1 Preface

This document specifies the expression language 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 statemachines can be developed in minutes. However, software systems do not consist of structures solely. At some point a system needs to show 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. 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 ships in form 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 like Java statements+expressions, with the following differences:

- No checked exceptions
- Pure object-oriented, i.e. no built-in types and no arrays
- 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. They start with a "Java letter", which is a character for which the Java method *Character.isJavaIdentifierstart(char)* returns *true*. For the other characters also digits (0...9) are allowed. For those the method *Character.isJavaIdentifierPart(char)* must be true. See § 3.8 Identifiers for the original definition in the Java Language Specification.

2.1.1 Syntax

//TODO

2.1.2 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.

2.1.3 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 (**??**)). 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. **new**
5. **null**
6. **false**
7. **true**
8. val
9. var
10. **if**
11. **else**
12. **switch**
13. **case**
14. **default**
15. **do**
16. **while**

17. **for**
18. **class**
19. throw
20. try
21. catch
22. finally

However, in case some of the keywords have to be used as identifiers, the escape character for identifiers (2.1) comes in handy.

3 Types

Xbase binds to the Java Virtual Machine. This means that expressions written in Xbase refer to Java types and Java type members. Xbase itself uses only types defined in the Java language, such as classes, interfaces, annotations and enums. It also supports Java generics and shares the known syntax. In addition to Java, Xbase comes with the notion of function types.

Xbase does not bind to any of the built-in types such as *int* or *boolean*, instead any references to those built-in types will automatically use the corresponding wrapper type, e.g. *java.lang.Boolean* instead of *boolean*. Arrays are supported also in Xbase but are translated to *java.util.List*s: An array *int[]* binds to *java.util.List<java.lang.Integer>* (in short *List<Integer>*). This means when referring to `myList.isEmpty()` within an Xbase expression the static return type is *java.lang.Boolean*. At runtime however the compiler may be smarter and use the native types. Especially the types *int* and *boolean* are most often used with built-in operators and the wrapper type is only occasionally needed (for instance when putting `int`s into collections).

3.1 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).

3.1.1 Syntax

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

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

3.1.2 Examples

- `java.lang.String`
- `String`

3.2 Function Types

Xbase introduces *closures*, which require an additional kind of type. 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∗, dependening 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.

3.2.1 Syntax

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

3.2.2 Examples

- =>Boolean *//predicate without parameters*
- (String)=>Boolean *//One argument predicate*
- (Mutable)=>Void *//A method doing side effects on an instance of Mutable and returns null*
- (List<String>)=>String

3.3 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 fo type references is shown, including function types and type arguments.

3.3.1 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;

3.3.2 Examples

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

3.4 The type java.lang.Void

The **null** reference is the only valid value of the type Void, which gets some special treatment in Xbase. Every Java method which is declared *void* (i.e. without a return value) is translated to a method with return type *java.lang.Void*. At runtime such method invocations will result in *null*. The speciality is that while it is allowed to pass **null** everywhere (TODO discuss use of nullable annotation) instead of any other value, this does not mean that *java.lang.Void* is a subtype of any other type. The instanceOf operator as well as the type matchers in the section 4.6 do not match **null**.

3.5 Conformance Rules

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.

A type *T1* conforms to a type *T2* if

- T1==T2
- T1==java.lang.Void
- T1 is a subtype of T2

T1<T1P,..T1Pn> conforms to *T2<T2P, ... T2Pn>* if *T1* conforms to *T2* and each upper bound of a *T1Pn* conforms to the corresponding upper bound of *T2Pn*.

3.5.1 Common Super Type

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 their qualified name just to ensure deterministic results.

4 Expressions

Expressions are the main language constructs which are used to express behavior and computation of values. Xbase does not support the concept of a statement, but instead comes with powerful expressions to handle situations in which the imperative nature of statements are a better fit. An expression always results in a value (might be the value 'null' though). In addition expressions can be statically typed. By default it is assumed that languages making use of Xbase provide enough static context information for static type analysis, which is the basis of a lot of IDE features coming with Xbase. However, the static typing is not mandatory and might be completely skipped if not wished. The openness of the compiler even allows to change the generation of concrete feature invocations to reflective calls, such that the language can be fully dynamically typed.

4.1 Literals

A literal denotes a fixed unchangeable value. Xbase comes with the following literals

4.1.1 String Literals

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

4.1.2 Syntax

```
XStringLiteral:
    STRING;
```

4.1.3 Integer Literals

An integer literal as defined in section 2.3 creates an instance of Integer.

