Xtend User Guide

December 10, 2013

Contents

Part I.

Getting Started

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 (§5.8)* - enhance closed types with new functionality
- *Lambda Expressions (§6.8)* - concise syntax for anonymous function literals
- *ActiveAnnotations (§7)* - annotation processing on steroids
- *Operator overloading (§6.3)* - make your libraries even more expressive
- *Powerful switch expressions (§6.10)* - type based switching with implicit casts
- *Multiple dispatch (§5.6.6)* - a.k.a. polymorphic method invocation
- *Template expressions (§6.17)* - with intelligent white space handling
- *No statements (§6)* - everything is an expression
- *Properties (§6.6.1)* - shorthands for accessing and defining getters and setter
- *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

Unlike other JVM languages Xtend has zero interoperability issues (§4.3) with Java: Everything you write interacts with Java exactly as expected. At the same time Xtend is much more concise, readable and expressive. Xtend's small library is just a thin layer that provides useful utilities and extensions on top of the Java Development Kit (JDK).

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 the Eclipse Java Development Tools (JDT), including features like call-hierarchies, rename refactoring, debugging and many more.

2. Hello World

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

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

You see that Xtend looks quite similar to Java. At a first glance the main difference seems to be the def keyword to declare a method. Also like in Java it is 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 is unleashed when you do real things as you will learn in a second.

An Xtend class resides in a plain Eclipse Java project. As soon as the SDK is installed, Eclipse will automatically translate all the classes to Java source code. By default you will find it in a source folder *xtend-gen*. 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");
  }
}
```

The only surprising fact in the generated Java code may be the referenced library class InputOutput. It is part of the runtime library and a nice utility that is quite handy when used in expressions.

You can put an Xtend class into a source folder of any Java project within Eclipse or any Maven project. If the project is not yet configured properly, Eclipse will complain about the missing library. The xtend.lib has to be on the class path. The IDE will 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. Right click anywhere in the *Navigator* view in Eclipse and select *New -> Example...*.

In the upcoming dialog you will find two examples for Xtend:

- *Xtend Introductory Examples* contains a couple of example code snippets illustrating certain aspects and strengths of Xtend. For instance it shows how to build an API which allows to write code like this:

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

  Also the movies example (§3) explained in detail in the next section (§3) is included there.

- *Xtend Solutions For Euler* contains solutions to some of the problems you will find at Project Euler. These examples are leveraging the whole expressive power of Xtend. For instance Euler Problem 1 can be solved with this expression :

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

3. The Movies Example

The movies example is included in the example project *Xtend Introductory Examples* (src/examples6/Movies.xtend) and is about reading a file with data about movies and doing some analysis on it.

3.1. The Data

The movie database is a plain text file (data.csv) with data sets describing movies. Here is 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 us define a data type Movie representing a data set:

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

A movie is a POJO with a strongly typed field for each column in the data sets. The @Data (§7.7) annotation will turn the class into an immutable value class, 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.

3.2. Parsing The Data

Let us now add another class to the same file and initialize a field called movies with a list of movies. For the initialization we parse the text file and turn the data records 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 (§5.5) can be inferred from the expression on the right hand-side. That is called local type inference and is supported everywhere 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 new FileReader. Then the method readLines() is invoked on that instance. But if you have a look at FileReader you will not find such a method. In fact readLines() is a static method from Google Guava's CharStreams which was imported as an extension (§5.8.3). Extensions allow us to use this readable syntax.

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

CharStreams.readLines(Reader) returns a List<String> on which we call another extension method map. This one is defined in the runtime library (ListExtensions.map(...)) and is automatically imported and therefore available on all lists. The map extension expects a function as a parameter. It basically invokes that function for each value in the list and returns another list containing the results of the function invocations. Actually this mapping is performed lazily so if you never access the values of the result list, the mapping function is never executed.

Function objects are created using lambda expressions (§6.8) (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 two whitespace characters as the separator. On the result of the split operation, the method iterator() is invoked. As you might know String.split(String) returns a string array (String[]), which Xtend auto-converts to a list (§4.2) when we call Iterable.iterator() on it.

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

Now we use the iterator to create an instance of Movie for each String that it yields. The data type conversion (e.g. String to int) is done by calling static methods (§6.6.3) from the wrapper types. The rest of the Iterable is turned into a set of categories. Therefore, the extension method IteratorExtensions.toSet(Iterator<T>) is invoked on the iterator to consume its remaining values.

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

3.3. Answering Some Questions

Now that we have parsed the text file into a List<Movie>, we are ready to execute some queries against it. We use *JUnit* to make the individual queries executable and to confirm their results.

3.3.1. Question 1 : What Is 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 contain the entry 'Action'. Note that unlike the lambda we used to turn the lines in the file into movies, we have not 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. it is an implicit variable (§6.6.2). It's uses are similar to the implicit variable this. We can write either

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

or even more compact

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

Eventually we call size on the resulting iterable which is an extension method, too. It is defined in the utility class IterableExtensions.

3.3.2. Question 2 : What Is The Year The Best Movie From The 80's Was Released?

