Java method made up of **if instanceof else** cascades. It dispatches to the different available methods. The actually declared methods are all compiled to Java methods that are prefixed with an underscore.

For the two dispatch methods in the example above the following Java code would be generated:

```
protected String _printType(final Number x) {  
    return "it's a number";  
}  
  
protected String _printType(final Integer x) {  
    return "it's an int";  
}  
  
public String printType(final Number x) {  
    if (x instanceof Integer) {  
        return _printType((Integer)x);  
    } else {  
        return _printType(x);  
    }  
}
```

Note that the **instanceof** cascade is ordered such that more specific types come first.

The default visibility of the underscore methods is **protected**. If all dispatch methods explicitly declare the same visibility, this will be the visibility of the inferred dispatcher, too. Otherwise it is **public**.

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:

```
def dispatch printTypes(Number x, Integer y) {  
    "it's some number and an int"  
}  
  
def dispatch printTypes(Integer x, Number y) {  
    "it's an int and a number"  
}
```

generates the following Java code :

```
public String printTypes(final Number x, final Number y) {  
    if (x instanceof Integer) {  
        return _printTypes((Integer)x, y);  
    } else if (y instanceof Integer) {  
        return _printTypes(x, (Integer)y);  
    } else {  
        throw new IllegalArgumentException("Unhandled parameter types: " +  
            Arrays.<Object>asList(x, y).toString());  
    }  
}
```

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`.

```
def dispatch printType(Number x) {  
    "it's some number"  
}  
  
def dispatch printType(Integer x) {  
    "it's an int"  
}  
  
def dispatch printType(Void x) {  
    "it's null"  
}
```

Which compiles to the following Java code:

```
public String printType(final Number x) {  
    if (x instanceof Integer) {  
        return _printType((Integer)x);  
    } else if (x instanceof Number) {  
        return _printType((Number)x);  
    } else if (x == null) {  
        return "it's null";  
    }  
}
```

```

        return _printType((Void)null);
    } else {
        throw new IllegalArgumentException("Unhandled parameter types: " +
            Arrays.<Object>asList(x).toString());
    }
}

```

## Dispatch Methods and Inheritance

Any visible 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 have the same name (starting with an underscore).

For example, consider the following Java class :

```

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

```

and the following Xtend class which extends the Java class :

```

class MyLabelProvider extends AbstractLabelProvider {
    def dispatch label(Entity it) {
        name
    }

    def dispatch label(Method it) {
        name+"("+params.join(",")+"): "+type
    }

    def dispatch label(Field it) {
        name+type
    }
}

```

The resulting dispatch method in the generated Java class MyLabelProvider would then look like this:

```

public String label(final Object it) {
    if (it instanceof Entity) {
        return _label((Entity)it);
    } else if (it instanceof Field) {
        return _label((Field)it);
    } else if (it instanceof Method) {
        return _label((Method)it);
    } else {
        return _label(it);
    }
}

```

## Static Dispatch Methods

Even static dispatch methods are allowed. The same rules apply, but you cannot mix static and non-static dispatch methods.

## 3.7 Annotations

Xtend supports Java annotations. The syntax is exactly like defined in the Java Language Specification. Annotations are available on classes, fields, methods and parameters. Here is an example:

```
@TypeAnnotation(typeof(String))
class MyClass {
    @FieldAnnotation(children = {@MyAnno(true), @MyAnno(false)})
    String myField

    @MethodAnnotation(children = {@MyAnno(true), @MyAnno})
    def String myMethod(@ParameterAnnotation String param) {
        //...
    }
}
```

## 3.8 Extension Methods

Extensions methods are a technique to add behavior to existing classes without modifying their code. This feature is actually where Xtend got its name from. They are based on a simple syntactic trick: Instead of passing the first argument of an extension method inside the parentheses of a call, the method is called on the argument parameter as if it was one of its members.

```
"hello".toFirstUpper() // calls toFirstUpper("hello")
```

Method calls in extension syntax often result in much better readable code, as function calls are rather concatenated than nested. They also allow to add methods in a specific context only.

