Xtend User Guide

June 11, 2012

Contents

1 Introduction

Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects:

- *Extension methods (§3.8)* - enhance closed types with new functionality
- *Lambda Expressions (§4.8)* - concise syntax for anonymous function literals
- *Operator overloading (§4.3)* - make your libraries even more expressive
- *Powerful switch expressions (§4.10)* - type based switching with implicit casts
- *Multiple dispatch (§3.6.6)* - a.k.a. polymorphic method invocation
- *Template expressions (§4.17)* - with intelligent white space handling
- *No statements (§4)* - everything is an expression
- *Properties (§4.6.1)* - shorthands for accessing and defining getters and setter
- *Local type inference* - you rarely need to write down type signatures anymore
- *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

The language is not aiming at replacing Java all together. Its library (§1.1.2) is just a thin layer on top of the Java Development Kit (JDK) and unlike with other JVM languages there are zero interoperability (§2.3) issues : Everything you write in Xtend interacts with Java exactly as if it were written in Java in the first place. At the same time Xtend just much more concise, readable and expressive.

Of course, you can call Xtend methods from Java, too, in a completely transparent way. Furthermore, Xtend provides a modern Eclipse-based IDE closely integrated with Eclipse's Java Development Tools (JDT), including features like call-hierarchies, rename refactoring, debugging and many more.

1.1 Installation

Xtend requires Eclipse 3.5 or higher and a Java SDK 5 or higher. The easiest way to install the SDK is via Eclipse Marketplace. But there's also a complete Eclipse distribution available for download at http://xtend-lang.org.

If you don't want to use the recommended Eclipse Plug-in, you can compile Xtend code using the Maven plug-in (§1.1.1).

1.1.1 Maven Support

The runtime library (§1.1.2) as well as a plug-in to run the compiler in a Maven build can be be obtained from the following maven repository: http://build.eclipse.org/common/xtend/maven/.

Here's the XML for the repository:

```
<repositories>
  <repository>
    <id>xtend</id>
    <url>http://build.eclipse.org/common/xtend/maven/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>xtend</id>
    <url>http://build.eclipse.org/common/xtend/maven/</url>
  </pluginRepository>
</pluginRepositories>
```

Here's the XML for the dependency to the library:

```
<dependency>
  <groupId>org.eclipse.xtend</groupId>
  <artifactId>org.eclipse.xtend.lib</artifactId>
  <version>2.3.0</version>
</dependency>
```

And this is the XML for the plug-in:

```
<plugin>
  <groupId>org.eclipse.xtend</groupId>
  <artifactId>xtend-maven-plugin</artifactId>
  <version>2.3.0</version>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <!-- <goal>testCompile</goal> -->
      </goals>
      <!-- optionally you can configure a different target folder -->
      <!--
      <configuration>
        <outputDirectory>xtend-gen</outputDirectory>
```

```
      </configuration>
      -->
    </execution>
  </executions>
</plugin>
```

As you see the *outputDirectory* can be specified to match the default of the Eclipse plug-in (*xtend-gen*). Of course you can also change the configuration in Eclipse to match the Maven default (*generated-sources*). To do so right-click on the project and select *Properties* or if you prefer a global setting choose *Eclipse->Preferences*. In the category *Xtend/Compiler* enter the directory name (see screenshot). It's interpreted as a relative path to the parent of the source folder, which includes the to-be-compiled Xtend file.

