Xtend Reference Documentation

October 31, 2011

Contents

Part I.

Xtend

1. Xtend

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

- *Advanced Type Inference* - You rarely need to write down type signatures
- *Full support for Java Generics* - Including all conformance and conversion rules
- *Closures* - concise syntax for anonymous function literals
- *Operator overloading* - make your libraries even more expressive
- *Powerful switch expressions* - type based switching with implicit casts
- *No statements* - Everything is an expression
- *Template expressions* - with intelligent white space handling
- *Extension methods* - Enhance closed types with new functionality optionally injected via JSR-330
- *property access syntax* - shorthands for getter and setter access
- *multiple dispatch* aka polymorphic method invocation
- *translates to Java* not bytecode - understand what's going on and use your code for platforms such as Android or GWT

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

1.1. Getting Started

The best way to get started is to materialize the *Xtend Tutorial* example project in your workspace. You'll find it in the new project wizard dialog.

The project contains a couple of sample Xtend files, which show the different language concepts in action. You should also have a look into the *xtend-gen* folder which contains the generated Java-version of them.

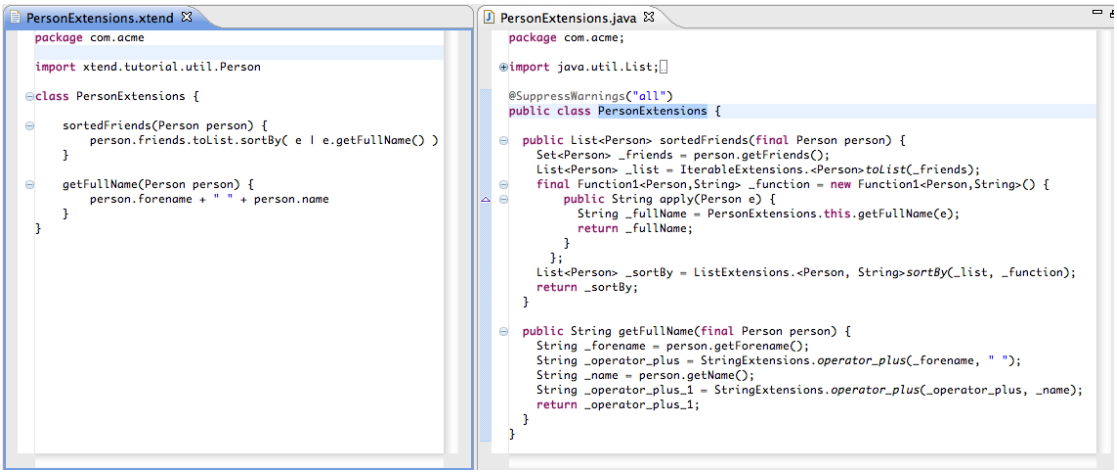

Figure 1.1.: Xtend and Java Vis-a-Vis

1.2. Classes and Functions

On 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 after that comes the class definition. That class in fact is directly translated to a Java class in the corresponding Java package.

Here is an example:

```
package com.acme

import java.util.List

class MyClass {
    String first(List<String> elements) {
        return elements.get(0)
    }
}
```

1.2.1. Package Declaration

Package declarations are like in Java, with the small difference, that an identifier can be escaped with a ^ in case it conflicts with a keyword. Also you don't terminate a package declaration with a semicolon.

```
package org.eclipse.xtext
```

Would be interesting to document how the following example translates to java

```
package my.^public.^package
```

1.2.2. Imports

The ordinary imports of type names are equivalent to the imports known from Java. Again one can escape any names conflicting with keywords using the ^. In contrast to Java, the import statement is never terminated with a semicolon. Xtend also features static imports but allows only a wildcard at the end. So far you cannot import single members using a static import. For non-static imports the use of wildcards is 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.*
```

Xtend supports extension methods, which allows to add functions to existing classes without modifying them. Static extension functions are just one possibility - simply put the keyword **extension** after the **static** keyword and the static functions will be made available as member functions on their first parameter's type.

That is the following import declaration

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

allows to use its methods for example like this :

