# SubJava pretranslator.

**Warning: It is the not completed product, but skilled realization of pretranslator. Although the program has been tested, it has not passed complete testing! We hope to receive your remarks! If you discover a problem with the program, please report it to Sergey Volotovskiy. The version 06 (from October 15 1999) now is accessible ( subjava.zip or subjava.jar ).**

**Warning: The support of the sponsors for completion the project is required. The financial questions decide with Elya Farberov or Nikolay Kazanskiy.**

## Contents



## Introduction.

When developing computational programs in Java (in particular, for operating complex numbers), the clearness of the program code severely suffers, so complicating the process of program development and maintenance. An essential difference of the program code from the habitual mathematical language is due to the absence in Java of functions and operator overloading (between-object operations ).

## SubJava pretranslator.

SubJava is a pretranslator which enable one to use the following possibilities:

- operator overloading,
- functions,
- global constants and variables,
- implicit data type transformation (implicit data type reduction)

without modification of the specification of language JAVA.

The availability of the above capabilities makes it possible to bring the text of the program closer to the habitual language of mathematics or another applied-field language.
The pretranslator converts the source .subjava file into a standard .java file which may be compiled using any

standard Java compiler.
In convertion, the syntax is checked in a volume necessary for performing operations (i.e., the errors having no effect on the convertion are disregarded).
The pretranslator replaces operator overloading, functions and global variables by constructions admissible in Java (i.e., performs substitutions similarly to macro definitions). In a similar way, the pretranslator performs implicit data type conversions when initializing variable values and when methods and functions are called (if these conversions were declared).
The source text is similar to Java-texts but may contain between-object operations, functions, global variables (constants) and the substitution operator, which points the pretranslator substitution declarations files of admissible operator overloadings, functions, global variables, and implicit data type conversions.

## Substitutions operator.

Substitutions operators are found after the import operators (for each file of substitution declarations there is its own substitutions operator). The operator's format:

```
substitutions < declaration file name >;
```

The name of the substitution declarations file is a Java name (similar to the class name in the import operator), therefore it can be full or partial (considering previous import operators). If the declarations file whose full name coincides with a specified name is absent, it is sought in the packages announced in the import operators (i.e. an attempt is made to find a file whose full name consists of the name of the package announced in import and a specified name). Example:

```
substitutions forDouble;
substitutions mySubstitutions.forDouble;
```

## The substitution declarations file.

The substitution declarations files are realized as text files with the enhancement substitutions.
Substitution declarations (hereafter, referred to as declarations) within the file are placed in succession in any order.
The declarations may be found in one or several lines. There may be more than one declaration in one line. Any number of separating blanks is allowed.
Declarations files may contain comments analogous to the comments found in JAVA-programs (with one exception: if a comment is found in the pattern field, it will be included in the pattern and, hence, will be inserted in the text of .java file when applying this substitution).
While determining the full name of data type (operand type or arguments type) the pretranslator takes into account the import operators found in the initial text. This allows one to make the declaration shorter (instead of java.math.BigInteger one may use BigInteger).
Syntax of declaration patterns is checked. In the case of error, diagnostics is output.
Now set of the allowable operators and their precedence and associativity are defined by the Java specification.
The following types of declarations are admissible (the declaration type, the format and one or two examples are given):

- prefix operator declaration

  ```
  prefixOperator : < operator > < operand type > : < pattern > :
  prefixOperator : ++double : ++{0} :
  ```

- postfix operator declaration

  ```
  postfixOperator : < operand type > < operator > : < pattern > :
  postfixOperator : java.lang.Double++ : new Double( {0}.doubleValue()++ ) :
  ```

- binary operator declaration

```
binaryOperator : <operand type><operand><operand type> : <pattern> :
binaryOperator : java.lang.Double+java.lang.Double : {0}.doubleValue()+{1}.doubleValu
```

- friendly binary operator declaration (independent of the sequence of operands);
  The pattern is changed automatically; this type of declaration replaces two declarations of binary operation, thus, on one side reducing the number of necessary declarations and on the other (and main) side, allowing the reduction of the number of errors while generating the full set of declarations

```
friendOperator : <operand type><operand ><operand type> : <pattern>  :
friendOperator : java.lang.Double+double : {0}.doubleValue()+{1} :
```

