

The NetRexx Interpreter

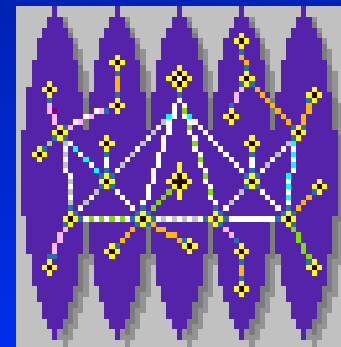
<http://www2.hursley.ibm.com/netrexx/>

RexxLA / WarpTech -- 26 May 2000

Mike Cowlshaw

IBM Fellow

mfc@uk.ibm.com



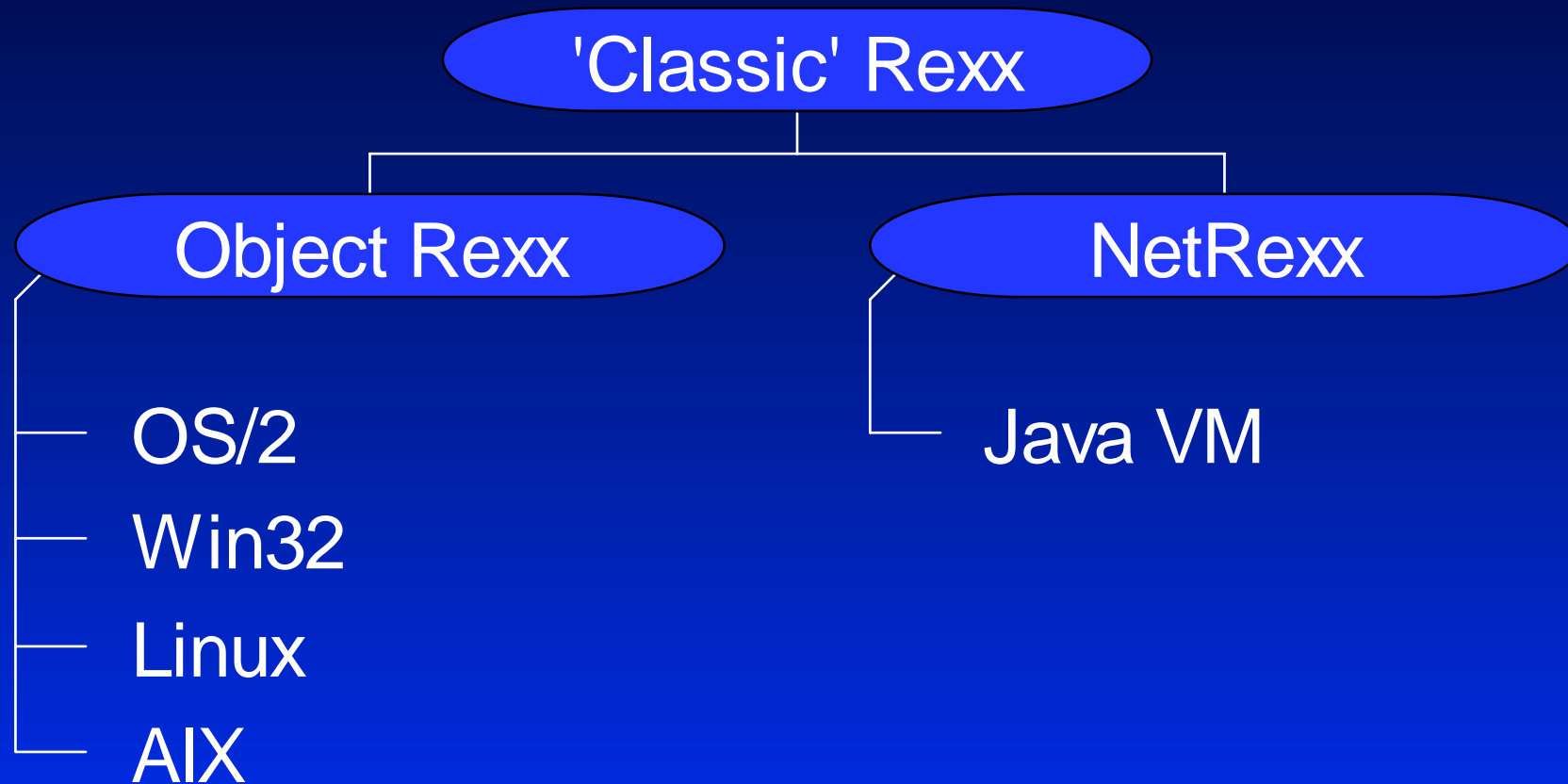
Overview

- Introduction to NetRexx
- Demo. -- compiling and interpreting NetRexx programs
- The compiler/interpreter implementation
- Questions?

What is NetRexx?

- A complete *alternative* to the Java language, for writing classes for the Java Virtual Machine
- Based on the simple syntax of Rexx, with Rexx decimal arithmetic
- Fully exploits the Java object model, exceptions, and binary arithmetic
- Automates type selection & declaration
- Removes many historical quirks

The Rexx language family



NetRexx Java implementation

- Current implementation first *translates* NetRexx to accessible Java source, or *interprets* it directly (or both)
- Runs on any Java platform
- Any class written in Java can be used
 - GUI, TCP/IP, I/O, DataBase, *etc.*
- Anything you could write in Java can be written in NetRexx

. . . and it's free.

NetRexx programs

toast.nrx

```
/* This wishes you good health. */  
say 'Cheers!'
```

Control constructs

```
if answer='yes' then say 'OK!'  
else say 'shucks'
```

```
loop i=0 for mystring.length  
  say i ':' mystring[i]  
end i
```

also do. . end for simple grouping, with
label for leave

Control constructs - select

```
select case i + 1
  when 1, 2 then say 'small'
  when 3 then say 'medium'
  otherwise say 'large'
end
```

*(The usual Rexx **select** without case is also supported, and select may have a label)*

Strings - the base type

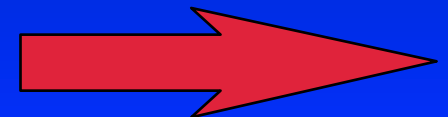
- Strings in NetRexx are of type *Rexx*
 - by default, data and numbers are strings
 - standard methods from Object Rexx
 - conversions
- Automatic inter-conversion with Java String class, char and char[] arrays, and numeric primitives (optional)

Arithmetic

- Preferred arithmetic is from ANSI Rexx
- Decimal, just one type of number
 - follows human rules ($2 * 1.20$ is 2.40)
 - gives exact results when expected (*e.g.*, for 0.1 , 0.3)
 - no overflow at binary boundaries
 - arbitrary precision

numerical digits 300

say $1/7$



numeric digits 300

0. 14285714285714285714285714285714285714
2857142857142857142857142857142857142857
1428571428571428571428571428571428571428
5714285714285714285714285714285714285714
2857142857142857142857142857142857142857
1428571428571428571428571428571428571428
5714285714285714285714285714285714285714
2857142857142857142857142857142857142857
142857142857142857142857142857142857

Binary classes and methods

- The **binary** keyword instructs the compiler to use native (binary) arithmetic types and operations (boolean, byte, int, long, float, *etc.*)
- Achieves the full speed of the Java Virtual Machine and JIT compilers
- No performance penalty for using NetRexx instead of Java

Explicit typing

- Casting/conversions use the *blank* (concatenation) *operator*

```
number=i nt 7*y    -- number is an i nt  
number2=i nt      -- variable declarati on
```

- Consistently extends to method arguments

```
method size(x=i nt, y=i nt, depth=i nt 3)
```

Other features from Rexx

- Case-insensitivity
- Parse
- Trace (methods, all, results)

```
2 *=*      number=1/7
```

```
>v> number "0.142857143"
```

```
3 *=*      parse number before '.' after
```

```
>v> before "0"
```

```
>v> after "142857143"
```

```
4 *=*      say after '.' before
```

```
>>> "142857143.0"
```

Exceptions

- Semantics from Java
- Generalized and simplified syntax (extends all existing control constructs)

```
say 'Please enter a number:'
number=ask    -- read a line
do
  say 'reciprocal is: ' 1/number
catch Exception
  say 'Sorry, could not divide' -
    "'number'" into 1'
end
```

NetRexx JavaBean support

- JavaBean (indirect) properties

```
properties indirect  
    filling=Color.red
```

generates (or checks):

```
method getFilling returns java.awt.Color  
    return filling  
method setFilling($1=java.awt.Color)  
    filling=$1
```