```
@Test def void yearOfBestMovieFrom80s() {
  assertEquals(1989,
    movies.filter[ (1980..1989).contains(year) ].sortBy[ rating ].last.year)
}
```

Here we filter for all movies whose year is included in the range from 1980 to 1989 (the 80's). The .. operator is again an extension defined in IntegerExtensions and returns an instance of IntegerRange. Operator overloading is explained in section (§6.3).

The resulting iterable is sorted (IterableExtensions.sortBy) by the rating of the movies. Since it is 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
```

Another possible solution would be to reverse the order of the sorted list:

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

Note that first sorting and then taking the last or first is slightly more expensive 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 (§6.6.1).

3.3.3. Question 3 : What Is 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 Long by adding the values.

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

Part II.

Reference Documentation

4. Java Interoperability

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

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 as defined in the Java Language Specification.

4.1. Type Inference

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

4.2. 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 in Xtend.

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

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

Subsequent changes to the array are reflected by the list and vice versa. Arrays of primitive types are converted to lists of their respective wrapper types.

The conversion works the other way round, too. In fact, all subtypes of Iterable are automatically converted to arrays on demand.

Another very useful conversion applies to lambda expressions. A lambda expression usually is of one of the types declared in Functions or Procedures. However, if the

expected type is an interface or a class with a single abstract method declaration, a lambda expression is automatically converted to that type. This allows to use lambda expressions with many existing Java libraries. See subsection 6.8.1 for more details.

4.3. Interoperability with Java

Resembling and supporting every aspect of Java's type system ensures that there is no impedance mismatch between Java and Xtend. This means that Xtend and Java are 100% interoperable. There are no exceptional cases and you do not have to think in two worlds. You can invoke Xtend code from Java and vice versa without any surprises or hassles.

As a bonus if you know Java's type system and are familiar with Java's generic types, you already know the most complicated part of Xtend.

5. 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 class definitions. The classes in fact are directly translated to Java classes in the corresponding Java package. A class can have constructors, fields, methods and annotations.

Here is an examplary Xtend file:

```
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)
  }
}
```

5.1. Package Declaration

Package declarations can look like those in Java. Two small, optional differences:

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

```
package com.acme
```

5.2. 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 cannot import single members using a static import. Non-static wildcard imports are deprecated for the benefit of better usability and well defined dependencies.

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

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

Static methods can also be imported as extensions. See the section on extension methods (§5.8) for details.

5.3. Class Declaration

The class declaration reuses a lot of Java's syntax but still is a bit different in some aspects: All Xtend types are public by default since that's the common case. Java's "package private" default visibility is declared by the more explicit keyword package in Xtend. In contrast to Java, Xtend supports multiple public top level class declarations per file. Each Xtend class is compiled to a separate top-level Java class.

Abstract classes are defined using the abstract modifier as in Java. See also abstract methods (§5.6.1).

Xtend's approach to inheritance is conceptually the same as in Java. Single inheritance of classes as well as implementing multiple interfaces is supported. Xtend classes can extend other Xtend classes, and even Java classes can inherit from Xtend classes. If no super type is specified, Object is used.

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 {
  ...
}
```

5.4. Constructors

An Xtend class can define any number of constructors. Unlike Java you do not have to repeat the name of the class over and over again, but use the 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 an explicit visiblity public, protected, package or private.

5.5. Fields

A field can have an initializer. Final fields are declared using val, while var introduces a non-final field and can be omitted. Yet, if an initializer expression is present, the type of a field can be inferred if val or var was used to introduce the field. The keyword final is synonym to val. 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 for fields is private. You can also declare it explicitly as being public, protected, package or private.

A specialty of Xtend are fields that provide *extension methods* which are covered in their own section (§5.8).

5.6. 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 later (§5.6.6).

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

Method declarations start with the keyword def. The default visibility of a method is public. You can explicitly declare it as being public, protected, package or private.

Xtend supports the static modifier for methods and can infer (§5.6.4) the return type if it is not explicitly given:

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

As in Java, vararg parameters are allowed and accessible as array values in the method body:

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

It is possible to infer the return type of a method from its body. Recursive methods and abstract methods have to declare an explicit return type.

5.6.1. Abstract Methods

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

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

5.6.2. Overriding Methods

Methods can override other methods from the super class or implement 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. The override semantics are the same as in Java, e.g. it is impossible to override final methods or invisible methods. Overriding methods inherit their return type from the super declaration.

Example:

```
override String second(List<String> elements) {
  elements.get(1)
}
```

5.6.3. Declared Exceptions

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

If you do not declare checked exceptions in your method but they are possibly thrown in your code, the compiler will throw 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
}
```

Optional validation of checked exception is supported, too, and can be configured on the respective Eclipse preference page for the Xtend Errors and Warnings.