### 3.8.1 Local Extension Methods

All methods of the current Xtend class are automatically available in extension syntax. For example

```
class MyClass {
    def doSomething(Object obj) {
        // do something with obj
    }

    def extensionCall(Object obj) {
        obj.doSomething() // calls this.doSomething(obj)
    }
}
```

```
}  
}
```

### 3.8.2 Library Extensions

The static methods of the classes in the Xtend runtime library (§1.1.2) are automatically available as extensions, e.g.

```
newArrayList() // CollectionLiterals.newArrayList()  
"hello".toFirstUpper // StringExtensions.toFirstUpper(String)
```

### 3.8.3 Extension Imports

In Java, you would usually write a helper class with static methods to decorate an existing class with additional behavior. In order to integrate such static helper classes, Xtend allows to put the keyword **extension** after the **static** keyword of a static import (§3.2) thus making all imported static functions available as extensions methods.

The following import declaration

```
import static extension java.util.Collections.*
```

allows to use its methods like this:

```
new MyClass().singletonList()  
// calls Collections.singletonList(new MyClass())
```

Although this is supported it is generally much nicer to use extension fields (§3.8.4), because they allow to change the actual implementation easily.

### 3.8.4 Extension Fields

You can make the instance methods provided by the field available as extension methods, by adding the keyword **extension** to the field declaration.

Imagine you want to add a method `fullName()` to a closed type `Person`. With extension methods, you could declare the following class

```
class PersonExtensions {  
    def getFullName(Person p) {  
        p.forename + " " + p.name  
    }  
}
```

And if you have an instance of this class injected as extension like this:

```
class PersonPrinter {  
    @Inject extension PersonExtensions  
    ...  
}
```

The method is available on the extension scope of the class `Person`. This is why you can skip the field's name. You can now write the following



```
def print(Person myPerson) {  
    myPerson.getFullName()  
}
```

which is translated to the Java method

```
public String print(final Person myPerson) {  
    String _fullName = this._personExtensions.getFullName(myPerson);  
    return _fullName;  
}
```

where `_personExtensions` is the default name of the field. Of course the property shorthand (see section on property access (§4.5.1)) is still available.

```
myPerson.fullName
```

Using dependency injection in combination with the extension modifier has a significant advantage over to static extension imports (§3.8.3): You can simply exchange the component that provides the referenced extension with another implementation without touching the client code. You will only have to change the binding in your dependency injection configuration. 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.

## 4 Expressions

There are no statements in Xtend. Instead, everything is an expression and has a return value. That allows to use every Xtend expression on the right hand side of an assignment. For example, as a **try catch** is an expression the following code is legal in Xtend:

```
val data = try {  
    fileContentsToString('data.txt')  
} catch (IOException e) {  
    'dummy data'  
}
```

If `fileContentsToString()` throws an `IOException`, it is caught and a default value is returned and assigned to the variable `data`.

In Xtend, expressions appear as initializers of fields (§3.5) or as the bodies of constructors or methods. A method body in Xtend can either be a single block expression (§4.10) or a template expression (§4.17).

### 4.1 Literals

A literal denotes a fixed unchangeable value. Literals for strings, integers, boolean values, **null** and Java types are supported.

#### 4.1.1 String Literals

A string literal is a valid expression and returns an instance of `java.lang.String` of the given value. String literals are enclosed by a pair of single quotes or double quotes allowing to use the respective other unquoted inside the string. Special characters can be quoted with a backslash or defined using Java's unicode notation. On the contrary to Java, Xtend's strings can span multiple lines without the need to quote newline characters.

```
'Hello World !'  
"Hello World !"  
'Hello "World" !'  
"Hello \'World\' !"  
'\u00a1Hola el mundo!'  
"Hello  
  
    World !"
```

### 4.1.2 Integer Literals

An integer literal creates an **int**. There is no signed integer. If you put a minus operator in front of an integer literal it is taken as a `UnaryOperator` with one argument (the positive integer literal).

```
42
234254
-1 // an expression consisting of the unary – operator and an integer literal
```

### 4.1.3 Boolean Literals

There are two boolean literals, **true** and **false** which correspond to their Java counterpart of type **boolean**.