NetRexx Inner Class support

- Minor and Dependent classes

```
class Foo
  x=Bar()
  y=Foo.Bar null
  z='Hello'
  x.Counter
```

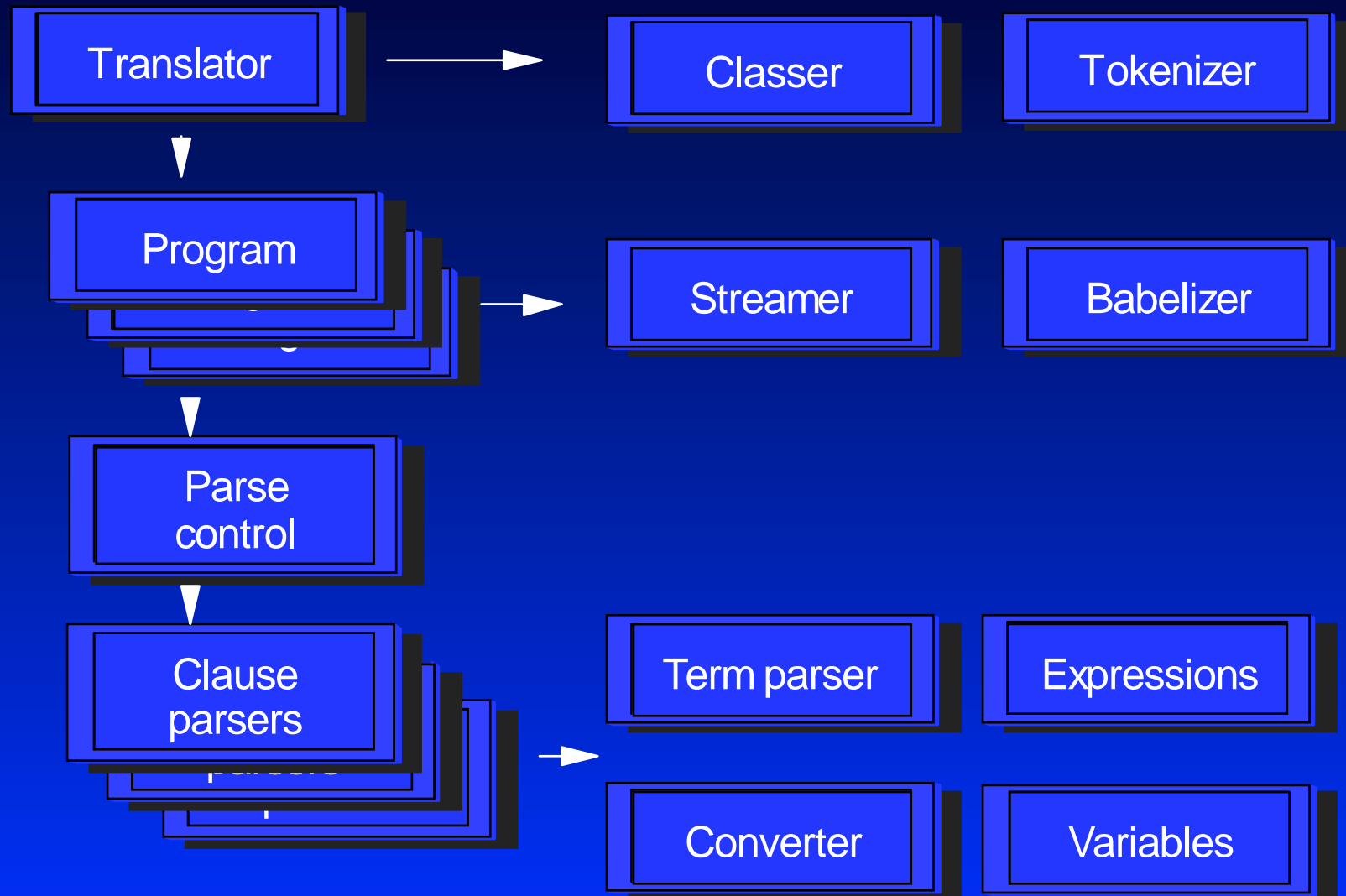
```
class Foo.Bar dependent extends Another
  method Counter
    say parent.z
```

Demonstration ...

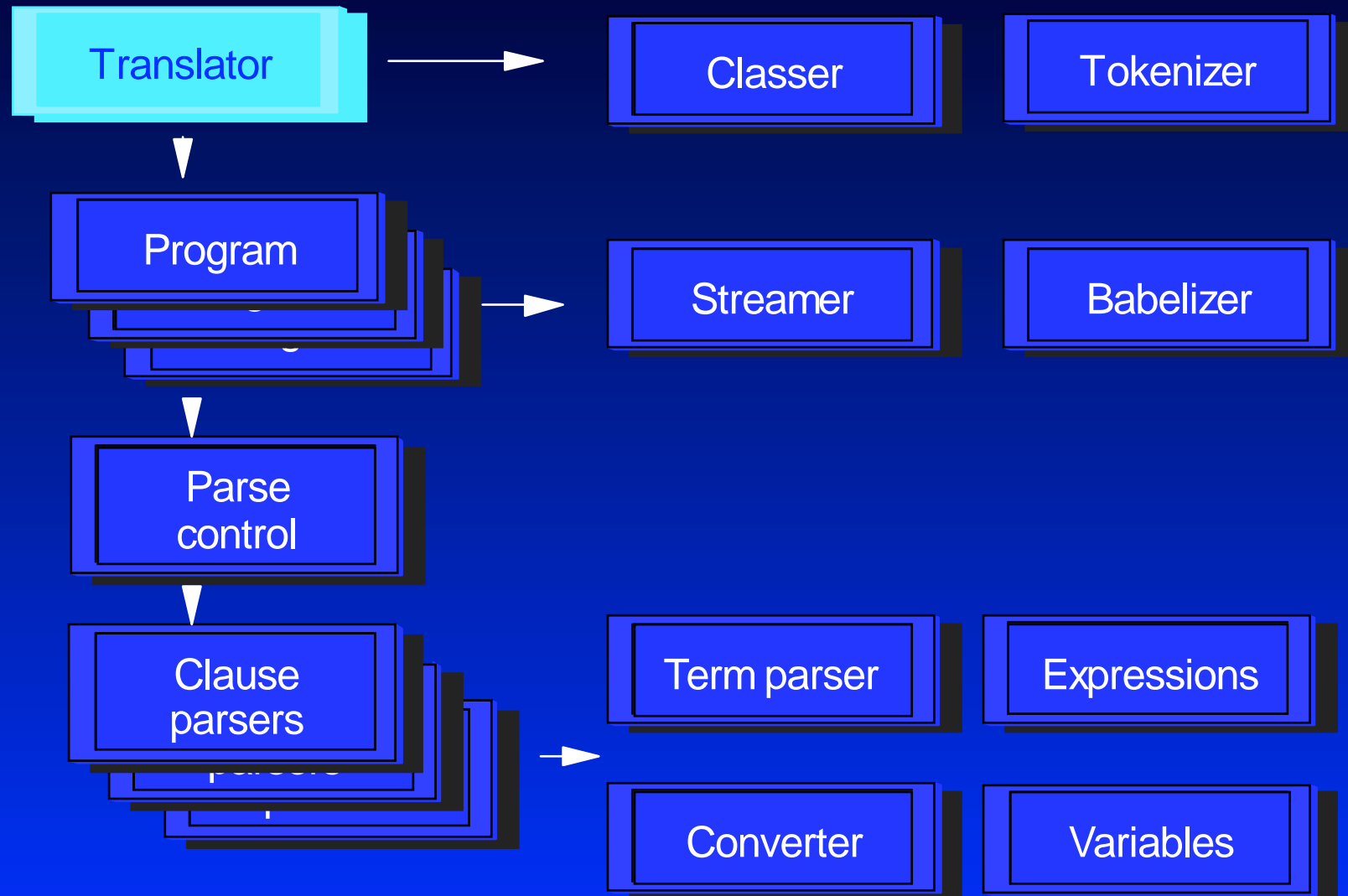
So how does it work?

