1 Introduction

The purpose of this document is to define the language extensions for the Java language that enable the programmer to specify immutability constraints about references.

It is assumed that the reader already knows the Java language. The specification in this document is for the language extensions only, and not for the syntax and semantics already present in the Java language. The Java language with the extensions described in this document is hereinafter referred to as BEJava.

2 Keywords

`mutable` and `template` are new keywords introduced into the language, and can no longer be used as identifiers. `const` was a keyword in the Java language, but was not used in any of its syntax; it is a keyword that is used in BEJava.

3 Types

3.1 The types

A new type hierarchy is introduced in BEJava, which includes in it the type hierarchy from the Java language. For any Java reference type `T`, a new type `const T` is created. References of type `const T` are just like those of type `T`, but cannot be used to modify the object to which they refer.

Formally, the types in BEJava are the following:

1. The null type `null`.
2. The Java primitive types.
3. Instance references. If `O` is any class or interface, then `O` is a type representing a references to an instance `O`.
4. Arrays. For any non-null type `T`, `T[]` is a type, representing an array of elements of type `T`.
5. Constant types. For any non-null non-constant type `T`, `const T` is a type.

For convenience in the usage later, we define several terms dealing with arrays.

For a type `T` and integer n, an n-dimensional array of `T` is defined as follows:

- if $n = 0$, the n-dimensional array of `T` is `T`.
- if `S` is the $(n-1)$-dimensional array of T, then `S[]` is the n-dimensional array of `T`.

For a type `T`, define $depth$(`T`) as follows:

- if `T` is null, primitive or instance reference, $depth(\texttt{T}) = 0$.
- if $depth(\texttt{T}) = n$, then $depth(\texttt{T[]}) = n + 1$
- if $depth(\texttt{T}) = n$, then $depth(\texttt{const T}) = n$

For a type `T`, define $base$(`T`) as follows:

- if `T` is null, primitive or instance reference, $base(\texttt{T}) = \texttt{T}$.
- if $base(\texttt{T}) = \texttt{S}$, then $base(\texttt{T[]}) = \texttt{S}$
- if $base(\texttt{T}) = \texttt{S}$, for a constant type S, then $base(\texttt{const T}) = \texttt{S}$
- if $base(\texttt{T}) = \texttt{S}$, for a non-constant type S, then $base(\texttt{const T}) = \texttt{const S}$

3.2 The syntax for types

Except for the null type, every type can be named in BEJava, although not in a unique way. The following syntax is used for naming the type (for clarity, non-terminals are enclosed in quotes).

```
Type ::= [ 'const' ] BasicType ( '[' ']' )*

BasicType ::=
   PrimitiveType
   Name
   '(' Type ')'
```

The type named by the a sequence of tokens, S, that parses according to the above grammar is determined as follows. Let S' be the subsequence of S matched by the `BasicType` non-terminal. First the type named by S' is determined. If `BasicType` matches according to the `PrimitiveType` production, S' names the primitive type matched there; if `BasicType` matched according to the `Name` production, the name matched by that production must name an accessible class or interface, and S' names a reference to an instance of that class or interface. If `BasicType` matches by the `'(' Type ')'` production, then S' names the same type as the subsequence matched by `Type` in the production.

Once it is determined what S' names, the type named by the S is determined as follows. Suppose `T` is the type named by S'. If n is the number of `'[' ']'` pairs following `BasicType`, let `U` be the n-dimensional array of `T`. Finally let `V` be `const U` or `U`, depending on whether the initial `const` in the grammar above is present in the parsing. The type named by S is `V`.

3.3 The equality for types and the subtype relationship

The equality relation is defined on the types as follows:

1. For primitive types, the null type and references to instances of classes and interfaces, two types are equal iff they are the same Java type.

2. `const T` and `const S` are equal iff $depth$(`T`) = $depth$(`S`) and $base$(`const T`) = $base$(`const S`).