5.6.4. 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 second(List<String> elements) {
  elements.get(1)
}
```

could be declared like this:

```
def second(List<String> elements) {
  elements.get(1)
}
```

The return type is mandatory for abstract method declarations as well as for recursive implementations.

5.6.5. Generic Methods

You can specify type parameters on methods. A parameterized variant of the method from the previous section, could look like this:

```
def <T> second(List<T> elements) {
  elements.get(1)
}
```

Type parameter bounds and constraints are supported and share the same syntax and semantics as defined in the the Java Language Specification

5.6.6. Dispatch Methods

Generally, method resolution and binding is done statically at compile time as 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 (§5.8) you would like to have polymorphic behavior.

A dispatch method is declared 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 with the same name and the same number of arguments, the compiler infers a synthetic dispatcher method. This dispatcher uses the common super type of all declared arguments. The method name of the actual dispatch cases is prepended with an underscore and the visibility of these methods is reduced to protected if they have been defined as public methods. Client code always binds to the synthesized 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 are handled first.

The default visibility of the dispatch cases 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 comparison of the parameter types is performed 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());
  }
}
```

The code is compiled in a way that a null reference is never a match. null values can be handled by dispatch cases that use the parameter type 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

All visible Java methods from all super types that are conformant to the compiled representation of a dispatch method are also included in the dispatcher. Conforming means they have the expected number of arguments and have the same compiled name with the prepended 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. A mixture of static and non-static dispatch methods is prohibited.

5.7. Annotations

Annotations are available on classes, fields, methods and parameters. They are prefixed with the @ character and accept a number of key-value pairs or a single default value for the annotation property named value. Annotation values that expect arrays can handle single values, too. Value arrays are enclosed in array literals #['first', 'second']. The semantics for annotations are exactly like defined in the Java Language Specification. Here is an example:

```
@TypeAnnotation("some value")
class MyClass {
  @FieldAnnotation(value = @NestedAnnotation(true))
  static val CONSTANT = 'a compile-time constant'

  @MethodAnnotation(constant = CONSTANT)
  def String myMethod(@ParameterAnnotation String param) {
    //...
  }
}
```

In addition Active Annotations (§7) allow users to participate in compilation of Xtend code to Java source code.

5.8. 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 method invocation, the method can be called with the first argument as its receiver - it can be called as if the method was one of the argument type's members.

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

Method calls in extension syntax often result in more readable code, as they are chained rather than nested. Another benefit of extensions is that you can add methods which are specific to a certain context or layer of your application.

For instance, you might not want to put UI-specific methods and dependencies into your domain model classes. Therefore this functionality is often defined in static methods or methods in utility classes or service layers. That works, but the code is less readable and less object-oriented if you call methods like this. In Java you often see code like this:

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

Without tying your entities to the persistenceManager, extension methods allow you to write

```
myObject.save
```

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

5.8.1. Extensions from the Library

The Xtend library 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>
```

Have a look at the JavaDoc to learn about the available functionality:

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

5.8.2. 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.

5.8.3. 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 (§5.2) thus making all imported static functions available as extensions methods.

The following import declaration

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

allows us to use its methods like this:

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

5.8.4. Extension Provider

By adding the extension keyword to a field, a local variable or a parameter declaration, its instance methods become extension methods.

Imagine you want to have some layer specific functionality on a class Person. Let us say you are in a servlet-like class and want to persist a Person using some persistence mechanism. Let us 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 EntityPersistence ep = Factory.get(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 values has a significant advantage over static extension imports (§5.8.3): 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.

5.9. Interface Declaration

An interface declaration is very similar to the one in Java. An interface can declare fields, which are by default final static therefore must have an initial value. And of course methods can be declared. They are public by default. Interfaces can extend any number of other interfaces and can declare type parameters.

Here's an example:

```
interface MyInterface<T> extends OtherInterface {
      val CONSTANT = 42
```

```
  def T doStuff(String ... varArg) throws SomeException
}
```

5.10. Annotation Type Declaration

Annotation types can also be declared. The are introduced by the keyword annotation and declare their values with a concise syntax:

```
annotation MyAnnotation {
        String[] value
        boolean isTricky = false
        int[] lotteryNumbers = #[ 42, 137 ]
}
```

5.11. Enum Type Declaration

Enumeration types are declared like this:

```
enum MyColor {
        GREEN,
        BLUE,
        RED
}
```

6. Expressions

In Xtend everything is an expression and has a return type. Statements do not exist. That allows you to compose your code in interesting 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 (§5.5), the body of constructors or methods and as values in annotations. A method body can either be a block expression (§6.4) or a template expression (§6.17).

6.1. Literals

A literal denotes a fixed, unchangeable value. Literals for strings (§6.1.1), numbers (§6.1.3), booleans (§6.1.4), null and Java types (§6.1.6) are supported as well as literals for unmodifiable collection types like lists, sets and maps or literals for arrays.

6.1.1. String Literals

A string literal is of type String. String literals are enclosed in a pair of single quotes or double quotes. Single quotes are more common because the signal-to-noise ration is better, but generally you should use the terminals which are least likely to occur in the string value. Special characters can be quoted with a backslash or defined using unicode notation. Contrary to Java, strings can span multiple lines.

```
'Hello World !'
"Hello World !"
'Hello "World" !'
"Hello \"World\" !"
"Hello  World !"
```

6.1.2. Character Literals

Character literals use the same notation as String literals. If a single character literal is used in a context where a primitive char or the wrapper type Character is expected, the compiler will treat the literal as such a value or instance.

```
val char c = 'c'
```

6.1.3. 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 a number literal it is treated as a unary operator (§6.3) 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 integral 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
```

6.1.4. Boolean Literals

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

6.1.5. Null Literal

The null pointer literal null has exactly the same semantics as in Java.

6.1.6. Type Literals

The syntax for type literals is generally the plain name of the type, e.g. the snippet String is equivalent to the Java code String.class. Nested types use the delimiter '.'.