- Unconventional organization
- Structured like an interpreter rather than a compiler
- Parsing is not carried out 'up front', but on demand
- Parsing is identical for translation to Java or for direct interpretation, with full error checking at the point of parsing

Overall translator organization



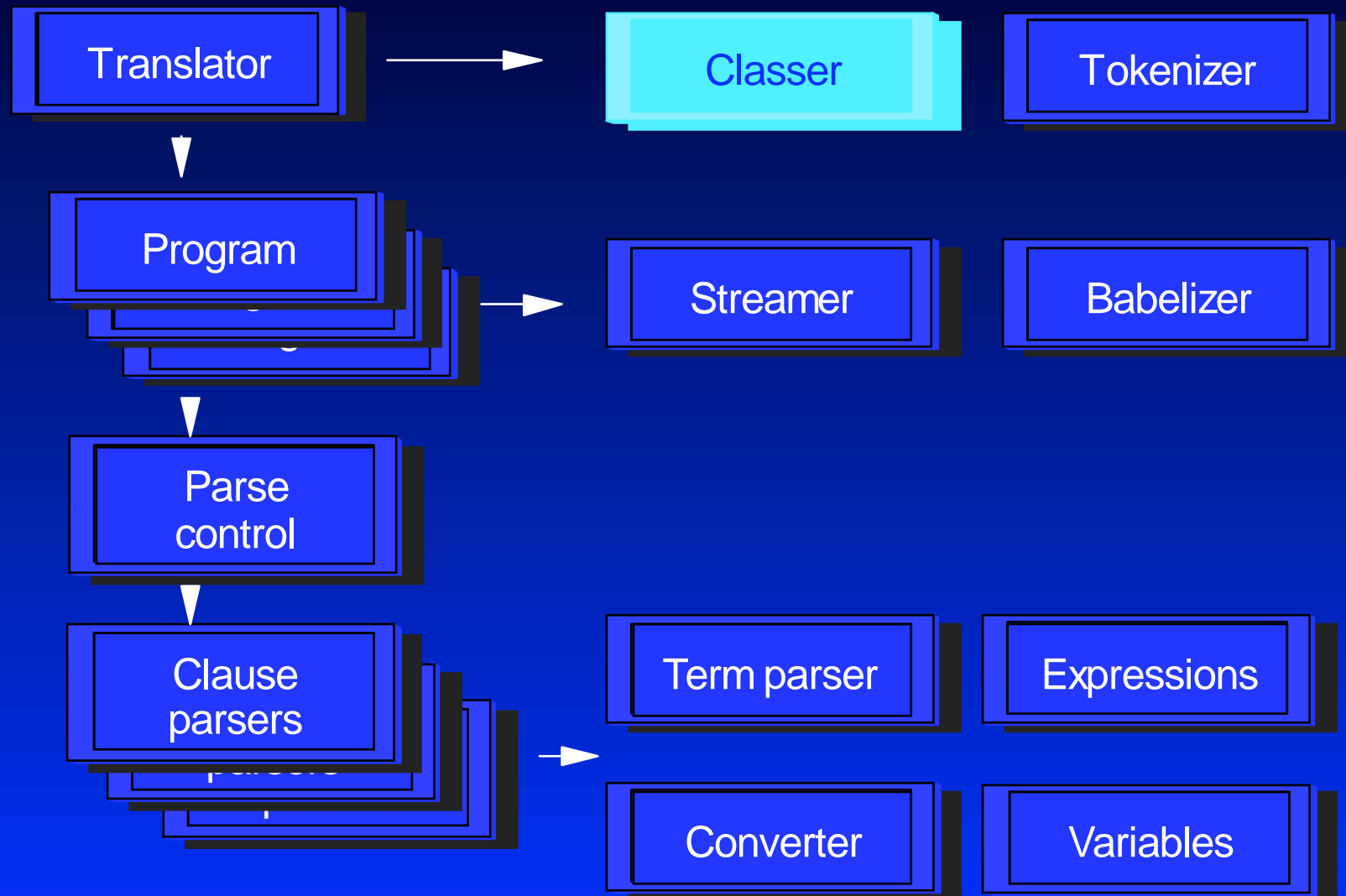
Overall translator organization



Translator

- Internal API for NetRexxC to use
- Factory, language, and programs setup
- Cross-program pass control (3 main passes)
- Manages compilation using javac
- Manages interpretation
- Top-level error handling

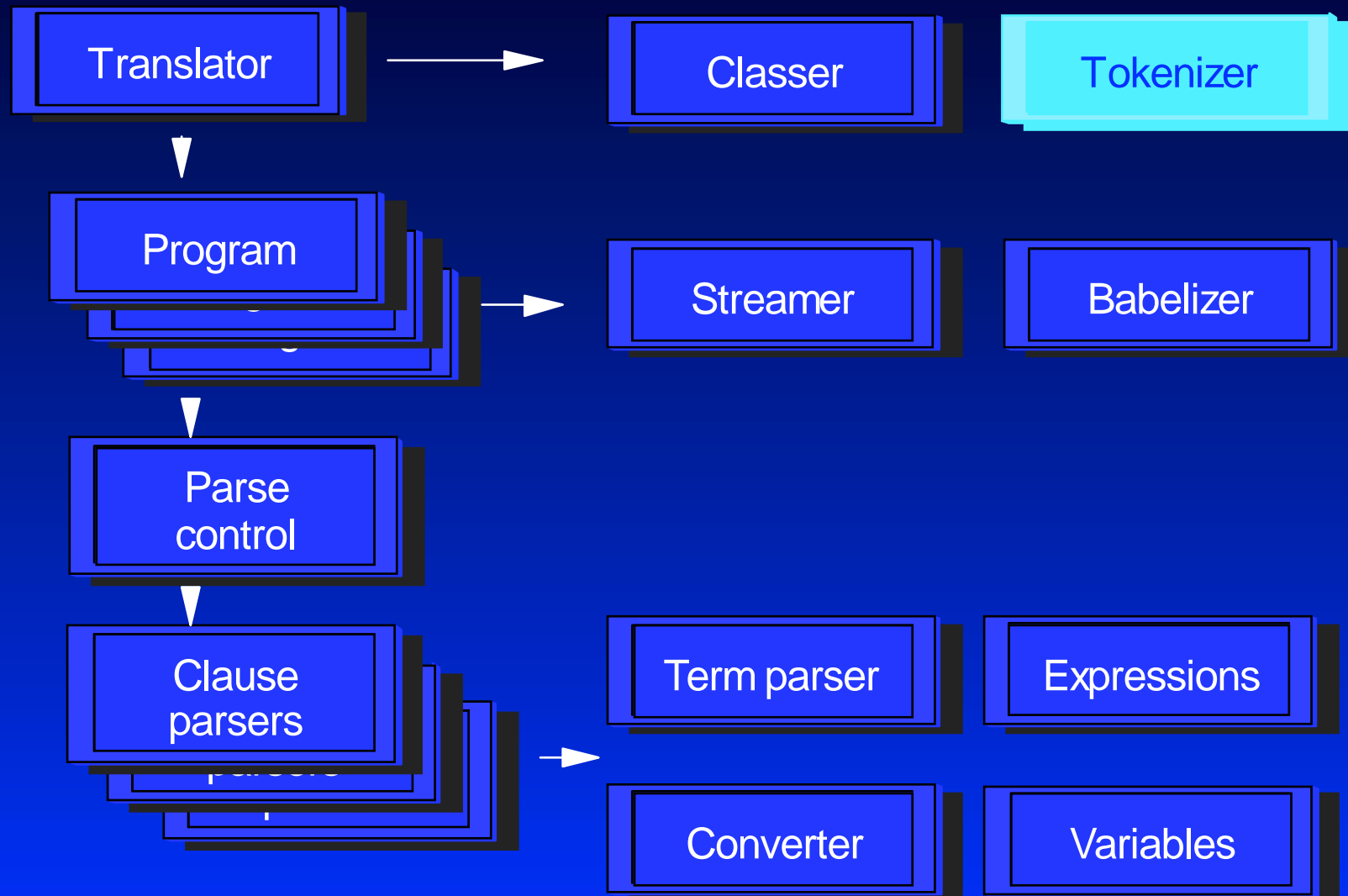
Overall translator organization



Classer

- Most difficult area of translation, due to changes in Java core over time
- In general 'owns' the external namespace
- Manages class path, ambiguous classes, *etc.*
- Locates, reads, and parses class images
- Locates methods and properties, based on costing algorithm