3. `T[]` and `S[]` are equal iff `T`, `S` are.

4. For a non-constant type `T`, `T` and `const S` are equal iff `T` and `S` are equal, and `T` is either primitive or is a reference to an instance of an immutable class or interface (see section 4.4 for description of immutable classes/interfaces).

As an example, note that item 2 implies that `const int[][]` and `const (const int[])[]` are equivalent, for example. In other words, constant array of array of `int` is the same as constant array of constant `int` arrays.

Equal types are considered to be the same type throughout the rest of this specification. They should be interchangeable in any BEJava program.

A subtyping relationship (`T` subtype of `S`, written as `T` < `S`) is also defined on types. It is the transitive reflexive closure of the following:

1. `byte` < `char`, `byte` < `short`, `char` < `int`, `short` < `int`, `int` < `long`, `long` < `float`, `float` < `double`.

2. `null` < `T` for any type `T` which is not a primitive type.

3. If `T`, `S` are classes such that `T` extends `S` or interfaces such that `T` extends `S`, or `S` is an interface and `T` is a class implementing `S`, then `T` < `S`.

4. For any non-null types `T`, `S`, if `T` < `S` then `T[]` < `S[]`.

5. For any non-constant non-null type `T`, `T` < `const T`.

6. For any non-constant non-null types `T`, `S`, `T` < `S` $\Rightarrow$ `const T` < `const S`.

7. For any non-null type `T`, if `S` is `T[]`, then `S` $<$ `java.io.Serializable`, `S` $<$ `Cloneable`, and `S` $<$ `Object`.

8. For any type non-constant non-null type `T`, `(const T)[]` $<$ `const T[]`.

4 Other new syntax

In addition to the new type hierarchy, BEJava has the following features: constant methods and constructors, mutable fields and immutable classes and interfaces. These are described in the subsections below.

4.1 Constant methods

A method declaration has the following grammar:

```
MethodDeclaration ::=
  MethodModifiers (Type | 'void') Identifier Arguments ['const'] [ThrowsClause]
    (MethodBody | ';')
```

The only difference from the Java grammar for method declaration is the optional keyword `const` immediately after the `Arguments` of the method. If the declaration contains this keyword `const`, it is said to declare a constant method. Only instance (non-static) methods can be declared as constant. There are no other restrictions on which methods can be declared as constant.

The semantics for constant methods are as follows. A constant method is an instance method that can be invoked through a constant reference. Non-constant methods cannot be so invoked. Additionally, `this` is of a constant type inside the body of a constant method, so that a constant method cannot modify the state of the object on which it is invoked. For a formal description of these rules, see the type checking rules described in section 5.

4.2 Constant constructors

A constructor declaration has the following grammar:

```
ConstructorDeclaration ::=
  ConstructorModifiers Identifier Arguments ['const'] [ThrowsClause]
  ConstructorBody
```

The only difference from the Java grammar for constructor declaration is the optional keyword `const` immediately after the `Arguments` of the constructor. If the declaration contains this keyword `const`, it is said to declare a constant constructor. If a constructor for a class that is not an inner class is declared to be constant, a compile-time error occurs. There are no other restrictions on which constructors can be declared as constant.

The semantics of constant constructors are as follows. A constant constructor is a constructor for an inner class that can be invoked with a constant reference for the enclosing object. Non-constant constructors for inner classes cannot be so invoked. Inside the body of a constant constructor, no modifications to the enclosing instance are allowed. For a formal description of these rules, see the type checking rules described in section 5.

4.3 Mutable fields

A field declaration has the following grammar:

```
FieldDeclaration ::=
  FieldModifiers Type Identifier [Initializer]
     (',' Identifier [Initializer])* ';'
```

```
FieldModifier ::=
  ('mutable' | 'private' | 'public' | 'protected' | 'final' |
   'static' | 'transient')*
```

The only difference from the grammar for field declaration in Java is the possibility of `mutable` appearing as a modifier. If a declaration contains `mutable` as a modifier, the fields declared in it are said to be mutable. Only instance (non-static) fields can be declared as mutable. There are no other restriction on which fields can be declared as mutable.