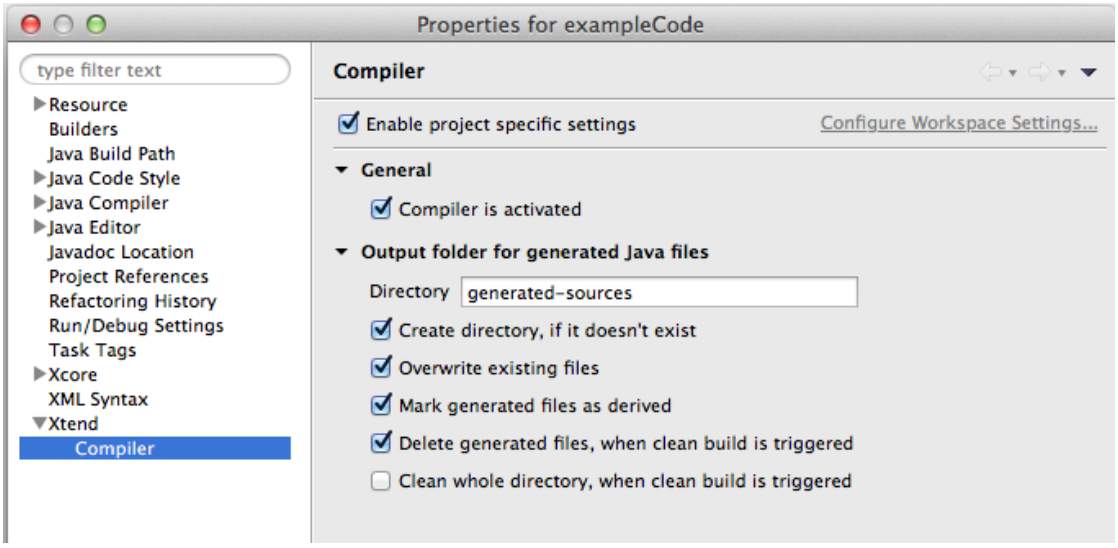

Figure 1.1: Configuring the compiler in Eclipse

1.1.2 The Runtime Library

The compiler requires a small runtime library on the classpath, which mainly provides useful extensions to existing classes and interfaces from the JDK. The class InputOutput providing the method <T> T println(T obj) being one of them.

The only surprising fact in the generated Java code may be the library class . Many features of Xtend are not built into the language itself but provided via the library *org.eclipse.xtend.lib*. The library is available from a Maven repository (§1.1.1) and via p2 update site (in case you do Eclipse Plug-In development).

The library 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 = listOfPersons.map[ name ]
```

Many operators to concat collections or to do arithmetics with types like BigDecimal are also available.

You might want to have a look at the JavaDoc API to see what's available.

1.2 Getting Started

The first thing you want to see in any language is the ubiquitous "Hello World" example. In Xtend, that reads as

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

You see that Xtend looks a lot like Java. At a first glance the main difference seems to be the **def** keyword for declaring a method. Also like in Java it's mandatory to define a class and a main method as the entry point for an application. Admittedly 'hello world' programs are not a particular strength of Xtend. The real expressiveness unleashes as soon as you do real stuff as you will learn in a moment.

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

```
// Generated Java Source Code
import org.eclipse.xtext.xbase.lib.InputOutput;

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

You can put an Xtend class in source folders of any Java project within Eclipse (or any Maven project). Eclipse will complain about the missing library if it's not on the classpath and provide a quick fix to add it.

The next thing you might want to do is materializing one of the example projects into your workspace. To do so right click anywhere in the *Navigator* view in Eclipse and select *New -> Example...*.

In the upcoming dialog you'll find two examples for Xtend:

- *Xtend Introductory Examples* contains a couple of example code snippets illustrating certain aspects and strengths of Xtend. It for instance shows how to build a builder API or an API which allows for writing code like this:

  ```
  assertEquals(42.km/h, (40_000.m + 2.km) / 60.min)
  ```

  Also the the movies example (§1.2.1) explained in detail in the next section (§1.2.1) is included here.

- *Xtend Solutions For Euler* contains solutions to a lot of the problems you'll find at Project Euler's website. These examples are leveraging the whole expressive power of Xtend. For instance Euler Problem 1 can be solved with the following expression :

  ```
  (1..999).filter[ i | i % 3 == 0 || i % 5 == 0].reduce[i1, i2 | i1 + i2]
  ```

1.2.1 The Movies Example (src/examples6/Movies.xtend)

The movies example is included in the example project *Xtend Introductory Examples* and is about reading a file with data about movies in and doing some analysis on it.

The Data

The movie database is a plain text file (data.csv) with data sets describing movies. Here's an example data set:

```
Naked Lunch 1991 6.9 16578 Biography Comedy Drama Fantasy
```

The values are separated by two spaces. The columns are :

1. title
2. year
3. rating
4. numberOfVotes
5. categories

Let's define a data type Movie representing a data set:

```
@Data class Movie {
  String title
  int year
  double rating
  long numberOfVotes
  Set<String> categories
}
```

It's a plain class with a typed field for each column in the data sets. The @Data (§5.2) annotation will turn this class into a value object, that is it will get

- a getter-method for each field,
- a hashCode()/equals() implementation,
- implementation of Object.toString(),
- a constructor accepting values for all fields in the declared order.

Parsing The Data

Let's now add another class to the same file (any number of classes (§3) per file is allowed) and initialize a field called movies with a list of movies. For the initialization we read in the text file and turn the data sets into Movies:

```
import java.io.FileReader
import java.util.Set
import static extension com.google.common.io.CharStreams.*

class Movies {

  val movies = new FileReader('data.csv').readLines.map[ line |
    val segments = line.split(' ').iterator
    return new Movie(
      segments.next,
      Integer::parseInt(segments.next),
      Double::parseDouble(segments.next),
      Long::parseLong(segments.next),
      segments.toSet
    )
  ]
}
```

A field's type (§3.5) can be inferred from the expression on the right hand-side. That's called local type inference and is supported everyhwere in Xtend. We want the field to be final, so we declare it as a value using the keyword **val**.