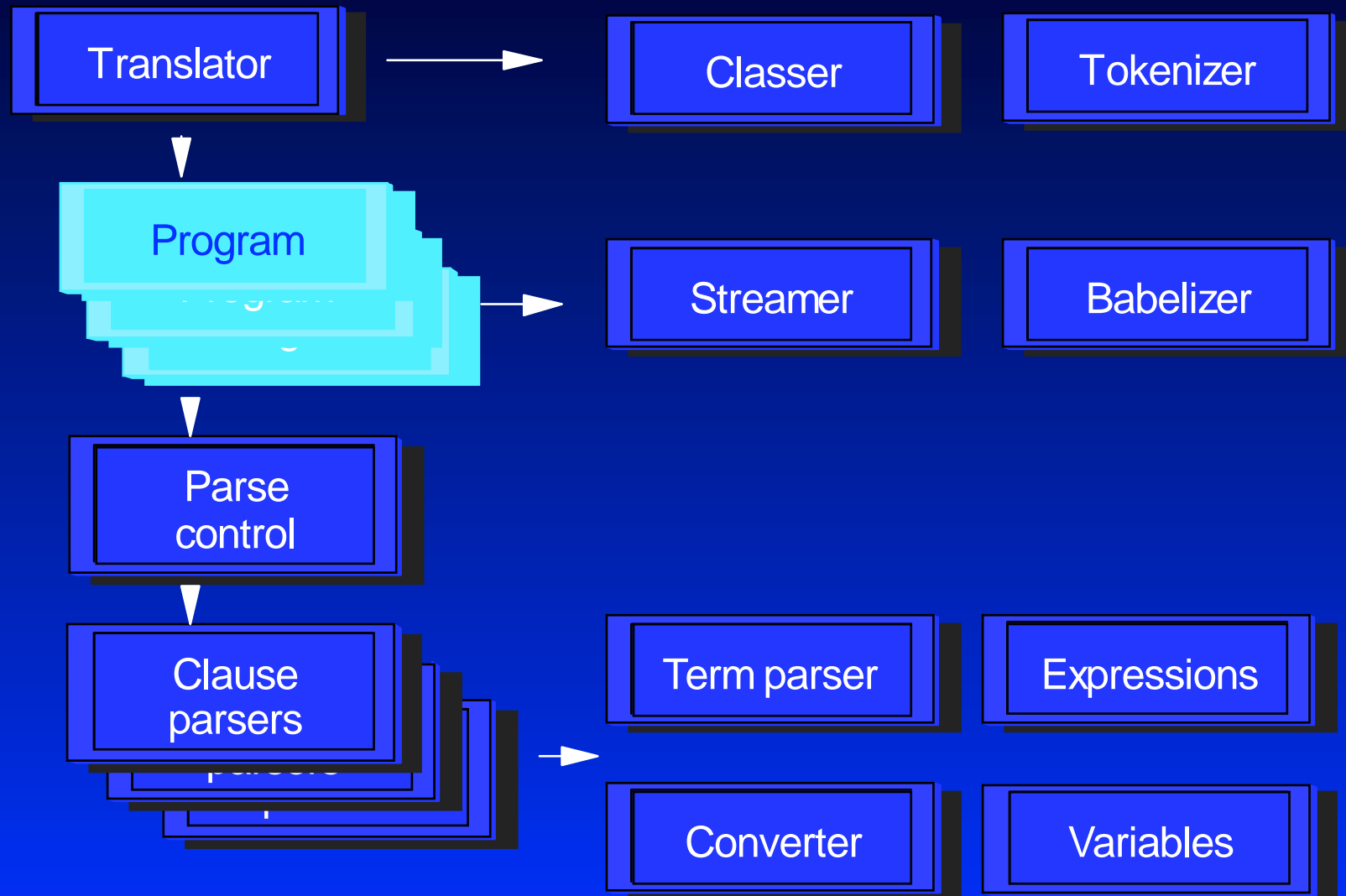
Overall translator organization



Tokenizer

- One of several shared resources
- Language-independent tokenizing of an input stream or array of character arrays
- Other shared resources include:
 - error message editor
 - base internal types (Tokens, Flags, Types, *etc.*)
 - trace code generator
 - interfaces (ClauseParser, ProgramSource, *etc.*)

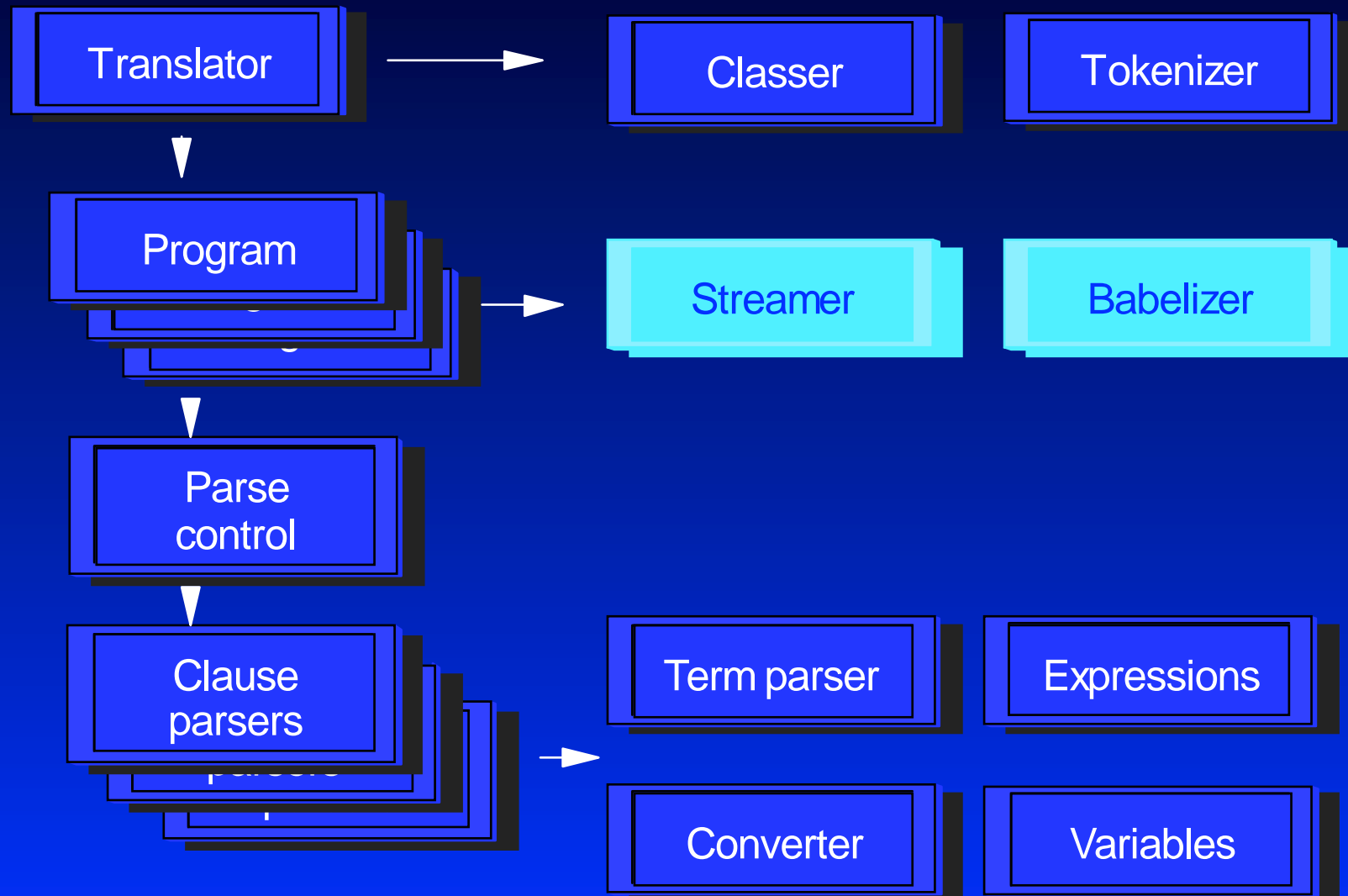
Overall translator organization



Program

- Represents exactly one of the programs being translated
- Each program may be in a different language, with different syntax (and different semantics at the statement level)
- Holds program-level objects (streamer, package information, imports, options, *etc.*)