To disambiguate the expression, type literals may also be specified using the keyword typeof.

- Map.Entry is equivalent to Map.Entry.class
- typeof(StringBuilder) yields StringBuilder.class

Consequently it is possible to access the members of a type reflectively by using its plain name String.getDeclaredFields.

The keyword typeof is mandatory for references to array types, e.g. typeof(int[]

Older versions of Xtend (2.4.1 and before) used the dollar as the delimiter character for nested types and enforced the use of typeof for all type literals:

- typeof(Map$Entry) yields Map.Entry.class

6.1.7. Collection Literals

The methods in CollectionLiterals are automatically imported so it's very easy and convenient to create instances of the various collection types the JDK offers.

```
val myList = newArrayList('Hello', 'World')
val myMap = newLinkedHashMap('a' -> 1, 'b' -> 2)
```

In addition xtend supports collection literals to create immutable collections and arrays, depending on the target type. An immutable list can be created like this:

```
val myList = #['Hello','World']
```

If the target type is an array as in the following example an array is created instead without any conversion:

```
val String[] myArray = #['Hello','World']
```

An immutable set can be created using curly braces instead of the squared brackets:

```
val mySet = #{'Hello','World'}
```

An immutable map is created like this:

```
val myMap = #{'a' -> 1 ,'b' ->2}
```

6.1.8. Arrays

Java arrays can be created either using a literal (§6.1.7) as described in the previous section, or if it should be a new array with a fixed size, one of the methods from ArrayLiterals can be used. The generic newArrayOfSize(int) method works for all reference types, while there is a specific factory method for each primitive type.

Example:

```
val String[] myArray = newArrayOfSize(400)
val int[] intArray = newIntArrayOfSize(400)
```

Retrieving and setting values of arrays is done through the extension methods get(int) and set(int, T) which are specifically overloaded for arrays and are translated directly to the equivalent native Java code myArray[int].

Also length is available as an extension method and is directly translated to Java's equivalent myArray.length.

Furthermore arrays are automatically converted to lists (java.util.List) when needed. This works similar to how boxing and unboxing between primitives and their respective wrapper types work.

Example:

```
val int[] myArray = #[1,2,3]
val List<Integer> myList = myArray
```

6.2. 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
```

Instead of a plain type cast it's also possible to use a switch with a type guard (§6.10) which performs both the casting and the instance-of check. Dispatch methods (§5.6.6) are another alternative to casts that offers the potential to enhance the number of expected and handled types in subclasses.

6.3. Infix Operators and Operator Overloading

There are a couple of common predefined infix operators. These operators are not limited to operations on certain types. Instead an operator-to-method mapping allows to redefine the operators for any type just by implementing the corresponding method signature. As an example, the runtime library contains a class BigDecimalExtensions that defines operators for BigDecimals. The following code is therefore perfectly valid:

```
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_remove(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_tripleEquals(e2)
e1 !== e2	e1.operator_tripleNotEquals(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_greaterThanDoubleDot(e2)
e1 ..< e2	e1.operator_doubleDotLessThan(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()
+ e1	e1.operator_plus()

The table above also defines the operator precedence in ascending order. The blank lines separate precedence levels. The assignment operators += and -= are 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. Parentheses can be used to adjust the default precedence and associativity.

6.3.1. Short-Circuit Boolean Operators

If the operators ||, &&, and ?: are bound to the library methods BooleanExtensions.operator_and(boolean l, boolean r), BooleanExtensions.operator_or(boolean l, boolean r) resp. <T> T operator_elvis(T first, T second) the operation is inlined and 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.
3. in the case of ?: the operand on the right hand side is not evaluated if the left operand evaluates to null.

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.

6.3.2. Defined Operators in The Library

Xtend offers operators for common types from the JDK.

Equality Operators

In Xtend the equals operators (==,!=) are bound to Object.equals. So you can write:

```
if (name == 'Homer')
        println('Hi Homer')
```

Java's identity equals semantic is mapped to the tripple-equals operators === and !== in Xtend.

```
if (someObject === anotherObject)
        println('same objects')
```

Comparison Operators

In Xtend the usual comparison operators (>,<,>=, and <=) work as expected on the primitive numbers:

```
if (42 > myNumber) {
	...
}
```

In addition these operators are overloaded for all instances of java.lang.Comparable. So you can also write

```
if (startTime < arrivalTime)
	println("You are too late!")
```

Arithmetic Operators

The arithmetic operators (+,-,*,/,%, and **) are not only available for the primitive types, but also for other reasonable types such as BigDecimal and BigInteger.

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

Elvis Operator

In addition to null-safe feature calls (§6.6.4) Xtend supports the elvis operator known from Groovy.

```
val salutation = person.firstName ?: 'Sir/Madam'
```

The right hand side of the expression is only evaluated if the left side was null.

With Operator

The with operator is very handy when you want to initialize objects or when you want to use a particular instance a couple of time in subsequent lines of code. It simply passes

the left hand side argument to the lambda on the right hand and returns the left hand after that.