- function declaration (it is only the function name which is replaced in this case, the arguments line remains unchanged)

```
function : <function name> : (<argument types>) : <method  name pattern>  (...) :
function : SIN( double ) : Math.sin( ... ) :
function : FUN( double, int ) : MyFunction.fun( ... ) :
```

- function declaration ( the entire expression is replaced in this case)

```
function : <function name> : (<argument types>) : <pattern>  :
function : SIN( java.lang.Double ) : Math.sin( {0}.doubleValue() ) :
function : FUN( double, int ) : MyFunction.fun( {1}, {0} ) :
```

- global variable and constant declaration (global names)

```
name     : <global name> : <pattern>  :
name     : PI  : Math.PI :
```

- implicit data type convertors declaration (the data type of the result is determined automatically)

```
convertor : <data type from  which to convert> : <pattern>  :
convertor : double : (int){0} :                  // double -> int
convertor : java.lang.Double : {0}.intValue() :  // Double -> int
```

## Use of substitutions by the pretranslator.

The substitutions of the operators are used at definition such as result of operator in expression: the required operator with the given types of operands (type of an operand for unary operator) is searched in the list of substitutions of the operators and, if not have found, the type of operator is defined by rules of implicit transformation in JAVA. If the type of result of operation did not manage to be defined - the message is given out.
The substitutions of functions are applied at definition such as returned value by a call of a method in expression: the method with a required name and list of types of parameters is searched in the list of methods of a class, in the list of methods of a super class, and if not have found, in the list of substitutions of functions; all satisfying methods (and among substitutions - functions) and then among found are determined, using algorithm of the exactest coincidence, leave superfluous; if the number of the stayed methods (functions) is distinct from 1 - the message is given out. By search of satisfactory methods and functions the substitutions of implicit converters of a type are used.
The substitutions of global variable and constants are applied at definition such as variable in expression: the

required name is searched in the list of the variable of the block, method, in the list of fields of a class, super class and interfaces, in the list of substitutions global variable and constants up to the first result (i.e. as soon as have found the further search have stopped). If not it was possible to define a type of variable - the message is given out.
The substitutions of implicit converters of a data type are used at initialization variable in the operators of the declaration such as or at transformation of formal parameters in actual: in case of discrepancy of types ( from type and to type) the converter is searched. If not it was possible to find of the converter - the message is given out.

## Disadvantages and limitations of realization.

The pretranslator's operation requires the time commensurable with the time needed for the Java compiler, since here are checked syntax (not fully), the availability of all methods, variables and fields (while defining their data types). But the greatest time is needed for download of necessary classes (both pretranslator classes and those used in the file to be converted).
Note that although it turns out to be inessential for the majority of situations, all the classes referred to in the file under translation should be compiled prior to running the pretranslator (as opposed to the java compiler, the pretranslator does not check the date of creation of the bite-code of classes and interfaces).
The pretranslator is realized in JAVA 1.1 (not for use with JAVA 1.0)

## Main advantages of the SubJava pretranslator.

The pretranslator allows one to write down the program code in JAVA making use an applied-field language (e.g., by bringing the program code closer to the mathematical language). It allows one to emulate

- operator overloadings,
- functions,
- global variables and constants,
- implicit conversion of types,

as if these had already been introduced in JAVA ( without modifications of Java compiler or Java Virtual Machine (JVM) ).
The present realization supports all capabilities of JAVA 1.1:

- implicit conversion of standard data types;
- heritage;
- declaration of several classes and interfaces in a single file;
- use of built-in classes and interfaces;
- use of local and anonymous classes.

The pretranslator has been written in JAVA, which makes it independent of the software-hardware environment.
The pretranslator is called from the command line, which makes it possible to use it in available programming environment (IDE).
A multi-language support is realized, thus allowing the translation of diagnostics into the user's language.

## Extra benefits of the SubJava pretranslator.

A long-debated problem in programming is "where to place methods for realizing operations overlay ?". In this program, all limitations on the location and naming of realizing methods are removed, thus allowing one to define operator overloadings without introducing changes in their classes (e.g., by creating a new class that contains necessary methods, or (see Example 1. for java.lang.Double) by describing everything in the substitution declarations file).
This pretranslator may be used when translating the program from C++ into JAVA. For this to do, one needs to

create declarations file for employed functions , operator overloadings, global variables and constants, necessary implicit data type conversions. This done, one generates a Java program from the texts written in C++ and then processes the resulting text by the pretranslator.
In converting, the structure of the initial .subjava file (indents, corresponding line indexes) are, where possible, preserved in a java file, thus simplifying analysis of the errors to be discovered by the Java compiler while translating the .java file.