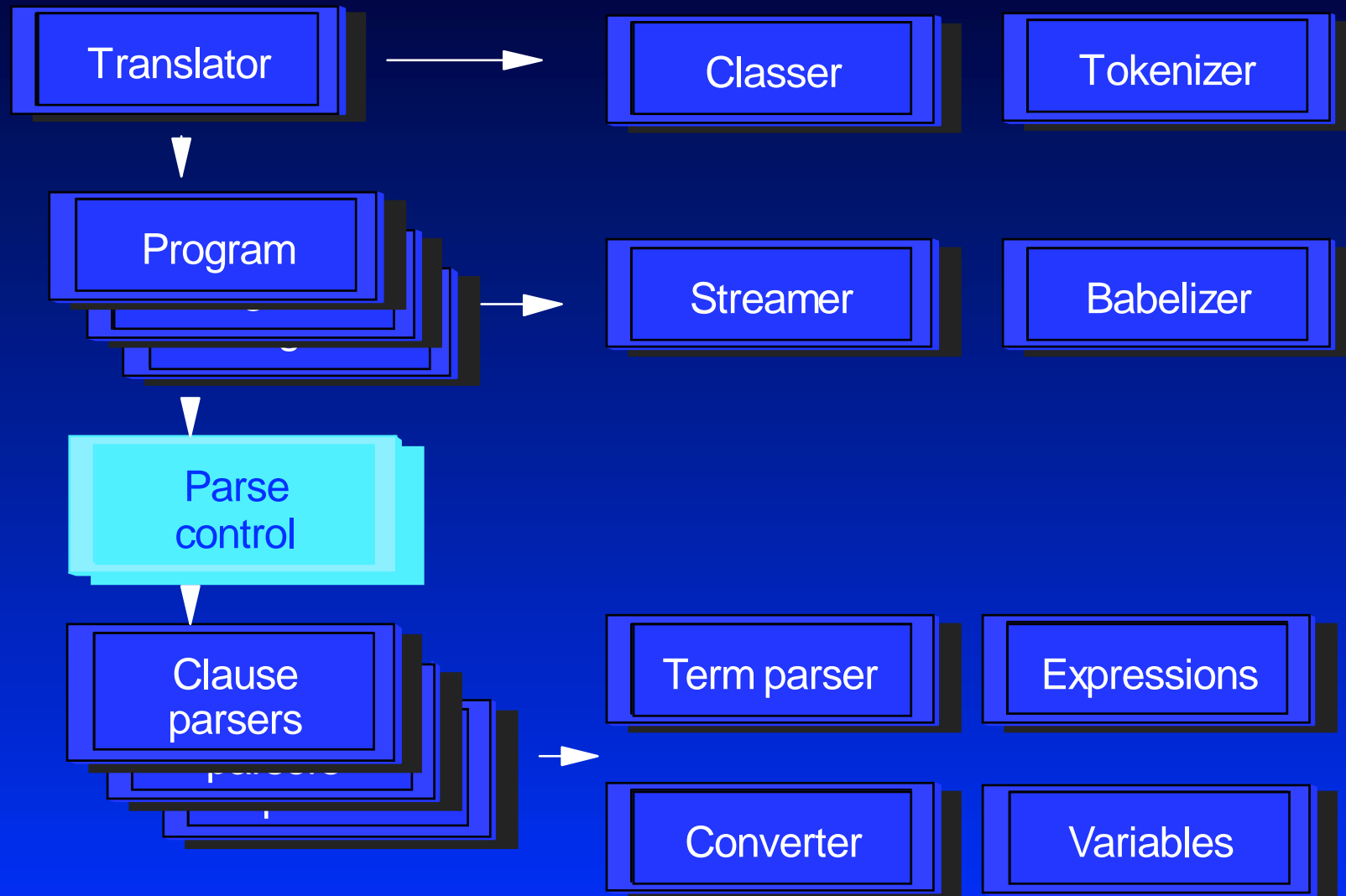
Overall translator organization



Streamer and Babelizer

- Streamer handles input and output streams
 - locates input files
 - names and creates output files
 - checks for conflicts
 - reads files on demand
- Babelizer converts internal representations to viewable strings, depending on the language
 - associates file extensions with languages
 - arrays shown as **[]** or **[,]** or **(,)**
 - attributes spelled as appropriate for the language; e.g., **shared** or **Friend**

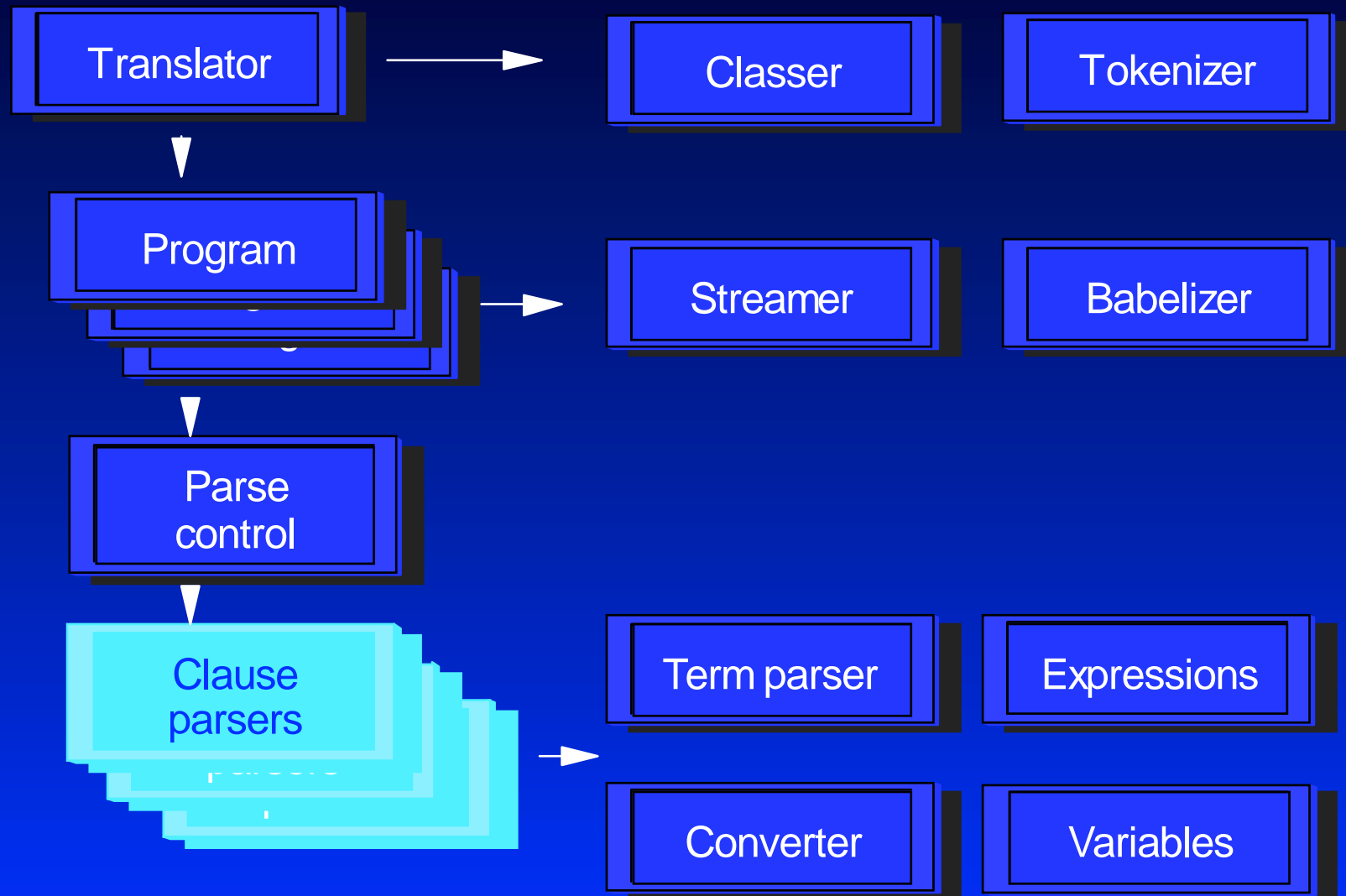
Overall translator organization



Parse control

- State machine for static parsing
- Language-dependent (hence one instance per program)
- Three levels of parsing, deferred where possible:
 - parseProgram
 - parseClassBody
 - parseMethodBody
- Parsing-related utilities (pushLevel, popLevel, *etc.*)