Here's an example:

```
val person = new Person => [
        firstName = 'Homer'
        lastName = 'Simpson'
        address = new Address => [
                street = '742 Evergreen Terrace'
                city = 'SpringField'
        ]
]
```

Range Operators

There are three different range operators. The most useful ones are ..< and >.. which create exclusive ranges.

```
// iterate the list forwards
for (i : 0 ..< list.size) {
        val element = list.get(i)
        ...
}
```

```
// or backwards
for (i : list.size >.. 0) {
        val element = list.get(i)
        ...
}
```

In addition there is the inclusive range, which is nice if you know both ends well. In the movies example the range is used to check whether a movie was made in a certain decade:

```
movies.filter[1980..1989.contains(year)]
```

Please keep in mind that there are other means to iterator lists, too. For example, you may want to use the forEach extension

```
list.forEach[ element, index |
        .. // if you need access to the current index
]
list.reverseView.forEach[
  .. // if you just need the element it in reverse order
]
```

Pair Operator

Sometimes you want to use a pair of two elements locally without introducing a new structure. In Xtend you can use the ->-operator which returns an instance of Pair<A,B>:

```
val nameAndAge = 'Homer' -> 42
```

If you want to surface a such a pair of values on the interface of a method or field, it's generally a better idea to use a data class with a well defined name, instead:

```
@Data class NameAndAge {
        String name
        int age
}
```

6.3.3. Assignments

Local variables (§6.5) can be assigned 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 is why the @Property annotation (§7.6) would rename the local field to _myProperty.

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

6.4. Blocks

The block expression allows to have imperative code sequences. It consists of a sequence of expressions. The value of the last expression in the block is the value of the complete block. The type of a block is also the type of the last expression. Empty blocks return null and have the type Object. Variable declarations (§6.5) are only allowed within blocks and cannot be used as a block's last expression.

A block expression is surrounded by curly braces. The expressions in a block can be terminated by an optional semicolon.

Here are two examples:

```
{
  doSideEffect("foo")
  result
}
```

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

6.5. Variable Declarations

Variable declarations are only allowed within blocks (§6.4). They are visible from any subsequent expressions in the block.

A variable declaration starting with the keyword val denotes a value, which is essentially a final, unsettable variable. The variable needs to be declared with the keyword var, which stands for 'variable' if it should be allowed to reassign its value.

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 (§6.6.2) it.

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

A local variable can be marked with the extension keyword to make its methods available as extensions (see extension provider (§5.8.4)).

6.5.1. Typing

The type of the variable itself can either be explicitly declared or it can be inferred from the initializer expression. Here is an example for an explicitly declared type:

```
var List<String> strings = 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 inferred from the initializater:

```
var strings = new ArrayList<String> // -> msg is of type ArrayList<String>
```

6.6. 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 parentheses are optional.

6.6.1. 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.)
```

6.6.2. Implicit Variables this and it

Like in Java the current instance of the class is bound to this. This allows for either qualified field access or method invocations like in:

```
this.myField
```

or it is possible to 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 is allowed to be shadowed. This is especially useful when used together with lambda expressions (§6.8).

As this is bound to the surrounding object in Java, it can be used in finer-grained constructs such as lambda expressions (§6.8). That is why it.myProperty has higher precedence than this.myProperty.

6.6.3. Static Access

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

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

Alternatively you could import the method using a static import (§5.2).

6.6.4. 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
```

Arguments that would be passed to the method are only evaluated if the method will be invoked at all.

6.7. Constructor Call

Constructor calls have the same syntax as in Java. The only difference is that empty parentheses are optional:

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

If type arguments are omitted, they will be inferred from the current context similar to Java's diamond operator on generic method and constructor calls.

6.8. 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 is best to think of a lambda expression as an anonymous class with a single method, 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 do not 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 notation for them, which is to leave the declaration including the vertical bar out. The name of the single variable (§6.6.2) will be it 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 parameter list. 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 parentheses for methods which get a lambda as their only argument, you can reduce the code above further down to:

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

A lambda expression also captures the current scope. Any final local variables and all parameters that are visible at construction time can be referred to from within the lambda body. That is exactly what we did with the variable textField above.

The variable this refers to the outer class. The lambda instance itself is available with the identifier self.

```
val lineReader = new LineReader(r);
val AbstractIterator<String> lineIterator = [|
        val result = lineReader.readLine
        if (result==null)
                self.endOfData
```

```
        return result
]
```

6.8.1. Typing

Lambdas 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 or abstract class which has declared only one abstract method. 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 inner types found in Functions or Procedures. It is a procedure if the return type is void, otherwise it is a function.

Xtend supports a shorthand syntax for function types. Instead of writing Function1<? super String,? extends String> which is what you will 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 lambda expression but not declared in the implemented method of the target type are thrown using the sneaky-throw technique (§5.6.3). Of course you can always catch and handle (§6.16) them.

6.9. If Expression

An if-expression is used to choose between 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
```

6.10. 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. Object.equals(Object) 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 myString 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 is not of type boolean it is compared to the value of 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 switch expression.

6.10.1. Type guards

Instead of or in addition to the case guard you can specify a *type guard*. The case only matches if the switch value conforms to this type. A case with both a type guard and a predicate only matches if both conditions match. If the switch value is a field, parameter or variable, it is automatically casted to the given type within the predicate and the case body.

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

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

6.11. 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 Iterable. The local variable is final, hence cannot be updated.

The type of a for loop is void. The type of the local variable can be inferred from the iterable or array that is processed.