The initialization on the right hand side first creates a fresh instance of java.io.FileReader . Then the method readLines() is invoked on that instance. But if you have a look at FileReader you won't find such a method, because readLines() is in fact a static method from Google Guava's CharStream which was imported as an extension (§3.8.3), which allows us to use this readable syntax.

```
import static extension com.google.common.io.CharStreams.*
```

CharStream.readLines(Reader) returns a List<String> on which we call another extension method called map. That one is defined in the runtime library (§1.1.2) (ListExtensions. map(...) and is always imported and therefore automatically available on all lists. The

map extension expects a function as a parameter. Map basically invokes that function for each value in the list and returns another list containing the results of the function incovations.

Function objects are created using lambda expression (§4.8) (the code in squared brackets). Within the lambda we process a single line from the text file and turn it into a movie by splitting the string using the separator (two whitespaces) and calling iterator on the result. As you might know java.lang.String.split(String) returns a string array (String[]), which Xtend auto-convertes to a list (§2.2) when we call Iterable.iterator() on it.

```
val segments = line.split('  ').iterator
```

Now we use the iterator to create an instance of Movie. The data type conversion (e.g. String to **int**) is done by calling static methods (§4.6.3) from the wrapper types. The rest of the iterable is turned into a set using the extension method Iterators.toSet(Iterator<T>) and contains all the categories the movie is associated with.

```
return new Movie (
  segments.next,
  Integer::parseInt(segments.next),
  Double::parseDouble(segments.next),
  Long::parseLong(segments.next),
  segments.toSet
)
```

Answering som Questions

Now that we've the text file turned into a List<Movie>, we are ready to execute some queries against it. We use *Junit* to make the individual analysis executable.

What's The Number Of Action Movies?

```
@Test def numberOfActionMovies() {
  assertEquals(828,
    movies.filter[categories.contains('Action')].size)
}
```

First the movies are filtered. The lambda expression checks whether the current movie's categories contains the entry 'Action'. Note that unlike the lambda we used to turn the lines in the file into movies, we haven't declared a parameter name this time. We could have written

```
movies.filter[ movie | movie.categories.contains('Action')].size
```

but since we left out the name and the vertical bar the variable is automatically named **it** which (like **this**) is an implicit variable (§4.6.2). That's why we can write either

```
movies.filter[ it.categories.contains('Action')].size
```

or

```
movies.filter[categories.contains('Action')].size
```

.

Lastly we call size on the resulting iterable, which also is an extension method defined in IterableExtensions.

Question 2: What's The Year The Best Movie From The 80ies Was Released?

```
@Test def void yearOfBestMovieFrom80ies() {
  assertEquals(1989,
    movies.filter[(1980..1989).contains(year)].sortBy[rating].last.year)
}
```

Here we filter all movies where the year is not included in the range from 1980 to 1989 (the 80ies). The .. operator again is an extension defined in IntegerExtensions and returns an instanceof IntegerRange. Operator overloading is explained in section (§4.3).

The resulting iterable is sorted (IterableExtensions.sortBy) by the rating of the movies. Since it's sorted in ascending order, we take the last movie from the list and return its year.

We could have sorted descending and take the head of the list as well:

```
movies.filter[(1980..1989).contains(year)].sortBy[-rating].head.year
```

Note that first sorting and then taking the last or first is slightly more expnsive than needed. We could have used the method reduce instead to find the best movie, which would be more efficient. Maybe you want to try it on your own?

The calls to movie.year as well as movie.categories in the previous example in fact access the corresponding getter methods (§4.6.1).

Question 3: What's The The Sum Of All Votes Of The Top Two Movies?

```
@Test def void sumOfVotesOfTop2() {
  val long sum = movies.sortBy[-rating].take(2).map[numberOfVotes].reduce[a, b| a + b]
  assertEquals(47_229L, sum)
}
```

First the movies are sorted by rating, then we take the best two. Next the list of movies is turned into a list of their numberOfVotes using the map function. Now we have a List<Long> which can be reduced to a single Integer by adding the values.

You could also use reduce instead of map and reduce. Do you know how?

2 Static Typing and Java Interoperability

Xtend, like Java, is a statically typed language. In fact it completely supports Java's type system, including the primitive types as **int** or **boolean**, arrays and of course all classes, interfaces, enums and annotations that reside on the classpath.

Java Generics are fully supported as well: You can define type parameters on methods and classes and pass type arguments to generic types just as you are used to from Java. The type system and its conformance and casting rules are implemented after the Java Language Specification.

2.1 Type Inference

One of the problems with Java is, that you are forced to write type signatures over and over again. That's why so many people don't like static typing. This is in fact not a problem of static typing but simply a problem with Java. Although Xtend is typed just like Java, you rarely have to write types down because they can be computed from the context.

2.2 Conversion Rules

In addition to Java's autoboxing to convert primitives to their corresponding wrapper types (e.g. int is automatically converted to Integer when needed), there are additional conversion rules.