```
new Foo().singletonList()
```

Although this is supported it is generally much nicer to use injected extensions (§1.2.4), because they don't bind you to the actual implementation.

1.2.3. Class Declaration

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

To be implemented: The **abstract** as well as the **final** modifiers are directly translated to Java and have the exact same meaning.

Inheritance

Also inheritance is directly reused from Java. Single inheritance of Java classes as well as implementing multiple Java interfaces is supported.

Generics

Full Java Generics with the exact same syntax and semantics are supported. That is you can declare type parameters just as in Java and provide type arguments to types you refer to (i.e. extend or implement).

Examples

The most simple class :

```
class MyClass {
}
```

A more advanced class declaration in Xtend :

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

1.2.4. Fields

Fields in Xtend are commonly used together with an annotation for a dependency injection container. Example:

```
@Inject MyService myService
```

This will translate to the following Java field:

```
@Inject
private MyService myService;
```

Note that the default visibility is **private**. You can also declare it explicitly as being **public**, **protected**, or **private**. Alternatively, you can write accessor methods. Xtend does not yet support any other modifier from Java on fields.

Extension Methods

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

Imagine you want to add a method 'fullName' to a closed type 'Entity'. With extension methods, you could declare the following class

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

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

```
@Inject extension PersonExtensions
```

The method is being put on the member scope of Person. This is why you can skip the field's name. You can now write the following

```
myPerson.getFullName()
```

which is translated to the Java code

```
_personExtensions.getFullName(myPerson)
```

where _personExtensions is the default name of the field. Of course the property shorthand is still available

```
myPerson.fullName
```

The nice thing with using dependency injection in combination with the extension modifier as opposed to static extensions (§1.2.2) is, that in case there is a bug in the extension or it is implemented inefficiently or you just need a different strategy, you can simply exchange the component with another implementation. You do this without modifying the library nor the client code. You'll only have to change the binding in your guice module. Also this gives you a general hook for any AOP-like thing you would want to do, or allows you to write against an SPI, where the concrete implementation can be provided by a third party.

1.2.5. Functions

Xtend functions are declared within a class and are translated to a corresponding Java method with the exact same signature. (The only exceptions are dispatch methods, which are explained here (§1.2.6)).

Let's start with a simple example

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

Visibility

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

1.2.6. Overriding Functions

Functions can override a function/method from the super class or implemented interfaces using the keyword **override**. If a function is annotated with the keyword **final**, it cannot be overridden. IF a function overrides a function (or method) from a super type, the override keyword is mandatory and replaces the keyword def.

Example:

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

Declared Exceptions

To be implemented:

Xtend doesn't force you to catch checked exceptions. If a called method throws a checked exception and it is not catched or explicitly declared to be rethrown it will be wrapped in a runtime exception and rethrown.

A declared checked exception will not be wrapped automatically.

```
/*
 * throws an IOException
 */
def void throwIOException() throws IOException {
    throw new IOException()
}

/*
 * throws a WrappedException
 */
def void throwWrappedException() {
    throw new IOException()
}
```

Inferred Return Types

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

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

could be declared like this:

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

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

Generics

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

Dispatch Functions

Generally function binding works just like method binding in Java. That is function calls are bound based on the static types of arguments. Sometimes this is not what you want. Especially in the context of extension methods one would like to have polymorphic behavior.

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

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

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

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

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

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

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

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

Note that the instanceof cascade is ordered by how specific a type is. More specific types come first.

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

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

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

generates the following Java code :

```
public String foo(Number x, Number y) {
    if ((x instanceof Number)
        && (y instanceof Integer)) {
        return _foo((Number)x,(Integer)y);
```

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

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

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

Which compiles to the following Java code:

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

Overloading Functions from Super Types

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.toString(",")+"):"+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(Object o) {
    if (o instanceof Field) {
        return _label((Field)o);
    } else if (o instanceof Method){
        return _foo((Method)o);
    } else if (o instanceof Entity){
        return _foo((Entity)o);
    } else if (o instanceof Object){
        return _foo((Object)o);
    } else {
        throw new IllegalArgumentException(
            "Couldn't handle argument o:"+o);
    }
}
```

1.2.7. Create Functions

Create functions in Xtend allow to do graph transformation in one pass where it usually needs two passes. That means you don't need to separate a translation from one graph to another in the typical two phases: tree construction and interlinking the tree nodes. You basically just need to write the whole transformation using create functions and the built-in identity preservation will take care of the rest.