The semantics of mutable fields are as follows. A mutable field is not part of the state of the object to which it belongs. Thus, the state of a mutable field of a given object can be changed through constant reference to that object, or by constant methods invoked on that object, or by constant constructors invoked with that object as the enclosing instance. For a formal description of these rules, see the type checking rules described in section 5.

4.4 Immutable classes and interfaces

Class and interface declarations have the following grammar:

```
ClassDeclaration ::=
  TypeModifiers class Identifier [ExtendsClause] [ImplementsClause]
    ClassBody

InterfaceDeclaration ::=
  TypeModifiers interface Identifier
    [InterfaceExtendsClause] ClassBody

TypeModifiers ::=
  ('const' | 'private' | 'protected' | 'public' | 'abstract' |
   'final' | 'strictfp' | 'static' )*
```

The only difference from the grammar for class/interface declaration in Java is the possibility of `const` appearing as a modifier. If a declaration contains `const` as a modifier, the class/interface declared in it is said to be immutable.

The semantics of immutable classes and interfaces are as follows. An instance of such a class or interface, once instantiated, cannot be modified. Therefore, in an immutable class/interface, every instance method and every constructor is implicitly declared as constant, and any instance field which is not explicitly declared as mutable is implicitly declared as final and if its type `T` is not a constant type, it is implicitly changed to be `const T`.

It is a compile-time error for a non-immutable class or interface to extend or implement an immutable one. It is a compile-time error for an immutable class or interface to inherit an instance field which is neither mutable nor final with a constant type, or to inherit, override or implement an instance method which which is not constant.

5 Type checking rules

BEJava has the same runtime behaviour as Java. However, at compile time, checks are done to ensure that modification of objects through constant references, or similar violations of the language, do not occur. These rules are described in this section. Section 5.1 introduces some definitions. Section 5.2 then presents the type checking rules.

5.1 Definitions

5.1.1 Definitions for types

- Primitive type: any Java primitive type, e.g., `boolean` or `double`

- Reference type: any non-primitive type.
- Numeric type: any primitive type other than `boolean`.
- Integral type: any numeric type other that `float` and `double`.
- Null type: `null`.
- Array type: Any type `S` such that `S = T[]` for some type `T`.
- Constant type: Any type `S` such that `S = const T` for some type `T`.

5.1.2 Definitions relation to method invocations

Compatibility: Given a method or constructor M and a list of arguments $A_1, A_2, \ldots A_n$, we say that the arguments are compatible with M if M is declared to take n paremeters, and for each i from 1 to n, the type of A_i is a subtype of the declared type of the ith parameter of M.

Specificity: Given two methods of the same name or two constructors of the same class, M_1, M_2, we say that M_1 is more specific than M_2 if the following three conditions hold:

1. M_1 and M_2 take the same number of parameters, say with types $P_1, P_2 \ldots P_n$ for M_1, and $Q_1, Q_2 \ldots Q_n$ for M_2, and for each i from 1 to n, P_i is a subtype of Q_i.
2. The class/interface in which M_1 is declared is a subclass/subinterface of the one where M_2 is declared, or M_1 and M_2 are declared in the same class/interface.
3. Either M_1 is not constant or M_2 is constant (or both).

Note that the definitions above are the same as those in Java, except for presence of the third clause in the definition of specificity.

5.2 Type checking rules

5.2.1 Programs

A program type checks if every top-level class/interface declaration in the program type checks.