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

for (s : myStrings)
  doSideEffect(s)
```

6.12. While Loop

A while loop

```
while (predicate) expression
```

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

```
while (true) {
  doSideEffect("foo")
}

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

6.13. Do-While Loop

A do-while loop

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

is used to execute a certain expression until the predicate is evaluated to false. The difference to the while loop (§6.12) is that the execution starts by executing the block once before evaluating the predicate for the first time. The type of a do-while loop is void.

```
do {
  doSideEffect("foo");
} while (true)


do doSideEffect("foo") while ((i=i+1)<max)
```

6.14. Return Expression

A method or lambda expression automatically returns the value of its body expression. If it is a block expression (§6.4) 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
]
```

6.15. Throwing Exceptions

Throwing 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")
  ...
}
```

6.16. Try, Catch, Finally

The try-catch-finally expression is used to handle exceptional situations. Checked exceptions are treated like runtime exceptions and only optionally validated. You can but do not have to catch them as they will be silently thrown (see the section on declared exceptions (§5.6.3)).

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

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

```
val name = try {
```

```
  readFromFile
} catch (IOException e) {
  "unknown"
}
```

6.17. 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 automatically 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 seldom 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 worse readability. The downside with the guillemets in Xtend is that you will have to have a consistent encoding. Always use UTF-8 and you are good.

If you use the Eclipse plug-in the guillemets will be inserted on content assist within a template. They are additionally bound to *CTRL+SHIFT+<* and *CTRL+SHIFT+* for « and » respectively. On a Mac they are also available with *alt+q* («) and *alt+Q* (»).

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

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

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

Template expressions can occur everywhere. Here is an example showing it in conjunction with the powerful switch expression (§6.10):

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

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

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

6.17.1. 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>
'''
```

6.17.2. Loops in Templates

Also a FOR expression 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 expression 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. It is executed if there are at least two iterations.

Here is 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>
'''
```

6.17.3. Typing

The template expression is of type CharSequence. It is automatically converted to String if that is the expected target type.

6.17.4. 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. The following three rules are applied 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 template 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 indentation 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 6.1.: 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 template string and thereby not considered to be relevant for the output but only for the 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.

7. Active Annotations

Active Annotations allow developers to participate in the translation process of Xtend source code to Java code via library. That's useful in cases where Java requires to write a lot of boilerplate manually. For instance, many of the good old design patterns fall into this category. With *Active Annotations* you no longer need to remember how the Visitor or the Observer pattern should be implemented. In Xtend you can implement the expansion of such patterns in a library and let the compiler do the heavy lifting for you.

An *Active Annotation* is just an annotation declared either in Java or Xtend, which is itself annotated with Active. @Active takes a type literal as a parameter pointing to the processor.

The IDE plugin comes with an example project, which you can easily materialize into your workspace. To do so use the new project wizard and in the category *Xtend Examples* choose the active annotation example. The examples contain three different annotations which we will use for further explanation.

For instance, @Extract is an annotation which extracts an interface for a class. The annotation declaration looks like this:

```
@Active(ExtractProcessor)
annotation Extract {}
```

7.1. Annotation Processor

A processor class must implement one or both of the lifecycle call-back interfaces provided by the compiler. There are some base classes for the most common usecases. These implement both interfaces:

- AbstractClassProcessor is a base class for class annotations
- AbstractMethodProcessor is a base class for method annotations
- AbstractFieldProcessor is a base class for field annotations

If you want to annotate other elements such as interfaces, parameters or constructors, you should have a look at the bases classes and adapt their implementation accordingly.

7.1.1. Phase 1: Register Globals

The first phase in the lifecycle of the compiler is about indexing the types as globally available symbols. By implementing a RegisterGlobalsParticipant you have the chance to create and register new Java types during this phase. It's important to do this in a first phase so later on the compiler can find and access these types.

For example the ExtractProcessor adds one interface per annotated class:

```
class ExtractProcessor extends AbstractClassProcessor {

  override doRegisterGlobals(ClassDeclaration annotatedClass, RegisterGlobalsContext context) {
    context.registerInterface(annotatedClass.interfaceName)
  }

  def getInterfaceName(ClassDeclaration annotatedClass) {
    annotatedClass.qualifiedName+"Interface"
  }

  ...
}
```

The RegisterGlobalsContext provides all the services that are available during this compilation step. It is passed into the method doRegisterGlobals() along with a read-only representation of the annotated Xtend elements. The AbstractClassProcessor in this example is invoked for all classes that are annotated with @Extract.

The compiler calls RegisterGlobalsParticipant once per compilation unit and provides access to all elements which are annotated with the *active annotation* this processor is registered for. Therefore the ExtractProcessor is invoked with a list of all classes that are defined in the same Xtend file for all the files that are being compiled.

7.1.2. Phase 2: Transformation

In the second phase developers can modify the compiled Java classes and Java code. Annotation processors that implement TransformationParticipant participate in this compile step. Similar to the RegisterGlobalsParticipant interface the compiler provides two arguments: The list of annotated elements and a TransformationContext. The context provides services that are specific for this second step.

A transformation participant can access and modify mutable Java elements. These are an in-memory representation of the generated Java code. They are usually very similar to the source elements, but can be modified and new methods, fields or constructors can be added. It is not possible to create new types during the transformation step.