Consider you want to create a copy of the following list of persons into a :

```
Fred Flintstone {
    marriedTo Willma Flintstone
    friendWith Barny Rubble
}
Willma Flintstone {
    marriedTo Fred Flintstone
}
Barny Rubble {
    friendWith Fred Flintstone
}
```

A function like the following could do the trick :

```
def List<Person> copyPersons(List<Person> persons) {
    persons.map( p | p.copy )
}

def copy(Person p) {
    val result = new Person()
    result.name = p.name
    // The following is wrong and results in a stack overflow
    result.friendWith = p.friendWith.map( p | p.copy )
    result.marriedWith = p.marriedWith.map( p | p.copy )
}
```

The problem with that code is that we don't track the origins of the created copies. This is the main problem with model transformations. The classic solution is to run the copying in two passes. First we create all instances and then we establish the links. Although it works it results in cluttered and non coherent code. Xtend's create functions handle this problem by introducing identity perservation by tracking the origin of each created instance. Therefore, a create function takes two expressions. One to instantiate the actual object and a second one to initialize it.

```
def create result: new Person() copy(Person p) {
    result.name = p.name
    // now it works
    result.friendWith = p.friendWith.map( p | p.copy )
    result.marriedWith = p.marriedWith.map( p | p.copy )
}
```

If you do not specify the name of the result variable it is assumed to be the implicit receiver variable (§**??**) `it`, which can be skipped in feature calls inside the body. Furthermore, you can define the return type of a create function:

```
def Person create new PersonImpl() copy(Person p) {
    /* it.*/name = p.name
    friendWith = p.friendWith.map( p | p.copy )
    marriedWith = p.marriedWith.map( p | p.copy )
}
```

How it works

In addition to the keyword **create** one specifies two expressions. The first expression is the factory to create an instance while the second will initialize it further. Prior to invoking the factory expression, a cache lookup is performed to find a previously created instance for the very same arguments. If there is no such instance, the factory expression is evaluated and the result is stored in the cache. Subsequently the main expression (also called the initializer expression) is evaluated. This happens only if there was no previously created instance available in the cache. If that expression in turn calls the create function transitively using the same set of arguments the previously instantiated

and cached object is returned. Note that the object is probably currently initialized. That is, its internal state may not yet be available.

The lifecycle of the cache is attached to the instance of the declaring Xtend class. That is you can control how long the cache lives by means of Guice.

1.2.8. Annotations

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

Example:

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

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

1.3. Expressions

The most important part of a function is of course its implementation, which in Xtend is either a single block expression (§1.3.10) or a rich string expression (§1.3.17).

1.3.1. Literals

A literal denotes a fixed unchangeable value. Literals for string, integers, booleans, null and Java types are supported.

String Literals

A string literal is a valid expression and returns an instance of java.lang.String of the given value.

- 'Hello World !'
- "Hello World !"
-
```
"Hello
        World !"
```

Integer Literals

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

- 42
- 234254

Boolean Literals

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

- true
- false

Null Literal

The null pointer literal is, like in Java, null. It is a member of any reference type.

- null

Type Literals

Type literals are specified using the keyword *typeof* :

- **typeof**(java.lang.String) which yields java.lang.String.class

1.3.2. Type Casts

Type cast behave like casts in Java, but have 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.