5.2.2 Class/Interface declarations

A class or interface declaration type checks if all of the following hold:

1. (a) The class/interface is immutable and each of the methods declared in any of its superclasses or superinterfaces is either private, static or constant, and each of the fields declared in any of its superclasses is either private, static, mutable or both final and of a constant type, or
   (b) the class or interface is not immutable, and neither is its direct superclass or any of its direct superinterfaces.
2. No two fields of the same name are declared within the body of the class/interface.
3. No two methods of the same name and signature are declared within the body of the class/interface. Signature includes number and declared types of parameters, as well as whether the method is constant.
4. Every field, method, member type, instance initializer and static initilizer declared within the class/interface type checks.

5.2.3 Variable declarations

For a field or local variable declaration of type `T`:

- If it does not have an initializer, it type checks.
- If it has an initializer of the form `=` E for an expression E, it type checks iff the assignment of the expression E to a left hand side with type `T` would type check.
- If it has an initializer of the form `= { ... }` , it will type check iff `T` is an array type and the declaration would type check had the initializer been `= new T { ... }`.

5.2.4 Method declarations

A a method, constructor, instance initializer or static initializer type checks if every expression, local variable declaration, and local type declaration in the body of the method, constructor, instance initializer or static initializer type checks.

5.2.5 Expressions

Each expression has a type and a boolean property called assignability associated with it. An expression is type checked recursively, with all subexpressions type checked first. If the subexpressions type check, then their types and assignability are used to type check the given expression and deduce its type and assignability. Otherwise, the given expression does not type check. The rules for type checking and expression given types and assignabilities of subexpressions are given below.

Assignments

- An assigment A`=`B type checks if the expression A is assignable, and one of the following holds:
  - the type of A is supertype of that of B,or
  - A is of type `byte`, `short` or `char`, and B is of type `byte`, `short`, `char` or `int` and is a compile-time constant whose value is within the value range of the type of A.
- An assignment A`+=`B type checks if the expression A is assignable, and one of the following holds:
  - A and B are both of numeric types, or
  - A is of type `String`
- An assignment of one of the forms A`-=`B, A`*=`B, A`/=`B, A`%=`B type checks whenever A is assignable and A and B are of numeric types.
- An assignment of one of the forms A`<<=`B, A`>>=`B, A`>>>=`B type checks whenever A is assignable and A and B are of integral types.
- An assignment of one of the forms A`||=`B, A`&&=`B type checks whenever A is assignable and both A and B are of type `boolean`.
- An assignment of one of the forms A`&=`B, A`|=`B, A`^=`B type checks whenever A is assignable and either both A and B have integral types or they are both of type `boolean`.

The type of any assignment expression that type checks is the same as the type of the left hand side, and the expression is not assignable.

Other compound expressions

- A?B:C: In order for this expression to type check, A must be of type `boolean`. Also, if `T1` and `T2` denote the types of B and C, then one of the following must hold:
  1. `T1` $<$ `T2`, `T1` $<$ `const T2`, `T2` $<$ `T1`, or `T2` $<$ `const T1`. In that case expression is of the least common supertype of `T1`, `T2`.
  2. `T1` and `T2` are, in some order, `char` and `short`; in that case expression is of type `int`.
  3. One of B, C is of type `T`, where `T` is `byte`, `short`, or `char`, and the other is a constant expression of type `int` whose value is representable in type `T`. In that case expression is of type `T`.
- A||B, A&&B: the expression type checks if A, B are of type `boolean`; expression is of type `boolean`.
- A|B, A ^ B, A&B: the expression type checks if A, B are of type `boolean`, in which case the expression is of type `boolean`; or, if A, B are of integral type, in which case their least common supertype is the type of the expression.
- A==B, A!=B: Let `T1` and `T2` be the types of A and B respectively. The expression type checks if `T1` $<$ `T2`, `T1` $<$ `const T2`, `T2` $<$ `T1`, or `T2` $<$ `const T1`. The expression is of type `boolean`.
- A `instanceof T`: always type checks, is of type `boolean`.
- A<B, A>B, A<=B, A>=B: type checks if A, B are of numeric type; it is of type `boolean`.
- A<<B, A>>B, A>>>B: type checks if A, B are of integral type; the expression is of the least common supertype of the type of A and of `int`.
- A+B: type checks if A, B are of numeric type, in which case the expression is of the least supertype of `int` and the types of A, B; or if one of A, B is of type `String`, in which case the expression is of type `String`.
- A-B, A*B, A/B, A%B: type checks if A, B are of numeric type, in which case the expression is of the least supertype of `int` and the types of A, B.
- +A, -A: type checks if A is of numeric type; the expression is the least supertype of `int` and the type of A.
- ++A, --A, A++, A--: type checks if A is assignable and of numeric type; the expression is of same type.
- ~ A: type checks if A is of numeric type; the expression is the least supertype of `int` and the type of A.
- !A: type checks if A is of type `boolean`; the expression is of type `boolean`.
- (T)A: let `S` be the type of A; then the expression type checks if either `S` $<$ `T` or `T` $<$ `S`; the expression is of type `T`.