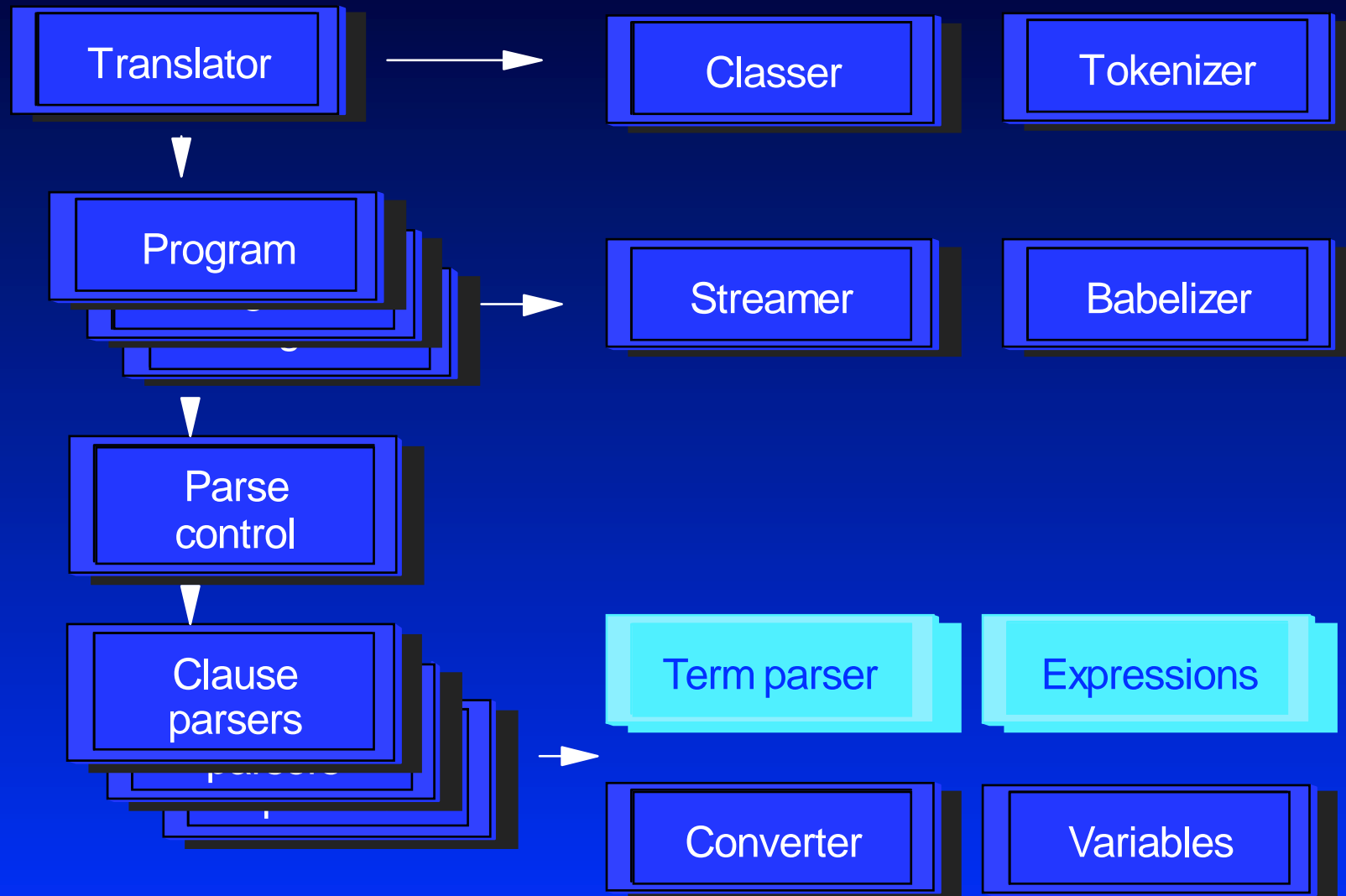
Overall translator organization



Clause parsers

- Each knows about a single clause in one language (Do, Catch, End, Nop, Say, *etc.*)
- Each has a **scan** method (lexical parse)
- Each has a **generate** method, for Java code
- Each has an **interpret** method
- **generate** and **interpret** share information gleaned during **scan** (which may have been multi-pass)

Overall translator organization



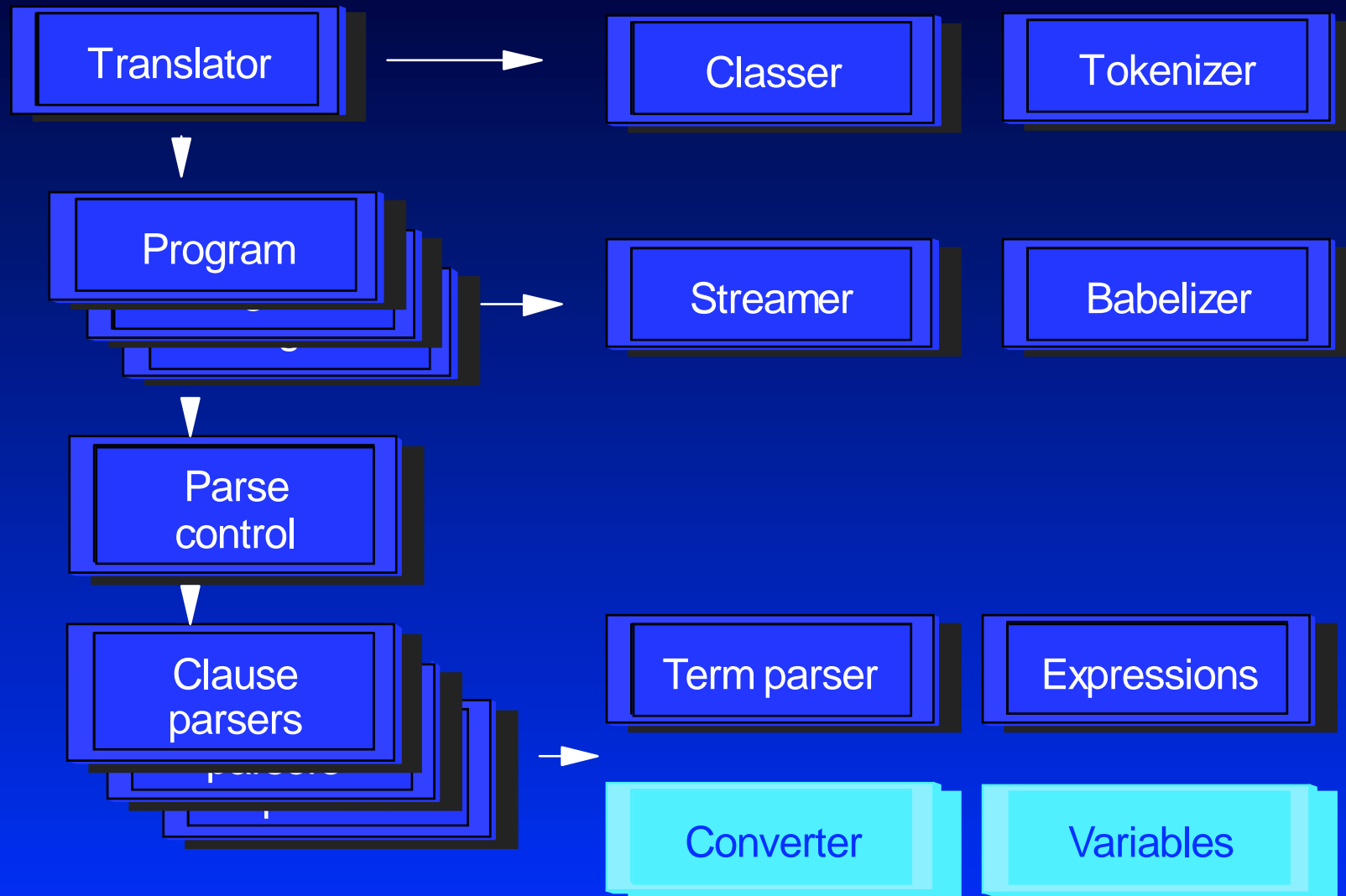
Term and Expression parsers

- Recursively call each other to parse terms and expressions. For example:

```
(Rexx vector.get('key')).substr(i+1, j)
```

- Term parser is more complicated than Expression parser, and is easily the largest class in the translator (100K characters, including comments)
- Like clause parsers, both can emit Java code or execute (interpret) the term or expression