## Installation and use of SubJava.

**Warning: It is the not completed product, but skilled realization of pretranslator. Although the program has been tested, it has not passed complete testing! We hope to receive your remarks!**

At present, the SubJava pretranslator is realized as a set of java packages. Therefore, for the purpose of installing, it will suffice to unwrap the archive ( subjava.zip ) using one of the roots indicated in classpath, which means that the SubJava .class file is to be located in the catalogue corresponding to the java package of subjava.*.

Command line call :
java subjava.SubJava <options> <file name>

It is possible to use jar-archive subjava.jar ( in this case there is no necessity to unpack archive).

Command line call :
java -jar <archive name> <options> <file name>

<archive name> - name of jar-archive with the path (for example d:\Java\subjava.jar).

<options>:

- -errors max number of deduced mistakes (default - 20)
- -compiler template string for call the compilator to bytecode (where {0} changed into file name without extension .java !!! ) (default - undefined)
- -output toggle to define output stream (System.out/System.err) (default - System.out)

The parameters can follow in any sequence.

<file name> - file name with the obligatory extension subjava. The file under processing is to be located in the current directory.

Examples of command line calls:

```
java subjava.SubJava a1.subjava

java subjava.SubJava -errors 10 a2.subjava

java subjava.SubJava -compiler "javac {0}.java" a3.subjava

java subjava.SubJava -output err a4.subjava

java -jar d:\Java\subjava.jar a1.subjava

java -jar c:\Java\subjava.jar -errors 10 a2.subjava

java -jar c:\Java\subjava.jar -compiler "javac {0}.java" a3.subjava

java -jar d:\Java\subjava.jar -output err a4.subjava
```

## The answers to some questions.

### 1. Why the form of pretranslator to Java is chosen?

The opponents of operator overloading assert, that using of operator overloading leads to ambiguity of interpretation of source code, that contradicts Java idealogy. And in some cases they are right. The inserting of operator overloading in the form of pretranslator to Java, allows to remove this lack (code in *.java, into which the code from *.subjava will be converted will have unequivocal interpretation).

### 2. Whether it is possible to set the precedence and associativity of operators?

The authors do not see necessity of such opportunity, because on their sight it really will lower clearness of the source text. Now the precedence and associativity of operators is determined by the language standard Java. The introduction of this opportunity basically is feasible.

### 3. Why there are no analogues template and enum as in C++ in pretranslator?

It is the first realization of pretranslator. At the following realization these needs will be taken into account. Now development of this project is suspended because of absence of financing, therefore when it will be while it is not known.

## Example 1.

This example illustrates the introduction of operations for the existing class java. lang. Double.

Text of forDouble.substitutions file:

```
convertor      : Double : {0}.doubleValue() :  //  Double -> double
convertor      : double : new Double( {0} ) :  //  double -> Double

binaryOperator : Double + Double :
                   new Double( {0}.doubleValue() + {1}.doubleValue() ) :
friendOperator : double + Double : new Double( {0} + {1}.doubleValue() ) :

binaryOperator : Double * Double :
                   new Double( {0}.doubleValue() * {1}.doubleValue() ) :
friendOperator : double * Double : new Double( {0} * {1}.doubleValue() ) :

binaryOperator : Double - Double :
                   new Double( {0}.doubleValue() - {1}.doubleValue() ) :
binaryOperator : double - Double : new Double( {0} - {1}.doubleValue() ) :
binaryOperator : Double - double : new Double( {0}.doubleValue() - {1} ) :

binaryOperator : Double / Double :
                   new Double( {0}.doubleValue() / {1}.doubleValue() ) :
binaryOperator : double / Double : new Double( {0} / {1}.doubleValue() ) :
binaryOperator : Double / double : new Double( {0}.doubleValue() / {1} ) :

binaryOperator : double += Double :  {0} += {1}.doubleValue() :
```