Every expression in this section is not assignable.

Primary Expressions

- A literal is of type `boolean`, of a numeric type, of type `String`, or of type `null`, depending on the value of the literal. Literals are not assignable.
- `this` does not type check in a static context; in a non-static context `this` has type `C` if `C` is a class and `this` appears inside a non-constant method, a non-constant constructor, or an initializer of C; `this` has type `const C` inside a constant method or a constant constructor of `C`. `this` is not assignable.

- `NAME.this` type checks if it occurs in a non-static context in a method, constructor or initializer of a class `I`, and `NAME` names a class `C` for which `I` is an inner class. The type of the expression is `C` unless it appears inside a constant method or a constant constructor of `I`, in which case the type is `const C`. This expression is not assignable.

- `(`A`)` always type checks and is of the same type as A, and is not assignable.

- `T.class` always type checks and is of type `Class` and is not assignable.

- `new NAME(ARGS)` type checks if `NAME` is the name of an accessible non-inner class `C`, and there exists the most specific accessible constructor that is compatible with `ARGS`; or if `C` is a direct inner class of a class `O`, and `O.this.new NAME(ARGS)` type checks. The expression has type `C` and is not assignable.

- A`.new NAME(ARGS)` where A is an expression, type checks if one of the following cases holds:

  1. A has type that is subtype of `O` for some class `O`, and `NAME` is the name of an accessible direct inner class `C` of the class `O`, and there exists the most specific accessible constructor that can be called on `ARGS`. In this case the expression is of type C.
  2. A has type `const O` for some class `O`, and `NAME` is the name of an accessible direct inner class `C` of the class `O`, and there exists the most specific accessible constant constructor that can be called on `ARGS`. In this case the type of the expression is `const C`.

  In either case, the expression is not assignable.

- Array instance creation expression: Let `T` be the array type whose instance is being created, and let `S` be such that `T = S[]`. The expression type checks whenever all index expressions involved are of integral types and the array initializer $\{E_1, \ldots, E_n\}$, if any, consists of expression E_i such that $E = E_i$ would type check for an expression E of type `S`. The type of the array instance creation is `T`. An array instance creation expression is not assignable.

- A`[`E`]` type checks if E is of integral type, A is of type `T[]` or `const T[]` for some type `T`; the type of the expression is respectively `T` or `const T`. The expression is assignable in the first case, and not assignable in the second.

- A`.IDENTIFIER`, where A is an expression: let `T` be a non-constant reference type such that A's type is `T` or `const T` (if no such type exists, the expression does not type check). Then the expression type checks if one of the following holds:

  1. `T` is a non-array type, and `IDENTIFIER` is the name of an accessible field of the class or interface named by `T`; the expression is assignable unless A is of type `const T` or the field is a non-blank final variable. If `S` is the declared type of the field, then the expression is of type `S` unless A is of type `const T` and the field is not static nor mutable and not of a constant type, in which case it is `const S`.
  2. `T` is an array type, and `IDENTIFIER` is `length`, in which case the expression is of type `int`; it is not assignable.
  3. `T` is an array type, and `IDENTIFIER` is a field of `Object`, `Cloneable` or `java.io.Serializable`; if `S` is the declared type of the field, then the type of the expression is `S` unless `S` is not a constant type, A is of type `const T` and the field is neither static nor mutable, in which case it `const S` and is not assignable.