### 4.1.4 Null Literal

The null pointer literal is **null**. It is a member of any reference type and the only member of the type `java.lang.Void`.

### 4.1.5 Type Literals

Type literals are specified using the keyword **typeof** :

```
typeof(java.lang.String) // yields java.lang.String.class
```

### 4.1.6 Function Types

Xbase introduces closures (§4.7), and therefore an additional function type signature. On the Java level a closure (or more generally any function object) is just an instance of one of the types in `Functions`, depending on the number of arguments. However, as closures are a very important language feature, a special syntax for function types has been introduced. So instead of writing

```
Function1<String,Boolean>
```

one can write

```
(String)=>boolean
```

## 4.2 Type Casts

Type cast behave like casts in Java, but have a slightly better readable syntax. Type casts bind stronger than any other operator but weaker than feature calls.

The conformance rules for casts are defined in the Java Language Specification. Here are some examples:

```
something as MyClass
"" as Object
```

### 4.3 Infix Operators and Operator Overloading

There are a couple of common predefined infix operators. In contrast to Java, the operators are not limited to operations on certain types. Instead an operator-to-method mapping allows users to redefine the operators for any type just by implementing the corresponding method signature. As an example, the Xtend runtime library (§1.1.2) contains a class `BigDecimalExtensions` that defines operators for `BigDecimal`s which allows the following code:

```
val x=new BigDecimal('2.71')
val y=new BigDecimal('3.14')
x+y // calls BigDecimalExtension.operator_plus(x,y)
```

The following defines the operators and the corresponding Java method signatures / expressions.

e1 += e2	e1.operator_add(e2)
e1    e2	e1.operator_or(e2)
e1 && e2	e1.operator_and(e2)
e1 == e2	e1.operator_equals(e2)
e1 != e2	e1.operator_notEquals(e2)
e1 < e2	e1.operator_lessThan(e2)
e1 > e2	e1.operator_greaterThan(e2)
e1 <= e2	e1.operator_lessEqualsThan(e2)
e1 >= e2	e1.operator_greaterEqualsThan(e2)
e1 -> e2	e1.operator_mappedTo(e2)
e1 .. e2	e1.operator_upTo(e2)
e1 + e2	e1.operator_plus(e2)
e1 - e2	e1.operator_minus(e2)
e1 * e2	e1.operator_multiply(e2)
e1 / e2	e1.operator_divide(e2)
e1 % e2	e1.operator_modulo(e2)
e1 ** e2	e1.operator_power(e2)
! e1	e1.operator_not()
- e1	e1.operator_minus()

The table above also defines the operator precedence in ascending order. The blank lines separate precedence levels. The assignment operator `+=` is right-to-left associative

in the same way as the plain assignment operator `=` is. That is `a = b = c` is executed as `a = (b = c)`, all other operators are left-to-right associative. Parenthesis can be used to adjust the default precedence and associativity.

### 4.3.1 Short-Circuit Boolean Operators

If the operators `||` and `&&` are used in a context where the left hand operand is of type `boolean`, the operation is evaluated in short circuit mode, which means that the right hand operand might not be evaluated at all in the following cases:

1. in the case of `||` the operand on the right hand side is not evaluated if the left operand evaluates to **true**.
2. in the case of `&&` the operand on the right hand side is not evaluated if the left operand evaluates to **false**.