Arrays are automatically converted to java.util.List<ComponentType> and vice versa. That is you can write the following:

```
def toList(String[] array) {
  val List<String> asList = array
  return asList
}
```

Another very useful conversion applies to lambda expressions. As explained in the previous section, a lambda expression usually is of one of the types listed in org.eclipse.xtext.xbase.lib.Functions or org.eclipse.xtext.xbase.lib.Predicates. However if the expected type is an interface which has just method declaration, a lambda expression is automatically converted to that type. This allows to use lambda expressions with many existing Java libraries. See section subsection 4.8.1 for more details.

2.3 Interoperability with Java

Resembling and supporting every aspect of Java's type system, ensures that there is no impedeance mismatch between Java and Xtend. This means that Xtend and Java are 100% interoperable and that there are no special cases and no thinking in two worlds is neccessary. You can call Xtend code from Java and vice versa without any surprises or hassles.

As a bonus if you know Java's type system (specifically generics), you already know the most complicated part of Xtend.

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. 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 look like in Java. There are two small differences:

- An identifier can be escaped with a ^ character in case it conflicts with a keyword.
- The terminating semicolon is optional.

```
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 terminating semicolon is optional. Xtend also features static imports but allows only a wildcard * at the end, i.e. you currently 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 **public** explicitly.

Since version 2.3, multiple class declaration per file are supported. Each of these classes is compiled to a separate top-level Java class.

Abstract classes are defined using the **abstract** modifier as in Java. See also subsection 3.6.1 on abstract methods.

Xtend's approach to inheritance is conceptionally the same as in Java. Single inheritance of classes as well as implementing multiple interfaces is supported. Xtend classes can of course 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 generic 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 you don't have to repeat the name of the class over and over again, but use keyword *new* 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

A field can have an initializer. Final fields are declared using **val**, while **var** introduces a non-final field and can be ommitted. Yet, if an initializer expression is present, the type of a field can be skipped after **val** and **var**. Fields marked as **static** will be compiled to static Java fields.

```
class MyClass {
  int count = 1
  static boolean debug = false
  var name = 'Foo' // type String is inferred
  val UNIVERSAL_ANSWER = 42 // final field with inferred type int
  ...
}
```

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 in section (§3.6.6).

```
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')
}
```

As in Java 5, Xtend allows vararg parameters:

```
def printAll(String... strings) {
  strings.forEach[s | println(s)]
}
```

3.6.1 Abstract Methods

An abstract method in Xtend just does not define a body and must be declared within an **abstract** class. Also specifying the return type is mandatory since it cannot be inferred.

```
abstract class MyAbstractClass() {
    def String abstractMethod() // no body
}
```

3.6.2 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)
}
```

3.6.3 Declared Exceptions

Xtend does not force you to catch or redeclare checked exceptions. Nevertheless, you can still declare the exceptions thrown in a method's body using the same **throws** clause as in Java.

If you don't declare checked exceptions in your method but they are possibly thrown in your code, the compiler will rethrow the checked exception silently (using the sneaky-throw technique introduced by Lombok).

```
/*
 * throws an Exception
 */
def void throwException() throws Exception {
   throw new Exception
}

/*
 * throws an Exception without declaring it
```

```
 */
def void sneakyThrowException() {
    throw new Exception
}
```

3.6.4 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.

3.6.5 Generic Methods

You can specify type parameters just like in Java. To generalize the method from the previous section, you would declare it like this:

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

Also bounds and the like are supported and share the same syntax as defined in the the Java Language Specification

3.6.6 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.

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 charing the same name and the same number of arguments, the compiler infers a synthetic method (the dispatcher) using the common super type of all declared arguments. The actual dispatch methods are reduced in visibility and renamed (prepending an underscore) so that client code always binds to the dispatcher method.

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

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**.

The comparism of the type parameters goes from left to right. That is in the following example, the second method declaration is considered more specific since its first parameter type is the most specific:

```
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
      && y != null) {
    return _printTypes((Integer)x, y);
```

```
  } else if (x != null
        && 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 dispatch case 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"
}
```

This compiles to the following Java code:

```
public String printType(final Number x) {
  if (x instanceof Integer) {
    return _printType((Integer)x);
  } else if (x != null) {
    return _printType(x);
  } else if (x == 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 if (it != null) {
    return super._label(it);
  } else {
    throw new IllegalArgumentException("Unhandled parameter types: " +
      Arrays.<Object>asList(it).toString());
  }
}
```

Static Dispatch Methods

Also static dispatch methods are supported. But you cannot mix static and non-static dispatch methods.