The ExtractProcessor implements the method doTransform like this:

```
class ExtractProcessor extends AbstractClassProcessor {
```

```
  override doRegisterGlobals(ClassDeclaration annotatedClass, RegisterGlobalsContext context) {
    context.registerInterface(annotatedClass.interfaceName)
  }

  def getInterfaceName(ClassDeclaration annotatedClass) {
    annotatedClass.qualifiedName+"Interface"
  }

  override doTransform(MutableClassDeclaration annotatedClass, extension TransformationContext context) {
    val interfaceType = findInterface(annotatedClass.interfaceName)

    // add the interface to the list of implemented interfaces
    annotatedClass.implementedInterfaces = annotatedClass.implementedInterfaces + #[interfaceType.newTypeReference

    // add the public methods to the interface
    for (method : annotatedClass.declaredMethods) {
      if (method.visibility == Visibility::PUBLIC) {
        interfaceType.addMethod(method.simpleName) [
          docComment = method.docComment
          returnType = method.returnType
          for (p : method.parameters) {
            addParameter(p.simpleName, p.type)
          }
          exceptions = method.exceptions
        ]
      }
    }
  }
}
```

In the first line, findInterface retrieves the interface which was registered during the registration of global symbols in the first phase: The method is defined in TransformationContext which is used as an extension provider (§5.8.4).

Next up the newly created interface is added to the existing list of implemented interfaces.

```
annotatedClass.implementedInterfaces = annotatedClass.implementedInterfaces + #[interfaceType.newTypeReference]
```

The code calls setImplementedInterfaces(Iterable<TypeReference>) on the annotated class. The right hand side of the assignment is a concatenation of the existing implemented interfaces and a type reference pointing to the freshly created interface.

A TypeReference can be created using one of the various methods from TypeReferenceProvider which is a super type of TransformationContext. These utilities are available as extensions, too.

7.1.3. Phase 3: Code Generation

A third phase in the lifecycle of the compiler lets you participate in writing and updating the files. In the IDE this phase is only executed on save , while the previous two get executed after minor edits in the editor as well. In order to participate your processor needs to implement CodeGenerationParticipant. The extract interface example doesn't make use of this hook, but another example for internationalization which is also included, generates a *.properties file, like this:

```
class ExternalizedProcessor extends AbstractClassProcessor {

  ...

  override doGenerateCode(List<? extends ClassDeclaration> annotatedSourceElements, extension CodeGenerationConte
    for (clazz : annotatedSourceElements) {
      val filePath = clazz.compilationUnit.filePath
      val file = filePath.targetFolder.append(clazz.qualifiedName.replace('.', '/') + ".properties")
      file.contents = '''
        «FOR field : clazz.declaredFields»
          «field.simpleName» = «field.initializerAsString»
        «ENDFOR»
      '''
    }
  }

}
```

The CodeGenerationContext provides all the services that are available during this phase. Have a look at the Java doc for more details.

7.2. On Expressions and Statements

Most of the generated Java code is represented in the form of in-memory elements. Basically all the structural elements are represented as a dedicated Element. If you want to generate the body of a method or the initializer of a field, you have two options.

7.2.1. Generating Blackbox Java Code

The first option is to assign a compilation strategy and take care of the Java code by yourself. As an example the body of a setter method of an observable instance is implemented by the following code snippet:

```
observableType.addMethod('set' + fieldName.toFirstUpper) [
  addParameter(fieldName, fieldType)
```

```
  body = ['''
    «fieldType» _oldValue = this.«fieldName»;
    this.«fieldName» = «fieldName»;
    _propertyChangeSupport.firePropertyChange("«fieldName»", _oldValue, «fieldName»);
  ''']
]
```

A lambda expression is used to implement the body. This function is invoked later when the actual code is generated. It accepts a single parameter which is a CompilationStrategy .CompilationContext. The compilation context provides a convenient way write a TypeReference into a Java source file. It takes existing imports into account and adds new imports on the fly if necessary.

7.2.2. Assigning Expressions

A second alternative is to put expressions from the Xtend source into the context of a generated Java element. This allows to directly use the code that was written in the source file. An annotation @Lazy which turns fields into lazily initialized properties, may be used like this:

```
class MyClass {
  @Lazy String myField = expensiveComputation()
}
```

The processor for this *active annotation* could infer a synthetic initializer method and add a getter-method, which calls the initializer if the field is still null. Therefore, the initialization expression of the field has to become the method body of the synthesized initializer method. The following code performs this transformation:

```
override doTransform(MutableFieldDeclaration field, extension TransformationContext context) {

  // add synthetic init-method
  field.declaringType.addMethod('_init' + field.simpleName) [
    visibility = Visibility::PRIVATE
    returnType = field.type
    // reassign the initializer expression to be the init method's body
    // this automatically removes the expression as the field's initializer
    body = field.initializer
  ]

  // add a getter method which lazily initializes the field
  field.declaringType.addMethod('get' + field.simpleName.toFirstUpper) [
    returnType = field.type
```

```
    body = [''' 
      if («field.simpleName»==null)
        «field.simpleName» = _init«field.simpleName»();
      return «field.simpleName»;
    ''']
  ]
}
```

7.3. Custom Compiler Checks

The previous example requires each annotated field to have an initializer. Otherwise it would not be possible to use lazy initialization to assign its value. Also a simple check for a null reference could cause trouble with primitive values. A validation for that case would be sensible, too. In order to guide the user dedicated compilation errors should be raised if these constrains are violated.