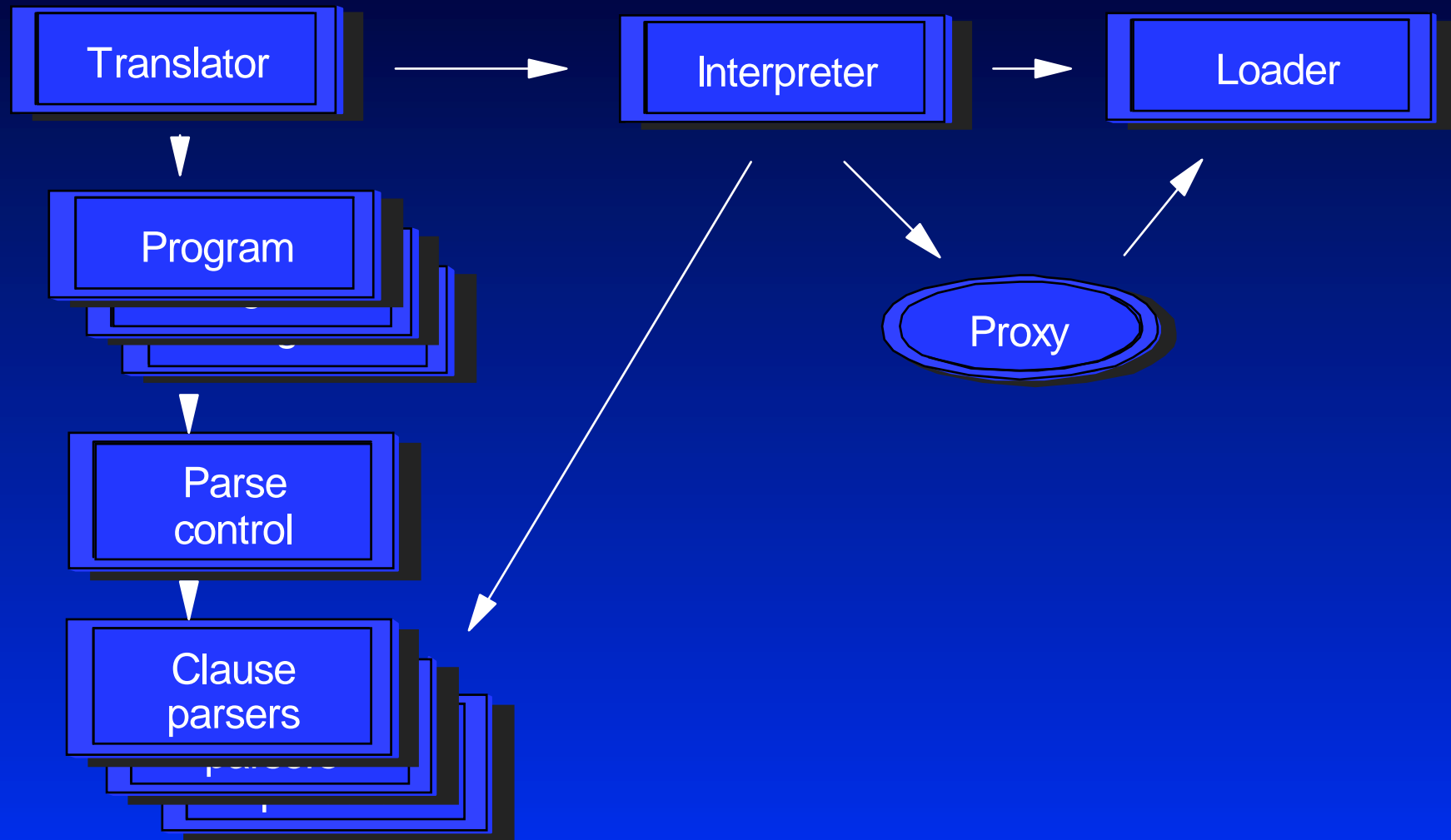
Overall translator organization



Converter and Variable manager

- Converter understands type inferences
 - costs conversions (used for method finding and error checking)
 - effects conversions (emits Java code or interprets)
- Variable manager handles both class and method variables
 - All properties and local variables during scan passes
 - Only static (Class) properties and local variables during interpretation - instance properties are held in a real object

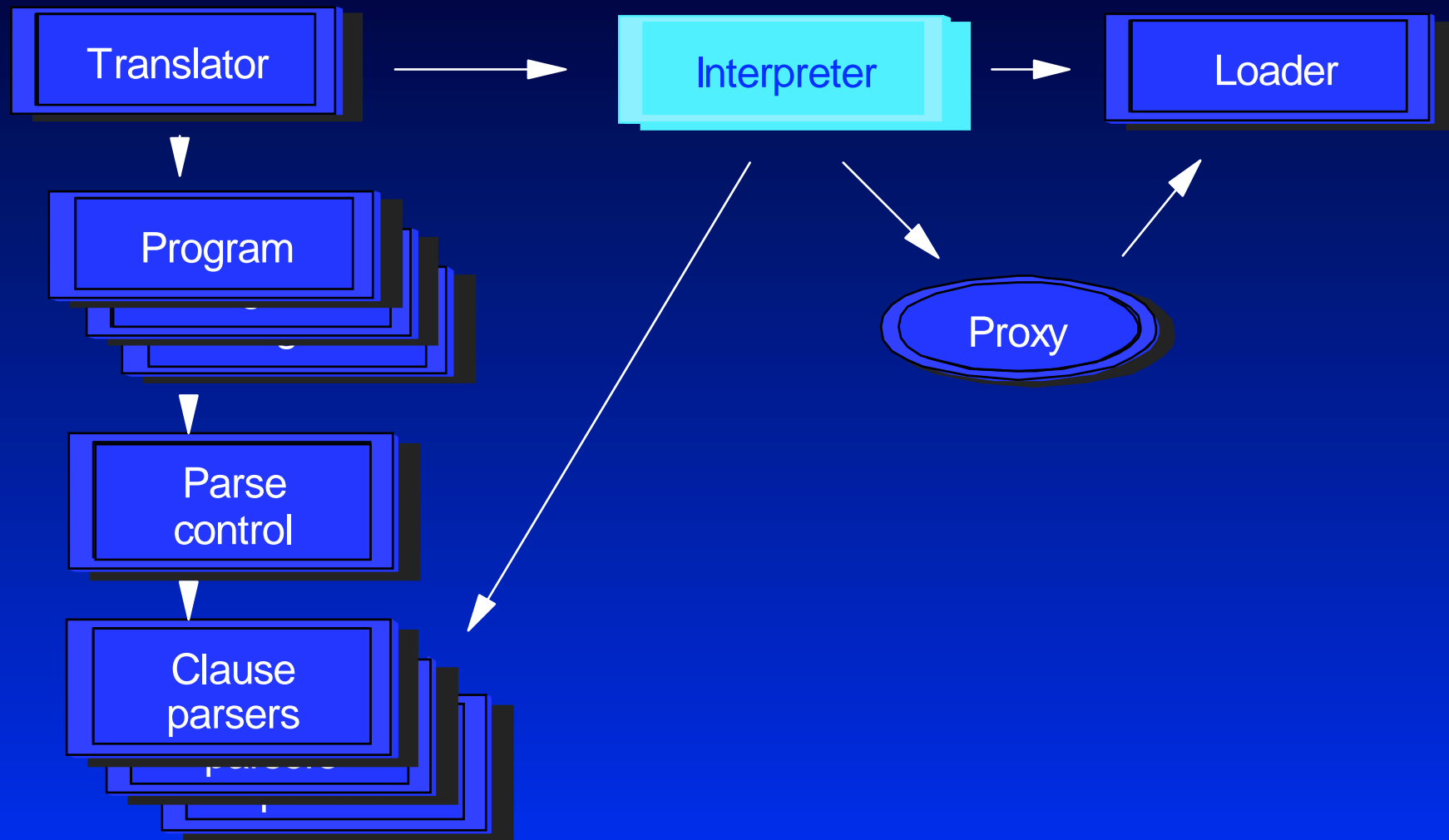
Interpretation



General principle

- First, programs are parsed (to determine classes, properties, and methods with their signatures)
- For each class, a *proxy* (stub) class is created
 - this has all the properties just as in a 'real' class
 - for each method, it has only the definition and return
 - when a method is invoked through Java reflection, it immediately calls the interpreter, which interprets the method body
- Real instances are created, so interpreted classes are visible to the JVM for callbacks, *etc.*