### 4.3.2 Examples

```
my.property = 23
myList += 23
x > 23 && y < 23
x && y || z
1 + 3 * 5 * (- 23)
!(x)
```

### 4.3.3 Assignments

Local variables (§4.4) can be reassigned using the `=` operator. Also properties can be set using this operator given the following expression:

```
myObj.myProperty = "foo"
```

The compiler first looks up whether there is an accessible Java Field called `myProperty` on the type of `myObj`. If there exists such a field it translates to the following Java expression:

```
myObj.myProperty = "foo";
```

Remember in Xtend everything is an expression and has to return something. In the case of simple assignments the return value is the value returned from the corresponding Java expression, which is the assigned value.

If there is no accessible field on the left operand's type, a method called `setMyProperty` (JavaBeans setter method) is looked up. It has to take one argument of the type (or a super type) of the right hand operand. The return value will be whatever the setter method returns (which usually is **void**). As a result the compiler translates to:

```
myObj.setMyProperty("foo")
```

## 4.4 Variable Declarations

Variable declarations are only allowed within blocks (§4.10). They are visible in any subsequent expressions in the block.

A variable declaration starting with the keyword **val** denotes a so called value, which is essentially a final (i.e. unsettable) variable. In some cases, one needs to update the value of a reference. In such situations the variable needs to be declared with the keyword **var**, which stands for 'variable'.

A typical example for using **var** is a counter in a loop:

```
{  
  val max = 100  
  var i = 0  
  while (i < max) {  
    println("Hi there!")  
    i = i + 1  
  }  
}
```

Although overriding or shadowing variables from outer scopes is allowed, it is usually only used to overload the implicit variable **it** (§4.5.2), in order to subsequently access an object's features in an unqualified manner.

Variables declared outside a closure using the **var** keyword are not accessible from within a closure.

### 4.4.1 Typing

The return type of a variable declaration expression is always **void**. The type of the variable itself can either be explicitly declared or be inferred from the right hand side expression. Here is an example for an explicitly declared type:

```
var List<String> msg = new ArrayList();
```

In such cases, the type of the right hand expression must conform (§2.1) to the type of the expression on the left side.

Alternatively the type can be left out and will be inferred from the initialization expression:

```
var msg = new ArrayList<String>(); // -> type ArrayList<String>
```

## 4.5 Feature Calls

A feature call is used to invoke members of objects, such as fields and methods, but also can refer to local variables and parameters, which are made available for the current expression's scope.

### 4.5.1 Property Access

Feature calls are directly translated to their Java equivalent with the exception, that for calls to properties an equivalent rule as described in subsection 4.3.3 applies. That is, for the following expression

```
myObj.myProperty
```

the compiler first looks for an accessible field in the type of `myObj`. If no such field exists it looks for a method called `myProperty()` before it looks for the getter methods `getMyProperty()`. If none of these members can be found the expression is unbound and a compilation error is indicated.

### 4.5.2 Implicit Variables `this` and `it`

If the current scope contains a variable named `this` or `it`, the compiler will make all its members available to the scope. That is one of

```
it.myProperty  
this.myProperty
```

is a valid expression

```
myProperty
```

is valid as well and is equivalent, as long as there is no local variable 'myProperty' on the scope, which would have higher precedence.

As `this` is bound to the surrounding object in Java, `it` can be used in finer-grained constructs such as function parameters. That is why `it.myProperty` has higher precedence than `this.myProperty`. `it` is also the default parameter name in closures (§4.7.3).

### 4.5.3 Null-Safe Feature Call

Checking for `null` references can make code very unreadable. In many situations it is ok for an expression to return `null` if a receiver was `null`. Xtend supports the safe navigation operator `?.` to make such code better readable.

Instead of writing

```
if (myRef != null) myRef.doStuff()
```

one can write

```
myRef?.doStuff()
```

## 4.6 Constructor Call

Construction of objects is done by invoking Java constructors. The syntax is exactly as in Java, e.g.

```
new String()  
new ArrayList<BigDecimal>()
```

## 4.7 Closures

A closure is a literal that defines an anonymous function. A closure also captures the current scope, so that any final variables and parameters visible at construction time can be referred to in the closure's expression.

```
val func = [String s | s.length>3]
```

The surrounding square brackets are optional if the closure is the single argument of a method invocation. That is you can write

```
myList.findFirst(e | e.name==null)
```

instead of

```
myList.findFirst([e | e.name==null])
```

But in all other cases the square brackets are mandatory.