- `NAME.IDENTIFIER`: If `NAME` resolves to a field, variable or parameter of type `T`, this type checks iff the expression E`.IDENTIFIER` with E of type `T` typechecks. Otherwise this expression type checks whenever `NAME` is the name of an accessible class or interface, and `IDENTIFIER` is the name of an accessible static field; the declared type of the field is the type of the expression and the expression is assignable unless the field is a non-blank final variable.

- `IDENTIFIER` type checks if one of the following holds:

1. There is a visible local variable or parameter declaration with a name `IDENTIFIER`; the type of the expression then is the declared type of that local variable or parameter, and the expression is assignable unless the variable in a non-blank final local variable or a final parameter (either a method parameter or a catch clause parameter).
2. 1 does not hold, the expression occurs in a static context and `IDENTIFIER` is the name of an accessible static field of a class `C` within whose declaration the expression occurs; the type and assignability of the expression is the same as that of `C.IDENTIFIER` for the innermost such class `C`.
3. 1 does not hold, the expression occurs in a non-static context and there exist a class `C` within whose declaration the expression occurs and which contains an accessible field with name `IDENTIFIER`, and for the innermost such class `C`, `C.this.IDENTIFIER` type checks; the type of `IDENTIFIER` is the type of `C.this.IDENTIFIER`. The assignability of `IDENTIFIER` is also the assignability of `C.this.IDENTIFIER`, unless `IDENTIFIER` is within the body of a constant constructor of the class `C`, in which case `IDENTIFIER` is assignable.

- `super.IDENTIFIER`. Does not type check in static context. In non-static context, let `C` be the innermost enclosing class or interface. If `C` is an interface or the class `Object`, this does not type check. Otherwise let `P` be the direct superclass of `C`. If `IDENTIFIER` names an accessible field of `P`, this type checks. Let the declared type of that field be `T`. If the field is mutable or static or `T` is a constant type, the type of `IDENTIFIER` is `T`. Otherwise, if `IDENTIFIER` occurs inside a constant method or constant constructor, its type is `const T`, otherwise it is `T`.

- A`.IDENTIFIER(ARGS)`, where A is an expression: let `T` be a non-constant reference type such that A's type is `T` or `const T` (if no such type exists, the expression does not type check). The expression type check if one of the following holds:

  1. `T` is a non-array reference and there exists the most specific accessible method of name `IDENTIFIER` of the class or interface named by `T` which is compatible with the arguments `ARGS` and is constant or static if type of A is `const T`; the type of the expression then is the declared return type of that method.
  2. `T` is an array reference and there is the most specific accessible method of name `IDENTIFIER` in `Object`, `Cloneable` or `java.io.Serializable` which is compatible with the arguments `ARGS` and is constant or static if type of A is is not in category 5; the type of the expression then is the declared return type of that method.

- `NAME.IDENTIFIER(ARGS)`: If `NAME` resolves to a field, variable or parameter of type `T`, this type checks iff the expression E`.IDENTIFIER(ARGS)` with E of type T typechecks. Otherwise the expression type checks whenever `NAME` is the name of an accessible class `C`, and there exists the most specific static method of name `IDENTIFIER` in `C` compatible with the arguments `ARGS`; the type of the expression then is the declared return type of that method. This expression is never assignable.