Interpretation



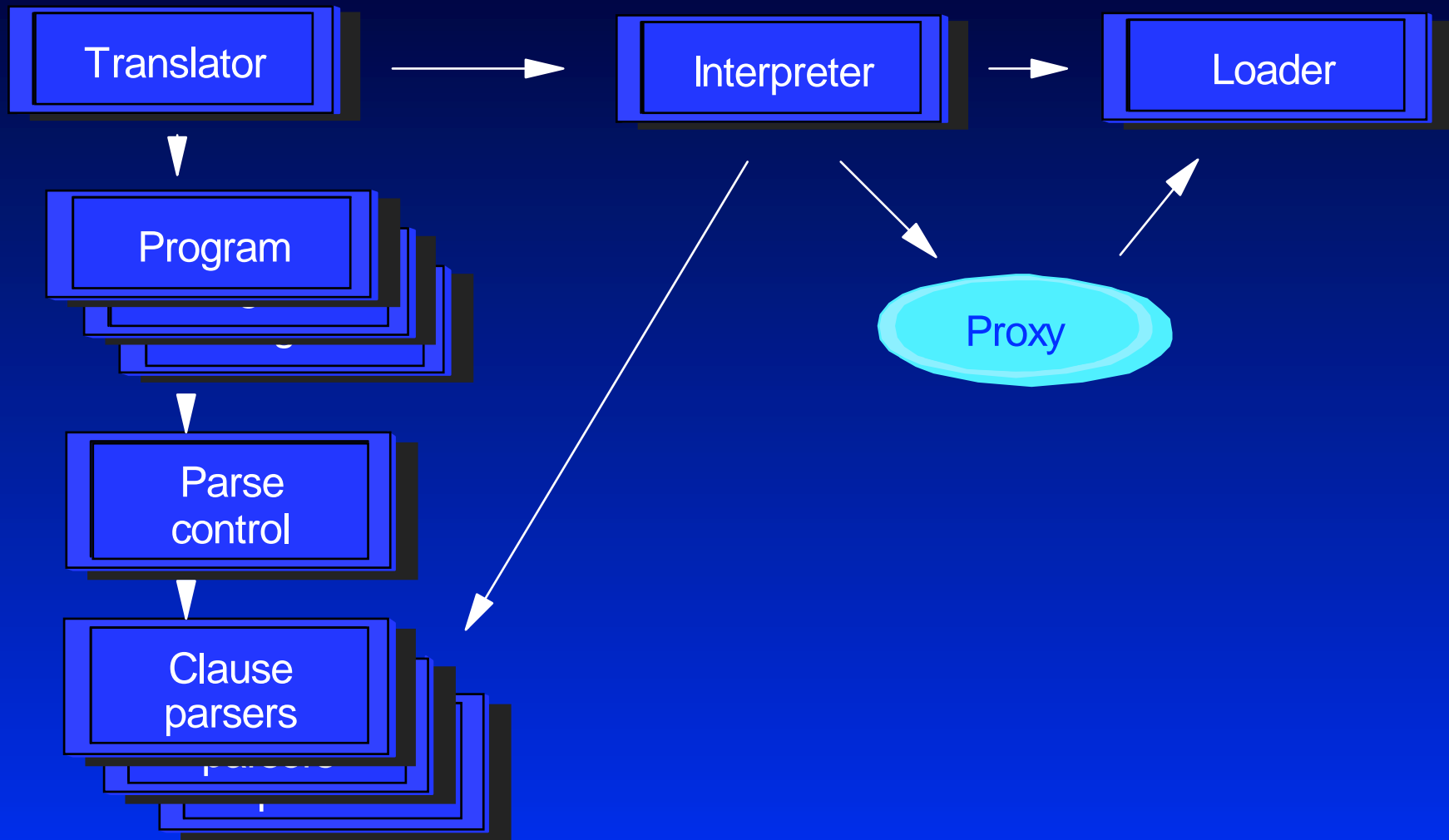
Interpreter

- Primary task is interpreting method bodies, by finding each clause in turn and invoking its **interpret** method
- When class first used or instance constructed, interprets initialization code (properties, *etc.*)
- Handles Java reflection (access to real properties, instances of objects, arrays, *etc.*)

Interpreter complications

- Signals - have to be wrapped, and cannot be passed through a reflection call
- Constructors - arguments to `super(x, y)` call must be interpreted, then the `super(x, y)` call must be made by the proxy class, and only then can the constructor method body be interpreted
- Protected (synchronized) blocks of code must truly be protected to be thread-safe

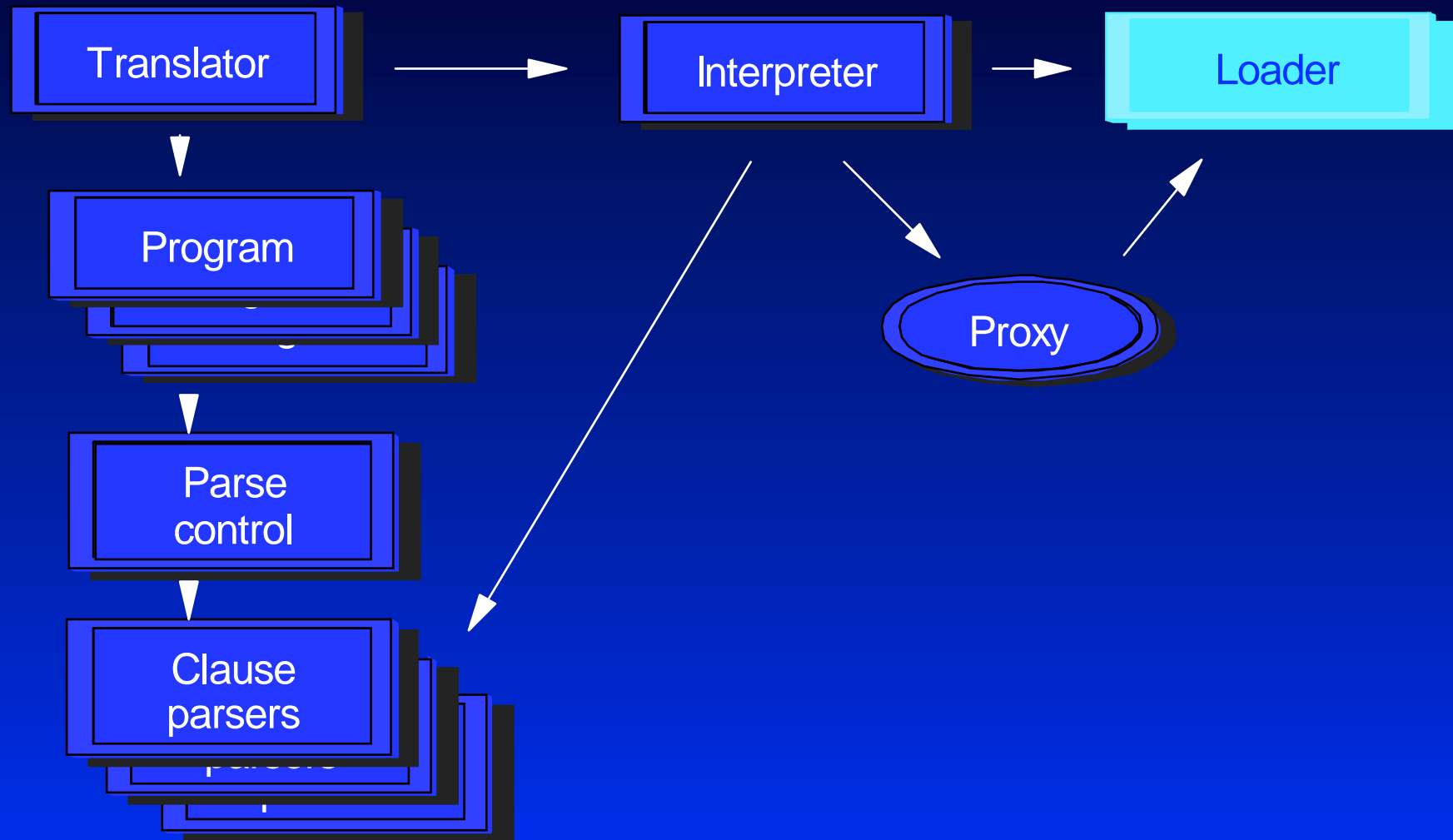
Interpretation



Proxy class

- Builds a binary class image (in a byte array) for a class that is to be interpreted
- Tedious but relatively straightforward - the code for every method is essentially the same
 - collect arguments (wrapped if necessary) into an Object array
 - invoke the interpreter to interpret the method body
 - get the returned Object; unwrap or cast it as required, and return it

Interpretation



Proxy class Loader

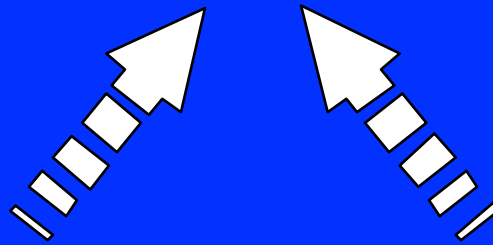
- A Java classloader is needed to actually load a class into the JVM
- If the built-in one were used then a class could never be redefined; classes are only unloaded when the object that loaded them is unloaded
- Complication: we also have to load any external (compiled) private classes, as otherwise they appear to be in a different package and hence would not be accessible when they should be

Summary

- A blend of Rexx and Java
 - scripting **and** application development
 - a truly general-purpose language
- Both decimal and binary arithmetic
- High productivity and simplicity
 - Java source is typically 35% bigger
 - Interpreter greatly speeds development
- Designed for **users**, not compilers.

<http://www2.hursley.ibm.com/netrexx/>

NetRexx



Rexx + Java

Strong typing doesn't need extra typing