Syntax

```
XCastedExpression:
    Expression 'as' JvmTypeReference;
```

Examples

- my.foo **as** MyType
- $(1 + 3 * 5 * (-23))$**as** BigInteger

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

e1 += e2	e1._operator_add(e2)
e1 || e2	e1._operator_or(e2)
e1 && e2	e1._operator_and(e2)
e1 == e2	e1._operator_equals(e2)
e1 != e2	e1._operator_notEquals(e2)
e1 < e2	e1._operator_lessThan(e2)
e1 > e2	e1._operator_greaterThan(e2)
e1 <= e2	e1._operator_lessEqualsThan(e2)
e1 >= e2	e1._operator_greaterEqualsThan(e2)
e1 -> e2	e1._operator_mappedTo(e2)
e1 .. e2	e1._operator_upTo(e2)
e1 + e2	e1._operator_plus(e2)
e1 - e2	e1._operator_minus(e2)
e1 * e2	e1._operator_multiply(e2)
e1 / e2	e1._operator_divide(e2)
e1 % e2	e1._operator_modulo(e2)
e1 ** e2	e1._operator_power(e2)
! e1	e1._operator_not()
- e1	e1._operator_minus()

The table above also defines the operator precedence in ascending order. The blank lines separate precedence levels. The assignment operator += is right-to-left associative in the same way as the plain assignment operator = is. That is a = b = c is executed as a = (b = c), all other operators are left-to-right associative. Parenthesis can be used to adjust the default precedence and associativity.

Short-Circuit Boolean Operators

If the operators || and && are used in a context where the left hand operand is of type boolean, the operation is evaluated in short circuit mode, which means that the right

hand operand might not be evaluated at all in the following cases:

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

Examples

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

Assignments

Local variables (§1.3.9) can be reassigned using the = operator. Also properties can be set using that operator: Given the expression

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

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

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

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

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

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

1.3.4. Feature Calls

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

Syntax

The following snippet is a simplification of the real Xtext rules, which cover more than the concrete syntax.

```
FeatureCall :
    ID |
    Expression ('.' ID ('(' Expression (',' Expression)* ')')?)*
```

Property Access

Feature calls are directly translated to their Java equivalent with the exception, that for calls to properties an equivalent rule as described in section 1.3.3 applies. That is, for the following expression

```
myObj.myProperty
```

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

Implicit 'this' variable

If the current scope contains a variable named **this**, the compiler will make all its members available to the scope. That is if

```
this.myProperty
```

is a valid expression

```
myProperty
```

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

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 more readable.

Instead of writing

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

one can write

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

1.3.5. Constructor Call

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

Examples

- `new String()`
- `new java.util.ArrayList<java.math.BigDecimal>()`

Syntax

```
XConstructorCall:
  'new' QualifiedName
        ('<' JvmTypeArgument (',' JvmTypeArgument)* '>')?
        ('('(XExpression (',' XExpression)*)?')')?;
```

1.3.6. Closures

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

Syntax

```
XClosure:
    '[' ( JvmFormalParameter (',' JvmFormalParameter)* )?
        '|' XExpression ']';
```

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

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

instead of

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

But in all other cases the square brackets are mandatory:

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

Typing

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

For instance, given the following Java method signature:

```
public T <T>getFirst(List<T> list, Function0<T,Boolean> predicate)
```

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

```
arrayList( "Foo", "Bar" ).findFirst( e | e == "Bar" )
```

instead of

```
arrayList( "Foo", "Bar" ).findFirst( String e | e == "Bar" )
```

Function Mapping

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

- *Function0<ReturnType>* for zero parameters,
- *Function1<Param1Type, ReturnType>* for one parameters,
- *Function2<Param1Type, Param2Type, ReturnType>* for two parameters,
- ...
- *Function6<Param1Type, Param2Type, Param3Type, Param4Type, Param5Type, Param6Type, ReturnType>* for six parameters,

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

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

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

Examples

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

1.3.7. If Expression

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

Syntax

```
XIfExpression:
    'if' '(' XExpression ')'
        XExpression
    ('else' XExpression)?;
```

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

```
if (foo) x
```

is the a short hand for

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

Typing

The type of an if expression is calculated by the return types T1 and T2 of the two expression e1 and e2. It uses the rules defined in **??**.

Examples

- **if** (isFoo)**this** **else** that
- **if** (isFoo){ **this** } **else if** (thatFoo){ that } **else** { other }
- **if** (isFoo)**this**

1.3.8. Switch Expression

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

For a switch expression

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

the main expression `e` is evaluated first and then each case sequentially. If the switch expression contains a variable declaration using the syntax known from subsection 1.3.11, the value is bound to the given name. Expressions of type java.lang.Boolean or boolean are not allowed in a switch expression.