### 4.7.1 Typing

Closures are expressions which produce *Function* objects. The type is a function type (§4.1.6), consisting of the types of the parameters as well as the return type. The return type is never specified explicitly but is always inferred from the expression. The parameter types can be inferred if the closure is used in a context where this is possible.

For instance, given the following Java method signature:

```
public <T> T find(List<T> list, Function1<T,Boolean> predicate)
```

the type of the parameter can be inferred. Which allows users to write:

```
newArrayList("Foo", "Bar").find(e | e=="Bar")
```

instead of

```
newArrayList("Foo", "Bar").find(String e | e=="Bar")
```

Here are some more examples:

```
[| "foo"] // closure without parameters  
[String s | s.toUpperCase()] // explicit argument type  
[a,b,c | a+b+c] // inferred argument types
```

### 4.7.2 Function Mapping

An Xtend closure is a Java object of one of the *Function* interfaces shipped with the runtime library of Xtend. There is an interface for each number of parameters (current maximum is six parameters). The names of the interfaces are

- *Function0<ReturnType>* for zero parameters,
- *Function1<Param1Type, ReturnType>* for one parameters,



- *Function2<Param1Type, Param2Type, ReturnType>* for two parameters,
- ...
- *Function6<Param1Type, Param2Type, Param3Type, Param4Type, Param5Type, Param6Type, ReturnType>* for six parameters,

or

- *Procedure0* for zero parameters,
- *Procedure1<Param1Type>* for one parameters,
- *Procedure2<Param1Type, Param2Type>* for two parameters,
- ...
- *Procedure6<Param1Type, Param2Type, Param3Type, Param4Type, Param5Type, Param6Type>* for six parameters,

if the return type is **void**.

In order to allow seamless integration with existing Java libraries such as the JDK or Google Guava (formerly known as Google Collect) closures are automatically coerced to expected types if those types declare only one method (methods from `java.lang.Object` do not count).

As a result given the method `IterableExtensions.sort(Iterable<T>, Comparator<? super T>)` is available as an extension method, it can be invoked like this

```
newArrayList( 'aaa', 'bb', 'c' ).sort(
    e1, e2 | if ( e1.length > e2.length ) {
        -1
    } else if ( e1.length < e2.length ) {
        1
    } else {
        0
    })
```

### 4.7.3 Implicit Parameter it

If a closure has a single parameter whose type can be inferred, the declaration of the parameter can be omitted. Use **it** to refer to the parameter inside the closure's body.

```
val (String)=>String function = [toUpperCase] // equivalent to [it | it.toUpperCase]
```

### 4.7.4 Exceptions in Closures

Checked exceptions that are thrown in the body of a closure are rethrown using the sneaky-throw technique (§3.6.2), i.e. you do not have to declare them explicitly.

### 4.7.5 Builder Syntax

If the last argument of a function call is a closure, it can be appended after the parenthesized parameter list. In combination with the implicit **it** parameter, skipping empty parentheses, and extension methods, this yields a very concise syntax.

```
val fruit = newArrayList('apple', 'pear', 'lemon')
fruit.map[toUpperCase] // same as fruit.map([it | it.toUpperCase])
```

This feature is especially useful when you are building object trees. A common pattern is to provide a set of extension functions taking two parameters: the parent object and a closure to initialize the new child. Here is an example for creating a simple tree of Nodes:

```
class Node {
    String label
    List<Node> children

    def createNode(Node parent, (Node)=>void initializer) {
        val child=new Node()
        initializer.apply(child)
        child
    }

    def tree() {
        createNode(null) [
            label="root"
            children += createNode [
                label="child0"
            ]
            children += createNode [
                label="child1"
            ]
        ]
    }
}
```

## 4.8 If Expression

An if expression is used to choose two different values based on a predicate. While it has the syntax of Java's if statement it behaves like Java's ternary operator (`predicate ? thenPart : elsePart`), i.e. it is an expression that returns a value. Consequently, you can use if expressions deeply nested within expressions.