Text of Test.subjava file:

```
substitutions  forDouble;

public class Test {

   public static void main( String[] args ) {
      Double d1 = 1,  d2 = 5;
      Double d3 = d1 + d2;
      d1 = 5 - d2;
      double d = d3;
      d += ( d * d1 ) / 7;
   }

}
```

Text of the resulting Test.java file:

```
public class Test {

   public static void main( String[] args ) {
       Double d1 = new Double( 1 ),  d2 = new Double( 5 );
       Double d3 = new Double( d1.doubleValue() + d2.doubleValue() );
       d1 = new Double( 5 - d2.doubleValue() );
       double d = d3.doubleValue();
       d += new Double( new Double( d * d1.doubleValue() ).doubleValue() / 7 ).doubleValue
   }

}
```

## Example 2.

This example illustrates the introduction of operations for the class Complex. Similarly, one may introduce functions and operations between any objects.

Text of forComplex.substitutions file:

```
//  substitutions for Complex
convertor      : double : new Complex( {0}) :  //  double -> Complex

binaryOperator : Complex + Complex : {0}.add( {1} ) :
friendOperator : double + Complex  : {1}.add( {0} ) :

binaryOperator : Complex * Complex : {0}.mul( {1} ) :
friendOperator : double * Complex  : {1}.mul( {0} ) :

binaryOperator : Complex - Complex : {0}.sub( {1} ) :
binaryOperator : double  - Complex :
                    new Complex( {0} - {1}.real(), -{1}.image() ) :
binaryOperator : Complex - double  : {0}.sub( {1} ) :

binaryOperator : Complex / Complex : {0}.div( {1} ) :
binaryOperator : double  / Complex : new Complex( {0} ).div( {1} ) :
binaryOperator : Complex / double  : {0}.div( {1} ) :

name           : IM  : Complex.I :

function : sin( Complex ) : {0}.sin() :
function : cos( Complex ) : Complex.cos( {0} ) :
```

Text of TestForComplex.subjava file:

```
substitutions  forComplex;

public class TestForComplex {

   static Complex im = IM;
   Complex p1 = sin( im );
   Complex p2 = cos( im );

   public static void main( String[] args ) {
      Complex c1 = 1,  c2 = -5;
      Complex c3 = new Complex( 1, 1 );
      Complex c4 = c1 * c2 / c3;
      Complex c5 = 7 / 8;
      Complex c6 = 7 * c2 / 8;
      Complex c7 = sin( c1 ) * cos( c2 ) / c3;
   }

}
```

Text of the resulting TestForComplex.java file:

```
public class TestForComplex {

   static Complex im = Complex.I;
   Complex p1 = im.sin();
   Complex p2 = Complex.cos( im );

   public static void main( String[] args ) {
      Complex c1 = new Complex( 1),  c2 = new Complex( -5);
      Complex c3 = new Complex( 1, 1 );
      Complex c4 = c1.mul( c2 ).div( c3 );
      Complex c5 = new Complex( 7 / 8);
      Complex c6 = c2.mul( 7 ).div( 8 );
      Complex c7 = c1.sin().mul( Complex.cos( c2 ) ).div( c3 );
   }

}
```

## Example 3.

This example illustrates the introduction of standard functions.

Text of forMath.substitutions file:

```
//  substitutions for Math

name           : PI  : Math.PI :

function : sin( double )   : Math.sin( ... ) :
function : cos( double )   : Math.cos( ... ) :
```

Text of TestForMath.subjava file:

```
// example TestForMath

substitutions  forMath;
```

```
public class TestForMath {

   static double pi = PI;

   public static void main( String[] args ) {
      double p1 = sin( pi );
      double p2 = cos( 2*pi );
   }

}
```

Text of the resulting TestForMath.java file:

```
// example TestForMath

public class TestForMath {

   static double pi = Math.PI;

   public static void main( String[] args ) {
      double p1 = Math.sin( pi );
      double p2 = Math.cos( 2*pi );
   }

}
```

## From the version to the version.

### The version 04 (from Jule 30 1999)

### The version 05 (from September 10 1999)

The pretranslator is adapted to JDK 1.2.2 .

The small discrepancy in processing CLASSPATH is eliminated.

### The version 06 (from October 15 1999)

The processing of local and anonymous classes is added.

More correct work with "path".