Xbase Language Reference

Sven Efftinge, Sebastian Zarnekow, et al.

May 11, 2011

Contents

1 Preface

This document describes the expression language library Xbase. Xbase is a partial programming language implemented in Xtext and is meant to be embedded and extended within other programming languages and domain-specific languages (DSL) written in Xtext. Xtext is a highly extendable language development framework covering all aspects of language infrastructure such as parsers, linkers, compilers, interpreters and even full-blown IDE support based on Eclipse.

Developing DSLs has become incredibly easy with Xtext. Structural languages which introduce new coarse-grained concepts, such as services, entities, value objects or state-machines can be developed in minutes. However, software systems do not consist of structures solely. At some point a system needs to have some behavior, which is usually specified using so called *expressions*. Expressions are the heart of every programming language and are not easy to get right. On the other hand, expressions are well understood and many programming languages share a common set and understanding of expressions. That is why most people do not add support for expressions in their DSL, but try to solve this differently. The most often used workaround is to define only the structural information in the DSL and add behavior by modifying or extending the generated code. It is not only unpleasant to write, read and maintain information which closely belongs together in two different places, abstraction levels and languages. Also, modifying the generated source code comes with a lot of additional problems. But as of today this is the preferred solution since adding support for expressions (and a corresponding execution environment) for your language is hard - even with Xtext.

Xbase serves as a language library providing a common expression language bound to the Java platform (i.e. Java Virtual Machine). It consists of an Xtext grammar, as well as reusable and adaptable implementations for the different aspects of a language infrastructure such as an AST structure, a compiler, an interpreter, a linker, and a static analyzer. In addition it comes with implementations to integrate the expression language within an Xtext-based Eclipse IDE. Default implementations for aspects like content assistance, syntax coloring, hovering, folding and navigation can be easily integrated and reused within any Xtext based language.

Conceptually and syntactically, Xbase is very close to Java statements and expressions, but with a few differences:

- Runs on the JVM
- No checked exceptions
- Object-oriented
- Everything is an expression, there are no statements

- Closures
- Type inference
- Properties
- Simple operator overloading
- Powerful switch expressions

2 Lexical Syntax

Xbase comes with a small set of lexer rules, which can be overridden and hence changed by users. However the default implementation is carefully chosen and it is recommended to stick with the lexical syntax described in the following.

2.1 Identifiers

Identifiers are used to name all constructs, such as types, methods and variables. Xbase uses the default Identifier-Syntax from Xtext - compared to Java, they are slightly simplified to match the common cases while having less ambiguities. They start with a letter *a-z*, *A-Z* or an underscore followed by more of these characters or a digit *0-9*.

2.1.1 Escaped Identifiers

Identifiers may not have the same spelling as any reserved keyword. However, identifiers starting with a ^ are so called escaped identifiers. Escaped identifiers are used in cases when there is a conflict with a reserved keyword. Imagine you have introduced a keyword service in your language but want to call a Java property *service* at some point. In such cases you use an escaped identifier ^service to reference the Java property.

2.1.2 Syntax

terminal ID:
    '^'? ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
;

2.1.3 Examples

- Foo
- Foo42
- FOO
- _42
- _foo
- ^extends

2.2 String Literals

String literals can either use single quotes (') or double quotes (") as their terminals. When using double quotes all literals allowed by Java string literals are supported. In addition new line characters are allowed, that is in Xbase all string literals can span multiple lines. When using single quotes the only difference is that single quotes within the literal have to be escaped and double quotes do not.

See § 3.10.5 String Literals

In contrast to Java, equal string literals within the same class do not neccessarily refer to the same instance at runtime.

2.2.1 Syntax

//TODO

2.2.2 Examples

- 'Foo Bar Baz'
- "Foo Bar Baz"
- " the quick brown fox
  jumps over the lazy dog."
- 'Escapes : \' '
- "Escapes : \" "

2.3 Integer Literals

Integer literals consist of one or more digits. Only decimal literals are supported and they always result in a value of type java.lang.Integer (it might result in native type int when translated to Java, see Types (§3)). The compiler makes sure that only numbers between 0 and *Integer.MAX* (0x7fffffff) are used.

There is no negative integer literal, instead the expression −23 is parsed as the prefix operator − applied to an integer literal.

2.3.1 Syntax

```
terminal INT returns ecore::EInt:
    ('0'..'9')+
;
```

2.4 Comments

Xbase comes with two different kinds of comments: Single-line comments and multi-line comments. The syntax is the same as the one known from Java (see § 3.7 Comments).