4.1.4 Syntax

```
XIntegerLiteral:
    INT;
```

4.1.5 Boolean Literals

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

4.1.6 Syntax

XBooleanLiteral:
'false' | 'true';

4.1.7 Null Literal

The null pointer literal is, like in Java, **null**. It is the only value of the type *java.lang.Void* which has a special meaning in Xbase (see section 3.4).

4.1.8 Syntax

XNullLiteral:
{XNullLiteral} 'null';

4.1.9 Type Literals

Type literals are written like in Java. They consist of a reference to a raw type suffixed with a dot and the keyword **class**.

4.1.10 Syntax

XTypeLiteral:
QualifiedName '.' 'class';

4.2 Infix Operators

Xbase supports a couple of predefined infix operators. In contrast to Java, the operators are not fixed to operations on certain types. Instead Xbase comes with an operator to method mapping, which 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.

<table>
<tr><td>e1.someProp = e2</td><td>e1.someProp = e2</td></tr>
<tr><td></td><td>e1.someProp(e2)</td></tr>
<tr><td></td><td>e1.setSomeProp(e2)</td></tr>
<tr><td>e1 += e2</td><td>e1.add(e2)</td></tr>
<tr><td>e1.someFeature += e2</td><td>e1.addSomeFeature(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1 || e2</td><td>e1.or(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1 && e2</td><td>e1.and(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1 instanceof RawTypeRef</td><td>e1 instanceof RawTypeRef /*direct translation to Java */</td></tr>
<tr><td>e1 == e2</td><td>e1.equals(e2)</td></tr>
<tr><td>e1 != e2</td><td>e1.notEquals(e2)</td></tr>
<tr><td>e1 < e2</td><td>e1.lessThan(e2)</td></tr>
<tr><td>e1 > e2</td><td>e1.greaterThan(e2)</td></tr>
<tr><td>e1 <= e2</td><td>e1.lessEqualsThan(e2)</td></tr>
<tr><td>e1 >= e2</td><td>e1.greaterEqualsThan(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1 -> e2</td><td>e1.mappedTo(e2)</td></tr>
<tr><td>e1 .. e2</td><td>e1.upTo(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1 + e2</td><td>e1.plus(e2)</td></tr>
<tr><td>e1 - e2</td><td>e1.minus(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1 * e2</td><td>e1.multiply(e2)</td></tr>
<tr><td>e1 / e2</td><td>e1.divide(e2)</td></tr>
<tr><td>e1 % e2</td><td>e1.modulo(e2)</td></tr>
<tr><td>e1 ** e2</td><td>e1.power(e2)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>! e1</td><td>e1.not()</td></tr>
<tr><td>- e1</td><td>e1.minus()</td></tr>
<tr><td></td><td></td></tr>
<tr><td>e1[e2]</td><td>e1.apply(<Exp2)</td></tr>
</table>

The table above also defines the operator precedence in ascending order. The blank lines separate precedence levels. The two assignment operators = and += are right-to-left associative, 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.

4.2.1 Property Assignment

The translation rule for the simple assignment operator = is a bit more complicated. 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, first a method called myProperty (OneArg) and then setMyProperty(OneArg) 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")

4.2.2 Add Assignment

The translation rule for the add assignment operator += is as follows: Given the expression

myObj.myProperty += "foo"

The compiler first looks up whether, the left hand side operand is of some type providing a method add(StringOrSuperType). In that case the expression translates to the following Java expression :

myObj.myProperty.add("foo");

If there is no such method, the compiler looks for a method called addMyProperty(OneArg) on the type of the target expression of the feature call. That is it looks whether the type of myObj provides a method called addMyProperty(StringOrSuperType).

The return value and compile-time type will be whatever the invoked Java method returns.

4.2.3 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**.

4.2.4 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 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 (TODO define how scopes are declared around expressions).

4.3.1 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)* ')')?)*
```

4.3.2 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 4.2.1 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.

4.3.3 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
```

if a valid expression

```
myProperty
```

is valid as well and is equivalent.

4.3.4 Examples

- `foo`
- `my.foo`
- `my.foo(x)`
- `oh.my.foo(bar)`

4.4 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.