An expression

```
if (p) e1 else e2
```

results in either the value `e1` or `e2` depending on whether the predicate `p` evaluates to **true** or **false**. The else part is optional which is a shorthand for **else null**. That means

```
if (foo) x
```

is the a short hand for

```
if (foo) x else null
```

The type of an if expression is the common super type of the return types T1 and T2 of the two expression e1 and e2.

## 4.9 Switch Expression

The **switch** expression is different from Java's. First, there is no fall through which means only one **case** is evaluated at most. Second, the use of **switch** is not limited to certain values but can be used for any object reference instead.

For a **switch** expression

```
switch e {  
  case e1 : er1  
  case e2 : er2  
  ...  
  case en : ern  
  default : er  
}
```

the main expression e is evaluated first and then each case sequentially. If the switch expression contains a variable declaration using the syntax known from section 4.11, the value is bound to the given name. Expressions of type `java.lang.Boolean` or `boolean` are not allowed in a switch expression.

The guard of each **case** clause is evaluated until the switch value equals the result of the case's guard expression or if the case's guard expression evaluates to `true`. Then the right hand expression of the case evaluated and the result is returned.

If none of the guards matches the default expression is evaluated and returned. If no default expression is specified the expression evaluates to **null**.

Example:

```
switch myString {  
  case myString.length>5 : "a long string."  
  case 'some' : "It's some string."  
  default : "It's another short string."  
}
```

### 4.9.1 Type guards

In addition to the case guards one can add a so called *Type Guard* which is syntactically just a type reference (§2) preceding an optional case keyword. The compiler will use that type for the switch expression in subsequent expressions. Example:

```

var Object x = ...;
switch x {
    String case x.length() > 0 : x.length()
    List<?> : x.size()
    default : -1
}

```

Only if the switch value passes a type guard, i.e. an instanceof operation returns **true**, the case's guard expression is executed using the same semantics explained previously. If the switch expression contains an explicit declaration of a local variable or the expression references a local variable, the type guard acts like a cast, that is all references to the switch value will be of the type specified in the type guard.

## 4.10 Blocks

The block expression allows to have imperative code sequences. It consists of a sequence of expressions, and returns the value of the last expression. The return type of a block is also the type of the last expression. Empty blocks return null. Variable declarations (§4.4) are only allowed within blocks and cannot be used as a block's last expression.

A block expression is surrounded by curly braces and contains at least one expression. It can optionally be terminated by a semicolon.

Here are two examples:

```

{
    doSideEffect("foo")
    result
}

```

```

{
    var x = greeting();
    if (x.equals("Hello ")) {
        x+"World!";
    } else {
        x;
    }
}

```

## 4.11 For Loop

The for loop

```

for (T1 variable : arrayOrIterable) expression

```

is used to execute a certain expression for each element of an array or an instance of `java.lang.Iterable`. The local variable is final, hence cannot be updated.

The return type of a for loop is **void**. The type of the local variable can be left out. In that case it is inferred from the type of the array or `java.lang.Iterable` returned by the iterable expression.

```
for (String s : myStrings) {  
    doSideEffect(s)  
}  
  
for (s : myStrings)  
    doSideEffect(s)
```

## 4.12 While Loop

A while loop

```
while (predicate) expression
```

is used to execute a certain expression unless the predicate is evaluated to **false**. The return type of a while loop is **void**.

```
while (true) {  
    doSideEffect("foo")  
}  
  
while ((i=i+1) < max)  
    doSideEffect("foo")
```