3.7 Annotations

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

```
@TypeAnnotation("some value")
class MyClass {
```

```
  @FieldAnnotation(children = {@MyAnno(true), @MyAnno(false)})
  String myField

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

Certain annotations defined in the library have a special effect on how the code is translated to Java. These annotations are explained in section (§5).

3.8 Extension Methods

Extension methods allow to add new methods to existing types without modifying them. 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 toFirstUper("hello")
```

Method calls in extension syntax often result in more readable code, as method calls are chained rather than nested. Another benefit of extensions is that you can add methods which are specific to a certain context (read layer) only. You might for instance not want to put UI-specific methods and dependencies to your domain model classes. Therefore such functionality is often defined in static methods or methods in some "service class". That works, but the code is less readable and object-oriented if you call methods like this. In Java for instance you often see code like this:

```
persistenceManager.save(myObject);
```

With extension methods you can write it like so:

```
myObject.save
```

There are different ways to make methods available as extensions, which are described in the following.

3.8.1 Extensions From The Library

The library (§1.1.2) puts a lot of very useful extension methods on existing types from the Java SDK without any further ado.

```
"hello".toFirstUpper // calls StringExtensions.toFirstUpper(String)
listOfStrings.map[toUpperCase] // calls ListExtensions.<T, R>map(List<T> list, Function<? super T, ? extends R> m
```

Have a look at the Java doc to see what's there:

- ObjectExtensions

- IterableExtensions
- MapExtensions
- ListExtensions
- CollectionExtensions
- BooleanExtensions
- IntegerExtensions
- FunctionExtensions

3.8.2 Local Extension Methods

All visible non-static methods of the current class and its super types are automatically available as extensions. For example

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

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

Local static methods have to be made available through an import like any other static method.

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())
```

3.8.4 Extension Fields

By adding the **extension** keyword to a field declaration, its instance methods become extension methods.

Imagine you want to have some layer specific functionality on a class Person. Let's say you are in a servlet-like class and want to persist a Person using some persistence mechanism. Let's assume Person implements a common interface Entity.

You could have the following interface

```
interface EntityPersistence {
  public save(Entity e);
  public update(Entity e);
  public delete(Entity e);
}
```

And if you have obtained an instance of that type (through a factory or dependency injection or what ever) like this:

```
class MyServlet {
  extension EntityPersitence ep = Factory.get(typeof(EntityPersistence))
  ...

}
```

You are able to save, update and delete any entity like this:

```
val Person person = ...
person.save // calls ep.save(person)
person.name = 'Horst'
person.update // calls ep.update(person)
person.delete // calls ep.delete(person)
```

Using the **extension** modifier on fields has a significant advantage over static extension imports (§3.8.3): Your code is not bound to the actual implementation of the extension method. You can simply exchange the component that provides the referenced extension with another implementation from outside, by providing a different instance. No matter you do so via a factory, dependency injection or simply using a setter.

4 Expressions

Xtend there are no statements. Instead, everything is an expression and has a return value. That allows to compose your code in intersting ways. For example, you can have a **try catch** expression on the right hand side of an assignment:

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

If fileContentsToString() throws an IOException, it is caught and the string 'dummy data' is assigned to the value data.

Expressions can appear as initializers of fields (§3.5), the body of constructors or methods and as values in annotations. A method body can either be a block expression (§4.4) or a template expression (§4.17).

4.1 Literals

A literal denotes a fixed unchangeable value. Literals for strings (§4.1.1), numbers (§4.1.2), booleans (§4.1.3), **null** and Java types (§4.1.5) are supported.

4.1.1 String Literals

A string literal is of type java.lang.String (just like in Java). String literals are enclosed by a pair of single quotes or double quotes. We mostly use single quotes because the signal-to-noise ration is a bit better, but generally you should use the terminals which are least likely occure in the actual string. Special characters can be quoted with a backslash or defined using Java's unicode notation. Contrary to Java, strings can span multiple lines.

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

  World !"
```

4.1.2 Number Literals

Xtend supports roughly the same number literals as Java with a few differences. First, there are no signed number literals. If you put a minus operator in front of an number literal it is taken as a UnaryOperator (§4.3) with one argument (the positive number literal). Second, as in Java 7, you can separate digits using _ for better readability of large numbers.

An integer literal creates an **int**, a **long** (suffix L) or a BigInteger (suffix BI). There are no octal numbers

```
42
1_234_567_890
0xbeef // hexadecimal
077 // decimal 77 (*NOT* octal)
-1 // an expression consisting of the unary - operator and an integer literal
42L
0xbeef#L // hexadecimal, mind the '#'
0xbeef_beef_beef_beef_beef#BI // BigInteger
```

A floating-point literal creates a **double** (suffix D or none), a **float** (suffix F) or a BigDecimal (suffix BD). If you use a . you have to specify both, the integer and the fractional part of the mantissa. There are only decimal floating-point literals.

```
42d // double
0.42e2 // implicit double
0.42e2f // float
4.2f // float
0.123_456_789_123_456_789_123_456_789e2000bd // BigDecimal
```

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** has exactly the same semantics as in Java.