2.4.1 Syntax

```
terminal ML_COMMENT :
    '/*' -> '*/'
;
terminal SL_COMMENT :
    '//' !('\n'|'\r')* ('\r'? '\n')?
;
```

2.5 White Space

The white space characters ' ', '\t', '\n', and '\r are allowed to occur anywhere between the other syntactic elements.

2.6 Reserved Keywords

The following list of words are reserved keywords, thus reducing the set of possible identifiers:

1. **extends**
2. **super**
3. **instanceof**
4. **as**
5. **new**
6. **null**
7. **false**
8. **true**
9. **val**
10. **var**
11. **if**
12. **else**
13. **switch**
14. **case**
15. **default**
16. **do**

17. **while**
18. **for**
19. **typeof**
20. **throw**
21. **try**
22. **catch**
23. **finally**

However, in case some of the keywords have to be used as identifiers, the escape character for identifiers (§2.1.1) comes in handy.

3 Types

Basically all kinds of JVM types are available and referable.

3.1 Arrays

Arrays cannot be declared explicitly, but they can be passed around and if needed are transparently converted to a List of the compound type.

In other words, the return type of a Java method that returns an array of ints (*int[]*) can be directly assigned to a variable of type *java.util.List<java.lang.Integer>* (in short *List<Integer>*). Due to type inference you can also defer the conversion. The conversion is bi-directional so any method, that takes an array as argument can be invoked with a *List* instead.

3.2 Simple Type References

A simple type reference only consists of a *qualified name*. A qualified name is a name made up of identifiers which are separated by a dot (like in Java).

3.2.1 Syntax

```
QualifiedName:
  ID ('.' ID)*
;
```

There is no parser rule for a simple type reference, as it is expressed as a parameterized type references without parameters.

3.2.2 Examples

- java.lang.String
- String

3.3 Function Types

Xbase introduces *closures*, and therefore an additional function type signature. On the JVM-Level a closure (or more generally any function object) is just an instance of one of the types in org.eclipse.xtext.xbase.lib.Function*, depending on the number of arguments. However, as closures are a very important language feature, a special sugared syntax for

function types has been introduced. So instead of writing Function1<String,Boolean> one can write (String)=>Boolean.

Primitives cannot be used in function types.

For more information on closures see section 4.6.

3.3.1 Syntax

```
XFunctionTypeRef:
  ('('JvmTypeReference (',' JvmTypeReference)*')')?
    '=>' JvmTypeReference;
```

3.3.2 Examples

- =>Boolean // predicate without parameters
- (String)=>Boolean // One argument predicate
- (Mutable)=>Void // A method doing side effects only – returns null
- (List<String>, Integer)=>String

3.4 Parameterized Type References

The general syntax for type references allows to take any number of type arguments. The semantics as well as the syntax is almost the same as in Java, so please refer to the third edition of the Java Language Specification.

The only difference is that in Xbase a type reference can also be a function type. In the following the full syntax of type references is shown, including function types and type arguments.

3.4.1 Syntax

```
JvmTypeReference:
  JvmParameterizedTypeReference |
  XFunctionTypeRef;

XFunctionTypeRef:
  ('(' JvmTypeReference (',' JvmTypeReference)* ')')?
    '=>' JvmTypeReference;

JvmParameterizedTypeReference:
  type=QualifiedName ('<' JvmTypeArgument (',' JvmTypeArgument)* '>')?;

JvmTypeArgument:
  JvmReferenceTypeArgument |
  JvmWildcardTypeArgument;

JvmReferenceTypeArgument :
```

```
  JvmTypeReference;

JvmWildcardTypeArgument:
  '?' (JvmUpperBound | JvmLowerBound)?;

JvmLowerBound :
 'super' JvmTypeReference;

JvmUpperBound :
 'extends' JvmTypeReference;
```

3.4.2 Examples

- String
- java.lang.String
- List<?>
- List<? extends Comparable<? extends FooBar>
- List<? super MyLowerBound>
- List<? extends =>Boolean>

3.5 Primitives

Xbase supports all Java primitives. The conformance rules (e.g. boxing unboxing) are also exactly like defined in the Java Language Specification.

3.6 Conformance and Conversion

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

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

3.6.1 Common Super Type

Because of type inference Xbase sometimes needs to compute the most common super type of a given set of types.