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

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

Example:

```
switch myString {
    case myString.length>5 : 'a long string.'
    case 'foo' : 'It's a foo.'
    default : 'It's a short non-foo string.'
}
```

Type guards

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

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

Only if the switch value passes a type guard, i.e. an instanceof operation returns true, the case's guard expression is executed using the same semantics explained in 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.

Typing

The return type of a switch expression is computed using the rules defined in **??**. The set of types from which the common super type is computed corresponds to the types of each case's result expression. In case a switch expression's type is computed using the expected type from the context, it is sufficient to return the expected type if all case branches types conform to the expected type.

Examples

- ```
  switch foo {
      Entity : foo.superType.name
      Datatype : foo.name
      default : throw new IllegalStateException
  }
  ```
- ```
  switch x : foo.bar.complicated('hello',42) {
      case "hello42" : ...
      case x.length<2 : ...
      default : ....
  }
  ```

Syntax

```
XSwitchExpression:
    'switch' (ID ':')? XExpression '{'
        XCasePart+
        ('default' ':' XExpression))?
    '}';

XCasePart:
    JvmTypeReference? ('case' XExpression)? ':' XExpression );
}
```

1.3.9. Variable Declarations

Variable declarations are only allowed within blocks (§1.3.10). They are visible in any subsequent expressions in the block. Although overriding or shadowing variables from outer scopes is allowed, it is usually only used to overload the variable name 'this', in order to subsequently access an object's features in an unqualified manner.

A variable declaration starting with the keyword **val** denotes a so called value, which is essentially a final (i.e. unsettable) variable. In rare 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
    }
}
```

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

Syntax

```
XVariableDeclaration:
    ('val' | 'var') JvmTypeReference? ID '=' XExpression;
```

Typing

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

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

In such cases, the right hand expression's type must conform (§**??**) to the type on the left hand side.

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

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

1.3.10. Blocks

The block expression allows to have imperative code sequences. It consists of a sequence of expressions, and returns the value of the last expression. The return type of a block is also the type of the last expression. Empty blocks return null. Variable declarations (§1.3.9) 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.

Examples

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

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

Syntax

```
XBlockExpression:
    '{'
        (XExpressionInsideBlock ';'?)*
    '}';
```

1.3.11. For Loop

The for loop **for** (T1 variable : iterableOfT1)expression is used to execute a certain expression for each element of an array of 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)
  ```

Syntax

```
XForExpression:
    'for' '(' JvmFormalParameter ':' XExpression ')'
        XExpression
    ;
```

1.3.12. 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**.

Syntax

```
XWhileExpression:
    'while' '(' predicate=XExpression ')'
        body=XExpression;
```

Examples

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

1.3.13. 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 (§1.3.12) 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**.

Syntax

```
XDoWhileExpression:
    'do'
        body=XExpression
    'while' '(' predicate=XExpression ')';
```

Examples

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

1.3.14. Return Expression

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

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

1.3.15. Throwing Exceptions

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

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

1.3.16. Try, Catch, Finally

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

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

1.3.17. Rich Strings

Rich Strings allow for readable string concatenation, which is the main thing you do when writing a code generator. Let's have a look at an example of how a typical function with template expressions looks like:

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

  «placeImports»
```

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

If you are familiar with Xpand, you'll notice that it is exactly the same syntax. The difference is, that the template syntax is actually an expression, which means it can occur everywhere where an expression is expected. For instance in conjunction the powerful switch expression (§1.3.8):

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

1.3.18. Conditions in Rich Strings

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

1.3.19. Loops in Rich Strings

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

1.3.20. Typing

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

1.3.21. White Space Handling

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

1. An evaluated rich string as part of another string will be prefixed with the current indentation of the caller before it is inserted into the result.

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

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

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

```
node NodeName{}
```

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

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

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

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

List of External Links

http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.5

Todo list

Would be interesting to document how the following example translates to java . . 5