The TransformationContext inherits methods for exactly that purpose from the ProblemSupport service.

Since the context is declared as an extension provider (§5.8.4), those methods can be used as extensions and it allows to implement the constraint check accordingly:

```
override doTransform(MutableFieldDeclaration field, extension TransformationContext context) {
  if (field.type.primitive)
    field.addError("Fields with primitives are not supported by @Lazy")

  if (field.initializer == null)
    field.addError("A lazy field must have an initializer.")
  ...
}
```

This ensures that the user is notified about invalid applications of the *active annotation* @Lazy.

7.4. Class Path Setup and Testing

An *active annotation* can not be used in the same project it is declared in, but has to reside on an upstream project. Alternatively it can be compiled and deployed in a jar. The annotation and the processor itself only rely on the interfaces defined in org.eclipse.xtend.lib.macro which is part of Xtend's small standard library.

When used in an IDE the compiled annotation processor is loaded and executed on the fly within the IDE process.

Therefore, careful testing and debugging of the processor is essential. It is best done in a unit test. Such a test needs the whole Xtend compiler on the class path, which you can

obtain by means of an OSGi bundle dependency or via Maven. The maven dependency is

```
<dependency>
  <groupId>org.eclipse.xtend</groupId>
  <artifactId>org.eclipse.xtend.standalone</artifactId>
  <version>2.4.2</version>
  <scope>test</scope>
</dependency>
```

and the equivalent OSGI bundle is org.eclipse.xtend.standalone.

7.4.1. Testing

The XtendCompilerTester is a convenient helper class for testing the processing and the compilation. It allows to test active annotation processors by either comparing the generated Java source using a String comparison or by inspecting the produced Java elements. In addition you can even compile the generated Java source to a Java class and create and test instances of it reflectively.

The example project contains a couple of test cases:

```
class LazyTest {
  extension XtendCompilerTester compilerTester = XtendCompilerTester.newXtendCompilerTester(Lazy)

  @Test def void testLazy() {
    '''
      import lazy.Lazy

      class Person {
        @Lazy String name = 'foo'
      }
    '''.assertCompilesTo(
    '''
      import lazy.Lazy;
      @SuppressWarnings("all")      public class Person {        @Lazy        private String name;        private String _ini
  }
}
```

This is a basic string comparison. It is a good way to start the development of a new annotation processor. Later on assertions against the produced elements and syntax tree (AST) may be a better choice since these are not so fragile against changes in the formatting. The @Extract-example uses this technique:

```
@Test def void testExtractAnnotation() {
  '''
    @extract.Extract
    class MyClass {
      override String doStuff(String myParam) throws IllegalArgumentException {
        return myParam
      }
    }
  '''.compile [
    // declare the transformation context as a local extensions
    val extension ctx = transformationContext

    // look up the interface and the class
    val interf = findInterface('MyClassInterface')
    val clazz = findClass('MyClass')


    // do assertions
    assertEquals(interf, clazz.implementedInterfaces.head.type)

    interf.declaredMethods.head => [
      assertEquals('doStuff', simpleName)
      assertEquals(string, returnType)
      assertEquals(IllegalArgumentException.newTypeReference, exceptions.head)
    ]
  ]
}
```

7.4.2. Wrap Up

Active Annotations are a powerful and unique concept that allows to solve a large class of problems that previously had to be solved in cumbersome ways. IDE wizards, many code generators or manually writing boilerplate code are no longer state of the art. Active annotations basically *is* a means of code generation, but its simple integration with existing projects and the fast development turn-arounds diminish the typical downsides of code generation.

Important note: The Active Annotation-API as of version 2.4 is provisional, and might be changed and improved in later releases.

7.5. Existing Active Annotations

Xtend comes with ready-to-use active annotations for common code patterns. They reside in the org.eclipse.xtend.lib plug-in/jar which must be on the class path of the project containing the Xtend files.

7.6. @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 accessor 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;
}
```

7.7. @Data

The annotation @Data, will turn an annotated class into a value object class. A class annotated with @Data is processed according to the following rules:

- all fields are final,
- getter methods will be generated (if they do not yet exist),
- a constructor with parameters for all non-initialized fields will be generated (if it does not exist),
- equals(Object) / hashCode() methods will be generated (if they do not exist),
- a toString() method will be generated (if it does not exist).

Example:

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

```
  def static void main(String[] args) {
    val p = new Person(args.get(0), args.get(1))
    println(p.getFirstName() + ' ' + p.lastName)
  }
}
```

List of External Links

http://docs.oracle.com/javase/specs/jls/se5.0/html/conversions.html#5.5
http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html
http://en.wikipedia.org/wiki/Observer_pattern
http://docs.oracle.com/javase/specs/jls/se5.0/html/classes.html#8.4.4
http://projecteuler.net/problem=1
http://en.wikipedia.org/wiki/Visitor_pattern
http://projecteuler.net/
http://projectlombok.org/features/SneakyThrows.html
http://docs.oracle.com/javase/specs/jls/se5.0/html/conversions.html