Blue

Language Specification

Blue Version 1.0

Manual Revision 1.1

Michael Kölling
John Rosenberg

24.7.98

Monash University, Department of Computer Science and Software Engineering, Technical Report TR97-13

1. INTRODUCTION 1

1.1 WHAT IS BLUE? 1
1.2 THE BLUE ENVIRONMENT 1
1.3 ABOUT THIS DOCUMENT 1
1.4 BLUE PROGRAMMING 2

2. ALIASES 2

3. CLASSES AND THEIR OPERATIONS 3

3.1 MANIFEST VS. DYNAMIC CLASSES 4
3.2 GENERAL OPERATIONS 4
3.2.1 Assignment (:=) 4
3.2.2 Comparison (=, <>) 5
3.3 PREDEFINED TYPES 5
3.3.1 Integer 6
3.3.2 Real 6
3.3.3 Boolean 7
3.3.4 String 8
3.3.5 Array 10
3.4 USER DEFINED TYPES 11
3.4.1 General classes 11
3.4.2 Enumeration classes 13
3.5 OPERATOR PRECEDENCE 14

4. IDENTIFIERS 15

5. THE USES CLAUSE 15

6. VARIABLES 16

7. CONSTANTS 19

7.1 LITERALS 19
7.2 NAMED CONSTANTS 19
7.3 SET CONSTANTS 20

8. ROUTINES AND PARAMETERS 21

8.1 BUILTIN ROUTINES 23

9. STATEMENTS 24

9.1 ASSIGNMENT 24
9.2 ASSIGNMENT ATTEMPT 25
9.3 PROCEDURE CALL 26
9.4 RETURN 26
9.5 ASSERTION 27

10. EXPRESSIONS 27

10.1 FUNCTION CALL 28
10.2 EQUALITY 28
10.3 TYPE EQUALITY 29
10.4 IN 29
10.5 CREATE 30
10.6 THIS 30

11. CONTROL STRUCTURES 31

11.1 CONDITIONAL: THE IF STATEMENT 31
11.2 SELECTION: THE CASE STATEMENT 32
11.3 ITERATION: THE LOOP STATEMENT 33

12. PRE/POST CONDITIONS 34

13. CLASS INVARIANTS 36

14. COMMENTS 36

14.1 INTERFACE COMMENTS 37
14.2 IMPLEMENTATION COMMENTS 38

15. I/O 39

15.1 STANDARD I/O 39
15.1.1 *Standard Output 39*
15.1.2 *Output Formatting 40*
15.1.3 *Standard Input 41*
15.1.4 *The TextTerminal Class 43*
15.2 FILE I/O 43

16. INHERITANCE 48

16.1 DEFINING SUBCLASSES 48
16.2 REDEFINITION 49
16.3 CALLING SUPERCLASS FUNCTIONS 49
16.4 DEFERRED ROUTINES 51

17. GENERICITY 51

17.1 UNCONSTRAINED GENERICITY 52
17.1.1 *Classes and Types 53*
17.1.2 *Operations on Formal Generic Types 54*
17.2 CONSTRAINED GENERICITY 55

18. CONCEPTS NOT INCLUDED IN BLUE 55

18.1 MULTIPLE INHERITANCE 55
18.2 ROUTINE PARAMETERS 55
18.3 USER DEFINED INFIX OPERATORS 56
18.4 FUNCTION OVERLOADING 56
18.5 UNION TYPE 56
18.6 EXPLICIT BLOCKS 56

APPENDIX A: EBNF 57

APPENDIX B: COMPLETE LIST OF ALIASES 63

APPENDIX C: IMPLEMENTATION-DEPENDENT DEFINITIONS 64

APPENDIX D: INTERFACES OF PREDEFINED CLASSES 65

D.1 INTEGER 65
D.2 REAL 66
D.3 BOOLEAN 69
D.4 STRING 70
D.5 ARRAY 73
D.6 ENUMERATION 74
D.7 TEXTTERMINAL 74
D.8 OUTPUTFORMAT 77

INDEX 79

1. Introduction

1.1 What is Blue?

Blue is an object-oriented programming language especially designed for teaching. Its emphasis is on conceptual clarity and consistency, suitability for first year teaching and support for "good" software design.

1.2 The Blue Environment

Blue is intended to be used in an integrated programming environment. The environment itself will not be described in this document, but is important for the overall character of Blue programming. In order to obtain a complete picture of programming in Blue, the reader is encouraged also to read "The Blue Programming Environment".

1.3 About this Document

This document describes the Blue language. The Blue programming language is part of the Blue programming system. The language and some of the most important standard libraries (such as those for standard I/O) will be described in this document. However, most standard libraries will not be included here, but will be described in a separate document "The Blue Libraries".

It is assumed that the reader is familiar with a Pascal- or C-like language and with object-oriented concepts. This document does not give an introduction to object-oriented programming in general. It is assumed that the most basic ideas of object-orientation are known.

At some points this document also includes thoughts and reasoning beyond pure description of the language. These comments typically include reasons for specific decisions made during language design. They may be skipped by readers interested only in the language, or read by someone interested in the design process itself.