4.4.1 Syntax

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

JvmFormalParameter:
    JvmTypeReference? ID;
```

4.4.2 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. 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 Google Guava (formerly known as Google Collect) the type Function1 extends com.google.common.base.Function<F, T> and the type *Function0* extends com.google.common.base.Supplier<T>. There is also an auto-coercion for any Function1<T,Boolean> to com.google.common.base.Predicate<T>.

TODO: Discuss, whether we want to go a step further and do auto conversion to any type declaring only one method. That would allow to pass closures also to methods expecting other such types like e.g. Iterable*. This would avoid a fixed dependency to Google Guava, too.*

4.4.3 Typing

Closures are expressions which produce function objects. The type is a function type (3.2), 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:

```
getFirst(arrayList("Foo","Bar"), e|e=="Bar")
```

instead of

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

4.4.4 Examples

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

4.5 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.

4.5.1 Syntax

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

An expression **if** (p)e1 **else** e2 results to 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 is

if (foo) x *// is the same as 'if (foo) x else null'*

4.5.2 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 3.5.1.

4.5.3 Examples

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

4.6 Switch Expression

4.6.1 Syntax

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

XCasePart:
    JvmTypeReference? ('case' XExpression)? ':'
        (XBlockExpression | XExpression ';');
```

The switch statement is a bit different from the one in Java. 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 a case expression en evaluates to something such that e == en the result of the switch expression is ern. If none of the case expressions e1...en is equal to the result of the main expression e the result of the optional default part er is returned. If not default is defined the expression returns **null**.

4.6.2 Leaving out the main expression

It is possible to leave out the main expression. Then the case expressions e1...en have to be of type Boolean. They are evaluated in the specified order and as soon as one predicate ex evaluates to true the corresponding then expression ex is the result of the switch expression. So it is mainly an alternative syntax to the section 4.5.

4.6.3 Type guards

In addition to the case predicate one can add a so called *Type Guard* which is syntactically just type reference (3.1) in front of the 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
}
```

The expression x, will be of type String in any expression within the first case and of type List<?> in the second. Note that the case expression has to be a predicate (i.e. it has to be of type Boolean). At runtime a type guard is translated to an **instanceof** of test and subsequent casts. The expression x is reevaluated everytime it is used. If the expression returns a different value on a subsequent evaluation, it might cause a ClassCastException at runtime.

4.6.4 Typing

The return type of a switch expression is computed using the rules defined in subsection 3.5.1. The set of types from which the common super type is computed corresponds to the types of each case's result expression.

4.7 Variable Declarations

Variable declarations are only allowed within a block (4.8). They are visible in any subsequent expressions in the block.

4.7.1 Syntax

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

Xbase resembles the keywords val and var known from Scala (The Scala Language Specification 2.8). 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 keyowrd var, which stands for 'variable'.

```
{
    var i = 0
    while (i>MAX) {
        print("Hi there!")
    }
}
{
    val myFoo = my.complex(expression)
    myFoo.call(myFoo)
}
```

4.7.2 Typing

The return type of a variable declaration expression is always java.lang.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 (3.5) 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>
```

4.8 Blocks

The block expression allows to simulate 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.

Variable declarations (4.7) are only allowed within blocks and cannot be used as a block's last expression.

4.8.1 Syntax

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

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

4.8.2 Examples

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

4.9 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 java.lang.Void and the return value is **null**.

4.9.1 Syntax

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

4.9.2 Examples

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

4.10 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 (4.9) 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 java.lang.Void and the return value is **null**.

4.10.1 Syntax

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

4.10.2 Examples

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

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

4.11 For Loop

The for loop **for** (T1 variable : iterableOfT1)expression is used to execute a certain expression for each element of an java.lang.Iterable. The local variable is final, hence canot be updated.

4.11.1 Syntax

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

4.11.2 Typing

The return type of a for loop is java.lang.Void and the return value is **null**. The type of the local variable can be left out. In that case it is inferred from the type of the java.lang.Iterable returned by the iterable expression.

4.11.3 Examples

- **for** (String s : myStrings) {
      doSideEffect(s);
  }

- **for** (s : myStrings)
      doSideEffect(s)

4.12 Constructor Call

Construction of objects is done by invoking Java constructors. Xbase uses the **new** keyword and the syntax is like the one known from Java.

4.12.1 Syntax

```
XConstructorCall:
  'new' (type=JvmTypeReference '('(XExpression (',' XExpression)*)?')')?;
```

4.12.2 Example

new Foo()

4.13 Throwing Exceptions

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

4.13.1 Syntax

```
XThrow:
  'throw' XExpression;
```

4.13.2 Typing

The type of a throw expression is always java.lang.Void. The type of the expression after the throw keyword needs to conform to java.lang.Throwable.

4.13.3 Example

```
throw new RuntimeException()
```

4.14 Try, Catch, Finally

The try-catch-finally expression is used to handle exceptional situations gracefully. Xbase never forces you to catch exceptions, because there is no such concept like checked exceptions in Java. The syntax again is like the one known from Java.

4.14.1 Syntax

```
XTryCatchFinally:
  'try' XBlockExpression
  CatchClause*
  FinallyClause?;

CatchClause:
    'catch' XDeclaredParameter
        XBlockExpression

FinallyClause:
    'finally' XBlockExpression
```

4.14.2 Example

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