4.1.5 Type Literals

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

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

4.2 Type Casts

A type cast behaves exactly like casts in Java, but has a slightly more 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
42 as Integer
```

Although casts are supported you might want to use a switch with a type guard (§4.10) as a better and safer alternative.

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 BigDecimals which allows the following code:

```
val x = 2.71BD
val y = 3.14BD
val sum = x + y // calls BigDecimalExtension.operator_plus(x,y)
```

This is the complete list of all available operators and their corresponding method signatures.

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_doubleArrow(e2)
e1 << e2	e1.operator_doubleLessThan(e2)
e1 >> e2	e1.operator_doubleGreaterThan(e2)
e1 <<< e2	e1.operator_tripleLessThan(e2)
e1 >>> e2	e1.operator_tripleGreaterThan(e2)
e1 <> e2	e1.operator_diamond(e2)
e1 ?: e2	e1.operator_elvis(e2)
e1 <=> e2	e1.operator_spaceship(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 bound to the library methods BooleanExtensions.operator_and (boolean l, boolean r) resp. BooleanExtensions.operator_or(boolean l, boolean r) the operation is

evaluated in short circuit mode. That 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**.

Still you can overload these operators for your types or even override it for booleans, in which case both operands are always evaluated and the defined method is invoked, i.e. no short-circuit execution is happening.

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 (§4.5) can be reassigned using the = operator.

```
var greeting = 'Hello'
if (isInformal)
  greeting = 'Hi'
```

Of course, also non-final fields can be set using an assignment:

```
myObj.myField = 'foo'
```

Setting Properties

The lack of properties in Java leads to a lot of syntactic noise when working with data objects. As Xtend is designed to integrate with existing Java APIs it respects the Java Beans convention, hence you can call a setter using an assignment:

```
myObj.myProperty = 'foo' // calls myObj.setMyProperty("foo")
```

The setter is only used if the field is not accessible from the given context. That's why the @Property annotation (§5.1) would rename the local field to _myProperty.

The return type of an assignment is the type of the right hand side, in case it's a simple assignment. If it's translated to a setter method it yields whatever the setter method returns.

4.4 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 (§4.5) 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
  }
}
```

4.5 Variable Declarations

Variable declarations are only allowed within blocks (§4.4). 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
  }
}
```

Shadowing variables from outer scopes is not allowed, the only exception is the implicit variable **it** (§4.6.2).

Variables declared outside a lambda expression using the **var** keyword are not accessible from within a lambda expressions.

4.5.1 Typing

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 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> // -> msg is of type ArrayList<String>
```

4.6 Field Access and Method Invocation

A simple name can refer to a local field, variable or parameter. In addition it can point to a method with zero arguments, since empty parenthesis are optional.

4.6.1 Property Access

If there is no field with the given name and also no method with the name and zero parameters accessible, a simple name binds to a corresponding Java-Bean getter method if available :

```
myObj.myProperty // myObj.getMyProperty() (.. in case myObj.myProperty is not visible.)
```

4.6.2 Implicit Variables this and it

Like in Java an instance of the class is bound to **this**. Which allows for either qualifying field access or method invocations like in :

```
this.myField
```

or omit the receiver:

```
myField
```

You can use the variable name **it** to get the same behavior for any variable or parameter:

```
val it = new Person
name = 'Horst' // translates to 'it.setName("Horst");'
```

Another speciality of the variable **it** is that it can be shadowed. This is especially useful when used together with lambda expressions (§4.8).

As **this** is bound to the surrounding object in Java, **it** can be used in finer-grained constructs such as lambda expressions (§4.8). That is why **it**.myProperty has higher precedence than **this**.myProperty.

4.6.3 Static Access

For accessing a static field or method you have to use the double colon :: like in this example:

```
MyClass::myField
MyClass::myMethod('foo')
```

Alternatively you could import the method using a static import (§3.2).

4.6.4 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
```

4.7 Constructor Call

ConstructorCalls have the same syntax as in Java. The only difference is that empty parenthesis are optional:

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

4.8 Lambda Expressions

A lambda expression is basically a piece of code, which is wrapped in an object to pass it around. As a Java developer it's best to think of a lambda expression as an anonymous class, i.e. like in the following Java code :

```
// Java Code!
final JTextField textField = new JTextField();
textField.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    textField.setText("Something happened!");
  }
});
```

This kind of anonymous classes can be found everywhere in Java code and have always been the poor-man's replacement for lambda expressions in Java.

Xtend not only supports lambda expressions, but offers an extremely dense syntax for it. That is the code above can be written in Xtend like this:

```
val textField = new JTextField
textField.addActionListener([ ActionEvent e |
  textField.text = "Something happened!"
])
```

As you might have guessed, a lambda expression is surrounded by square brackets (inspired from Smalltalk). Also a lambda expression like a method declares parameters. The lambda above has one parameter called `e` which is of type `ActionEvent`. You don't have to specify the type explicitly because it can be inferred from the context:

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

Also as lambdas with one parameter are a common case, there is a special short hand for them, which is to leave the declaration including the vertical bar out. The name of the single variable will be `it` (§4.6.2) in that case:

```
textField.addActionListener([
  textField.text = "Something happened!"
])
```

A lambda expression with zero arguments is written like this (note the bar after the opening bracket):

```
val Runnable runnable = [|
  println("Hello I'm executed!")
]
```

When a method call's last parameter is a lambda it can be passed right after the parameters. For instance if you want to sort some strings by their length, you could write :

```
Collections::sort(someStrings) [ a, b |
  a.length - b.length
]
```

which is just the same as writing

```
Collections::sort(someStrings, [ a, b |
  a.length - b.length
])
```

Since you can leave out empty parenthesis methods wich get a lambda as the single argument you can reduce the code above further more:

```
textField.addActionListener[
  textField.text = "Something happened!"
]
```

A lambda expression 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. That's exactly what we did with the variable textField.

4.8.1 Typing

Closures are expressions which produce *Function* objects. The type of a lambda expression generally depends on the target type, as seen in the previous examples. That is, the lambda expression can coerce to any interface which has declared only one method (in addition to the ones inherited from java.lang.Object). This allows for using lambda expressions in many existing Java APIs directly.

However, if you write a lambda expression without having any target type expectation, like in the following assignment:

```
val toUpperCaseFunction = [String s | s.toUpperCase] // inferred type is (String)=>String
```

The type will be one of the types found in Functions) or (Procedures. It's a procedure if the return type is void, otherwise it's a function.

Xtend supports a shorthand syntax for function types. Instead of writing Function1<? super String,? extends String> which is what you'll find in the generated Java code, you can simply write (String)=>String.

Example:

```
val (String)=>String stringToStringFunction = [toUpperCase]
// or
val Function1<? super String,? extends String> same = [toUpperCase]
// or
val stringToStringFunction2 = [String s | s.toUpperCase] // inferred type is (String)=>String
```

Checked exceptions that are thrown in the body of a closure but not declared in the implemented method of the target type are rethrown using the sneaky-throw technique (§3.6.3). Of course you can always catch and handle (§4.16) them.

4.9 If Expression

An if expression is used to choose two different values based on a predicate.

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 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.

While the **if**-expression has the syntax of Java's if statement it behaves more like Java's ternary operator (predicate ? thenPart : elsePart), because it is an expression and returns a value. Consequently, you can use if expressions deeply nested within expressions:

```
val name = if (firstName != null) firstName + ' ' + lastName else lastName
```

4.10 Switch Expression

The **switch** expression is very different from Java's switch statement. 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. Object.equals() is used to compare the value in the case with the one you are switching over.

Given the following example:

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

the main expression numberAsText is evaluated first and then compared to each case sequentially. If the case expression is of type **boolean**, the case matches if the expression evaluates to true. If it's not of type **boolean** it's compared to the value from the main expression using Object.equals(Object).

If a case is a match, that is it evaluates to **true** or the result equals the one we are switching over, the case expression after the colon is evaluated and is the result of the whole expression.

4.10.1 Type guards

Instead if or in addition to the case guard you can have a so called *Type Guard*. That is syntactically just a type reference (§2). The cool thing is that the value is automatically down casted in subsequent expressions. Example:

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

Only if the switch value is an instance of the specified type, 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.

Switches with type guards are a safe and much more readable alternative to instance of / casting orgies you might know from Java.

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 canot 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

A method or lambda expression automatically returns the value of it's expression. If it's a block expression (§4.4) this is the value of the last expression in it. However, sometimes you want to return early or make it explicit.

The syntax is just like in Java:

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

4.15 Throwing Exceptions

Throwing java.lang.Throwables up the call stack has the same semantics and syntax 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 catch checked exceptions, if you do not catch checked exceptions they are rethrown in a wrapping runtime exception if not declared in the method's signature. 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
}
```

For try-catch it's again beneficial that it is an expression, because you can write code like the following and don't have to rely on non-final variables:

```
val name = try {
    person.name
  } catch (NullPointerException e) {
    "no name"
  }
```

4.17 Template Expressions

Templates allow for readable string concatenation. Templates are surrounded by triple single quotes (''). A template expression can span multiple lines and expressions can be nested which are evaluated and their toString() representation is automatcially inserted at that position.

The terminals for interpolated expression are so called guillemets «expression». They read nicely and are not often used in text so you seldomly need to escape them. These escaping conflicts are the reason why template languages often use longer character sequences like e.g. <%= expression %> in JSP, for the price of worst readability. The downside with the guillemets in Xtend is that you'll have to use a proper encoding (Always use UTF-8 and you are good).