- `IDENTIFIER(ARGS)` type checks if one of the following holds:

  1. It occurs in a static context, and if `C` is a class or interface in whose declaration it occurs, then `C.IDENTIFIER(ARGS)` type checks; the type of the expression is the type of `C.IDENTIFIER(ARGS)` for the innermost such class or interface C. The expression is not assignable.
  2. It occurs in non-static context, and for some class or interface `C` in whose declaration this expression occurs, `C.this.IDENTIFIER(ARGS)` type checks; the type of the expression is the type of `C.this.IDENTIFIER(ARGS)` for the innermost such class `C`. The expression is not assignable.

- `super.IDENTIFIER(ARGS)`. Does not type check in static context. In non-static context, let `C` be the innermost enclosing class or interface. If `C` is an interface or the class `Object`, this does not type check. Otherwise let `P` be the direct superclass of `C`. If `IDENTIFIER` occurs inside a constant method or constant constructor, this type checks if there is the most specific accessible constant or static method of name `IDENTIFIER` compatible with `ARGS`, and this method's return type is the type of the expression.

otherwise it is T. Otherwise, the expression type checks if there is the most specific accessible method of name IDENTIFIER compatible with ARGS, and this method's return type is the type of the expression. The expression is never assignable.

6 Templates

In addition to the features of BEJava described above, a template feature is included in BEJava. This feature allows creation of polymorphic methods or types, with polymorphism over constness only being available.

6.1 Polymorphic methods and constructors

The syntax for polymorphic method/constructors is as follows

```
PolymorphicMethod ::=
  'template' '<' VariableList '>' MethodDeclaration

PolymorphicConstructor ::=
  'template' '<' VariableList '>' ConstructorDeclaration

VariableList ::=
  Identifier
  Identifier ',' VariableList
```

It is a compile-time error to repeat a variable in the variable list.

In order to expand a polymorphic method/constructor, first all templates of method, constructors, or types nested within it are expanded; the template declaration is replaced by a distinct method/constructor declaration for each possible assignment of booleans to variables in the VariableList. Inside the template, anywhere that const can appear in the usual grammar for the language, const ? Identifier may appear, where Identifier is the name of one of the variables bound in the template. When the template is expanded, in the newly created declarations corresponding to the binding of Identifier to true, const ? Identifier is replaced by const; in those where the variable is bound to false, const ? Identifier is simply removed.

For example

```
template<a> const?a Object identity(const?a Object o) {
  return o;
}
```

gets expanded as

```
Object identity(Object o) {
 return o;
}

const Object identity(const Object o) {
 return o;
}
```

6.2 Polymorphic types

To declare a polymorphic type, the following syntax must be used:

```
PolymorphicType ::=
  'template' '<' VariableList '>' TypeDeclaration
```

```
TypeDeclaration ::=
  ClassDeclaration
  InterfaceDeclaration
```

The template expansion happens the same way as it does for polymorphic methods and constructors. Namely, first all the templates nested within this one are expanded, then the template declaration is replaced with a separate type declaration for each possible boolean assignment to the variables in `VariableList`. `const ? Identifier` constructs are replaced within the body of the template in the same way as for method and constructor templates.

The only difference between polymorphic types and polymorphic methods is that types created from a template get distinct names. The name of a type created from the template is obtained by taking the original name of the type specified in the template, then appending `< VariableList >` (where `VariableList` is taken from the template declaration), and finally replacing each varible with `const` or nothing, depending on whether the variable is assigned true or false during the creation of this type declaration.

For example

```
template<a,b> class A ...
```

will produce four new classes, `A<,>`, `A<const,>`, `A<,const>` and `A<const,const>`.

In general, the syntax for name now changes to

```
Name ::=
  SimpleName
  Name '.' SimpleName

SimpleName ::=
  Identifier
  Identifier '<' ( 'const' | ) (',' ( 'const' | )* ) '>'
```

Of course, the second production for SimpleName can be used only to name classes or interfaces created during template expansion.

Note that our previous rule that `const ? Identifier` can appear anywhere where `const` can legally appear applies to the syntax for `SimpleName`. For example, `A<const?a,const?b>` is a legal name, which will refer to a different class depending on values of `a` and `b`.