For a set *[T1,T2,...Tn]* of types the common super type is computed by using the linear type inheritance sequence of *T1* and is iterated until one type conforms to each *T2,..,Tn*. The linear type inheritance sequence of *T1* is computed by ordering all types which are part if the type hierarchy of *T1* by their specificity. A type *T1* is considered more specific than *T2* if *T1* is a subtype of *T2*. Any types with equal specificity will be

sorted by the maximal distance to the originating subtype. *CharSequence* has distance 2 to *StringBuilder* because the supertype *AbstractStringBuilder* implements the interface, too. Even if *StringBuilder* implements *CharSequence* directly, the interface gets distance 2 in the ordering because it is not the most general class in the type hierarchy that implements the interface. If the distances for two classes are the same in the hierarchy, their qualified name is used as the compare-key to ensure deterministic results.

4 Expressions

Expressions are the main language constructs which are used to express behavior and computation of values. The concept of statements is not supported, but instead powerful expressions are used to handle situations in which the imperative nature of statements would be helpful, too. An expression always results in a value (might be the value 'null' though). In addition every resolved expressions is of a static type.

4.1 Literals

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

4.1.1 String Literals

A string literal as syntactically defined in section 2.2 is a valid expression and returns an instance of java.lang.String of the given value.

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

4.1.2 Integer Literals

An integer literal as defined in section 2.3 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

4.1.3 Boolean Literals

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

- **true**
- **false**

4.1.4 Null Literal

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

- **null**

4.1.5 Type Literals

Type literals are specified using the keyword *typeof* :

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

4.2 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.

4.2.1 Syntax

XCastedExpression:
    Expression 'as' JvmTypeReference;

4.2.2 Examples

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

4.3 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.

4.3.1 Short-Circuit Boolean Operators

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

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

4.3.2 Examples

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

4.3.3 Assignments

Local variables (§4.9) 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 Xbase 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")

4.4 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.

4.4.1 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)* ')')?)*
```

4.4.2 Property Access

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

myObj.myProperty

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

4.4.3 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.

4.4.4 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. Xbase supports the safe navigation operator ?. to make such code more readable.

Instead of writing

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

one can write

myRef?.doStuff()

4.5 Constructor Call

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

4.5.1 Examples

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

4.5.2 Syntax

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

4.6 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.

4.6.1 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]
```

4.6.2 Typing

Closures are expressions which produce function objects. The type is a function type (§3.3), 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,Boo`

the type of the parameter can be inferred. Which allows users to write: `arrayList( "Foo", "Bar").findFirst( e | e =`

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

4.6.3 Function Mapping

An Xbase closure is a Java object of one of the *Function* interfaces shipped with the runtime library of Xbase. 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
             })
```

4.6.4 Examples

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

4.7 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.

4.7.1 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**

4.7.2 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 subsection 3.6.1.

4.7.3 Examples

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

4.8 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 section 4.11, the value is bound to the given name. Expressions of type java.lang.Boolean or boolean are not allowed in a switch expression.

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

If none of the guards matches the default expression is evaluated 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.'
}
```

4.8.1 Type guards

In addition to the case guards one can add a so called *Type Guard* which is syntactically just a type reference (§3.2) 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.

4.8.2 Typing

The return type of a switch expression is computed using the rules defined in subsection 3.6.1. 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.

4.8.3 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 : ....
  }
  ```

4.8.4 Syntax

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

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

4.9 Variable Declarations

Variable declarations are only allowed within blocks (§4.10). 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.

4.9.1 Syntax

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

4.9.2 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 (§3.6) 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>*

4.10 Blocks

The block expression allows to have imperative code sequences. It consists of a sequence of expressions, and returns the value of the last expression. The return type of a block is also the type of the last expression. Empty blocks return null. Variable declarations (§4.9) 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.

4.10.1 Examples

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

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

4.10.2 Syntax

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

4.11 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)
  ```

4.11.1 Syntax

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

4.12 While Loop

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

4.12.1 Syntax

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

4.12.2 Examples

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

4.13 Do-While Loop

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

4.13.1 Syntax

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

4.13.2 Examples

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

4.14 Return Expression

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

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

4.15 Throwing Exceptions

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

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

4.16 Try, Catch, Finally

The try-catch-finally expression is used to handle exceptional situations. You are not forced to declare checked exceptions, if you 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
}
```

List of External Links

http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.7
http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.5
http://java.sun.com/docs/books/jls/third_edition/html/conversions.html
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.10.5

Todo list