If you use the Eclipse plug-in which is recommended, the guillemets will be inserted automatically using content assist. They are also bound to *CTRL+SHIFT+<* and *CTRL+SHIFT+* for « and » respectively. On a mac they are bound to *alt+q* («) and *alt+Q* (»).

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

```
def someHTML(String content) '''
  <html>
    <body>
      «content»
    </body>
  </html>
'''
```

As you can see, template expressions can be used as the direct body of a method. If an interpolation expression evaluates to **null** an empty string is added.

Template expressions can occur everywhere. Here's an example showing it in conjunction with the powerful switch expression (§4.10):

```
def toText(Node n) {
  switch n {
    Contents : n.text

    A : '''<a href="«n.href»">«n.applyContents»</a>'''

    default : '''
        <«n.tagName»>
          «n.applyContents»
        </«n.tagName»>
    '''
  }
}
```

4.17.1 Conditions in Templates

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

```
def someHTML(Paragraph p) '''
  <html>
    <body>
      «IF p.headLine != null»
        <h1>«p.headline»</h1>
      «ENDIF»
      <p>
        «p.text»
      </p>
    </body>
  </html>
'''
```

4.17.2 Loops in Templates

Also a **FOR** statement is available:

```
def someHTML(List<Paragraph> paragraphs) '''
  <html>
    <body>
      «FOR p : paragraphs»
        «IF p.headLine != null»
          <h1>«p.headline»</h1>
        «ENDIF»
        <p>
          «p.text»
        </p>
      «ENDFOR»
    </body>
```

```
  </html>
'''
```

The for statement optionally allows to specify what to prepend (**BEFORE**), put in-between (**SEPARATOR**), and what to put at the end (**AFTER**) of all iterations. **BEFORE** and **AFTER** are only executed if there is at least one iteration. (**SEPARATOR**) is only added between iterations, that it is executed if there are at least two iterations.

Here's an example:

```
def someHTML(List<Paragraph> paragraphs) '''
  <html>
    <body>
      «FOR p : paragraphs BEFORE '<div>' SEPARATOR '</div><div' AFTER '</div>'»
        «IF p.headLine != null»
          <h1>«p.headline»</h1>
        «ENDIF»
        <p>
          «p.text»
        </p>
      «ENDFOR»
    </body>
  </html>
'''
```

4.17.3 Typing

The template expression is of type CharSequence but auto-convertes to String if that is the expected target type.

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 template is evaluated.

1. 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.
2. 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.
3. Any newlines in appended strings (no matter they are created with template expressions or not) will be prepended with the current indentation when inserted.

Although this algorithm sounds a bit complicated at first it behaves very intuitively. In addition the syntax coloring in Eclipse communicates this behavior.

```
def someHTML(List<Paragraph> paragraphs) '''
  <html>
    <body>
      «FOR p : paragraphs BEFORE '<div>' SEPARATOR '</div><div' AFTER '</div>'»
        «IF p.headline != null»
          <h1>«p.headline»</h1>
        «ENDIF»
        <p>
          «p.text»
        </p>
      «ENDFOR»
    </body>
  </html>
'''
```

Figure 4.1: Syntax Coloring For Templates In Eclipse

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

```
class Template {
  def 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 {
  def 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 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.

5 Processed Annotations

Xtend comes with annotations that help to steer the compilation process. These annotations reside in the org.eclipse.xtend.lib plug-in/jar which must be on the classpath of the project containing the Xtend files.

5.1 @Property

For fields that are annotated as @Property, the Xtend compiler will generate a Java field, a getter and, if the field is non-final, a setter method. The name of the Java field will be prefixed with an _ and have the visibility of the Xtend field. The accessors methods are always **public**. Thus, an Xtend field

```
@Property String name
```

will compile to the Java code

```
private String _name;

public String getName() {
  return this._name;
}

public void setName(final String name) {
  this._name = name;
}
```

5.2 @Data

The annotation @Data, will turn an annotated class into a value object class. A class annotated with @Data has the following effect:

- all fields are flagged final,
- getter methods will be generated (if not existent),
- a constructor taking paramaters for all non-initialized fields will be generated (if not existent),
- equals(Object) / hashCode() methods will be generated (if not existent),
- a toString() method will be generated (if not existent).

Example:

```
@Data class Person {
  String firstName
  String lastName
}
```

List of External Links

http://build.eclipse.org/common/xtend/maven/
http://docs.oracle.com/javase/specs/jls/se5.0/html/conversions.html#5.5
http://marketplace.eclipse.org/content/eclipse-xtend
http://docs.oracle.com/javase/specs/jls/se5.0/html/classes.html#8.4.4
http://projecteuler.net/problem=1
http://www.eclipse.org/xtend/index.html#download
http://projecteuler.net/
http://projectlombok.org/features/SneakyThrows.html
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
http://docs.oracle.com/javase/specs/jls/se5.0/html/conversions.html
http://xtend-lang.org/api/2.3.0/index.html

Todo list