## 4.13 Do-While Loop

A do-while loop

```
do expression while (predicate)
```

is used to execute a certain expression unless the predicate is evaluated to **false**. The difference to the while loop (§4.12) is that the execution starts by executing the block once before evaluating the predicate for the first time. The return type of a do-while loop is **void**.

```
do {  
    doSideEffect("foo");  
} while (true)  
  
do doSideEffect("foo") while ((i=i+1)<max)
```

## 4.14 Return Expression

Although an explicit return is often not necessary, it is supported. In a closure for instance a return expression is always implied if the expression itself is not of type **void**. Anyway you can make it explicit:

```
listOfStrings.map(e| {  
    if (e==null)  
        return "NULL"  
    e.toUpperCase  
})
```

## 4.15 Throwing Exceptions

Like in Java it is possible to throw `java.lang.Throwable`. The syntax is exactly the same as in Java.

```
{  
    ...  
    if (myList.isEmpty)  
        throw new IllegalArgumentException("the list must not be empty")  
    ...  
}
```

## 4.16 Try, Catch, Finally

The try-catch-finally expression is used to handle exceptional situations. You are not forced to declare checked exceptions, if you do not catch checked exceptions they are rethrown in a wrapping runtime exception. Other than that the syntax again is like the one known from Java.

```
try {  
    throw new RuntimeException()  
} catch (NullPointerException e) {  
    // handle e  
} finally {  
    // do stuff  
}
```

## 4.17 Template Expressions

Templates allow for readable string concatenation, which is the main thing you do when writing a code generator. Template are surrounded by triple single or double quotes. The text of a template can be interrupted by expressions in french quotes «». One way to type these *guillemets* is to use content assist inside the template.

Let us have a look at an example of how a typical method with template expressions looks like:

```
def toClass(Entity e) '''
    package «e.packageName»;

    «placeImports»

    public class «e.name» «IF e.superClass!=null»extends «e.superClass»«ENDIF» {
        «FOR e.members»
            «member.toMember»
        «ENDFOR»
    }
'''
```

A template is actually an expression, which means it can occur everywhere where an expression is expected. For instance in conjunction the powerful switch expression (§4.9):

```
toMember(Member m) {
    switch m {
        Field : '''private «m.type» «m.name» ;'''
        Method case isAbstract : ''' abstract «...'''
        Method : ''' ..... '''
    }
}
```

#### 4.17.1 Conditions in Templates

There is a special **IF** to be used within templates 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.

#### 4.17.2 Loops in Templates

Also the **FOR** statement is available and can only be used in the context of a template. It also supports the **SEPARATOR** from Xpand. In addition, a **BEFORE** expression can be defined that is only evaluated if the loop is at least evaluated once before the very first iteration. Consequently **AFTER** is evaluated after the last iteration if there is any element.

#### 4.17.3 Typing

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

#### 4.17.4 White Space Handling

One of the key features of templates is the smart handling of white space in the template output. The white space 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 **FOR**-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 control structure if the previous line ends with a control structure followed by optional white space. The amount of white space is not taken into account but the delta to the other lines.
3. Lines that do not contain any static text which is not white space 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 n) '''
    node «n.name» {}
  '''
}
```

```
node NodeName{}
```

The indentation before `node «n.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 n) '''
    node «n.name» {
      «IF hasChildren»
        «n.children.map[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.map[print]` is not visible in the output as it is relative to 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.



## List of External Links

[http://java.sun.com/docs/books/jls/third\\_edition/html/conversions.html#5.5](http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.5)

<http://marketplace.eclipse.org/content/eclipse-xtend>

<http://projectlombok.org/features/SneakyThrows.html>

[http://java.sun.com/docs/books/jls/third\\_edition/html/conversions.html](http://java.sun.com/docs/books/jls/third_edition/html/conversions.html)

[http://java.sun.com/docs/books/jls/third\\_edition/html/j3TOC.html](http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html)

## Todo list