> *Comments about reasons for specific design decisions are set in greyed boxes like this and may be skipped without losing information about the Blue language.*

Descriptions of language constructs always include a formal syntax description in EBNF. A complete syntax description in EBNF can be found in Appendix A.

1.4 Blue Programming

Blue does not include a concept of programs. All programming is performed using classes. All code is part of a class. The classical idea of a program is represented in Blue as a collection of classes (called a *project* in Blue). Running a program is done by creating an instance of a class, and calling interface routines of that class.

A simulation of a classical "program" (a single entry point executable) can easily be achieved by having one designated top-level object with one interface routine which in turn creates and calls all other objects involved in the execution of the program. (This is the standard execution model in most existing object-oriented languages.) It is, however, only a small subset of the possible ways to execute code in Blue.

In the Blue environment, *all* classes in the project may be used to interactively create objects of that class and *all* those objects can immediately be used to call their interface routines (that is: without the necessity of writing test program shells). This changes programming in at least two ways: it allows very flexible (incremental) testing and incremental software development. All low level classes may immediately be used and tested for correct behaviour. It also allows the construction of software with more than one entry point, avoiding common awkward dispatch mechanisms envoked by command line options.

Classes are types[1]. So all programming comes down to the definition of types. This is why type declarations are the first major part of this document: They specify the overall structure of Blue programs.

2. Aliases

All types in Blue are classes and all data are objects. This general rule simplifies the language design. There are, however, a number of data types for which it is convenient to use syntax other than the Blue object call to perform one of their operations. The reason for this can be:

- Another syntax is commonly used and is therefore more intuitive (e.g. *3 + 5* for integer addition, rather than *3.add (5)*).
- Another syntax simplifies use of elementary constructs which should be used by beginners before the underlying language concepts need to be understood, e.g.
  print ("result=", 42)
  instead of
  output.write ("result=".concat (42.toString))
- Another syntax is more convenient (usually because it is shorter, see above).

[1] It is not strictly true that classes and types are the same, but most of the time each class represents exactly one type. This is not true when it comes to generic classes (which do not represent a type, but rather a *type pattern*). The difference will be explained in detail in section 17. For most of this document, however, this simplification is not a problem and makes expressing things a bit easier.

For these reasons, several operations on the predefined classes are supported by special syntax. This special syntax is allowed in addition to the standard object-call syntax generally available for all classes, and is referred to as *aliases*.

The prime reason for the introduction of aliases is to make the reading and writing of simple programs performing elementary tasks easy. Aliases provide an easy, intuitive syntax for the most common operations and considerably increase the ease with which Blue can be used by beginners.

Aliases will be learnt as statements or expressions in their own right by beginners, making it unnecessary to understand all underlying concepts right from the start. The expert programmer or compiler implementor, however, will appreciate the unifying concept for all data types.

Note that aliases are a pure *syntactic* addition which does not add any functionality to the language. They do not affect the semantics or the theoretical language description of Blue (although they are part of the language), and are purely intended to increase readability and intuitivity of statements.

Some common aliases and their resolutions are:

alias	*resolution*
3+6	3.add (6)
b1 or b2	b1.or (b2)
a[i]	a.getElem (i)
str (num)	num.toString
str (a, b, ...)	a.toString.concat (b.toString.concat (...))
print (a, b)	output.write (str (a.b))

The list of existing aliases is short and fixed – programmers cannot define additional aliases.

Where aliases exist for a construct explained in this document, they will be mentioned when that construct is introduced. A full list of aliases is given in Appendix B.

3. Classes and their Operations

All types derive from classes. Classes for the most commonly used data types are predefined in the language. The language offers customised syntax for these classes.

The predefined classes are:

- Integer
- Real
- Boolean
- String
- Array <*T*> (generic)

Two different kinds of classes exist: manifest classes and dynamic classes. The following sections will introduce manifest and dynamic classes, describe operations that are available on all classes, and then list the predefined classes and their operations. Finally, user defined classes are described.

3.1 Manifest vs. Dynamic Classes

Manifest classes are classes where all objects are known statically. The objects pre-exist with the definition of the class and do not have to be created. The manifest classes are Integer, Real, Boolean, String and Enumeration classes. The first four of those are predefined and all values are known to the Blue compiler. Enumeration classes are user defined. The definition of such a class consists of an enumeration of all existing objects of that class, simultaneously creating a named reference to each object.

The literal '2', for instance, is a reference to the unique integer object with the value *2*. The code segment

```
a := 2
b := 2
```

assigns references *to the same object* to a and b. Only one integer object with the value *2* exists. This does not create two distinct objects.

All literals are constant references to objects of manifest classes.

Dynamic classes are classes where, rather than listing all objects, a creation method for objects is specified. Dynamic classes are arrays and user defined general classes.

With the definition of a dynamic class, no object is created automatically. The user has to execute an explicit *create* operation to create objects of these types.

Care must be taken not to confuse this with pointer and non-pointer types in other languages. In Blue, **all variables hold references to objects**. An integer variable holds, when assigned a value, the *reference* to that integer object. Thus the object model is simple: only references to objects exist. The difference between manifest and dynamic objects affects only the time and method of creation of the objects of the class, not the mechanism by which they are referenced.

3.2 General Operations

This section describes operation that are common to all classes.

The general operations are:

:=	assignment
=	comparison (equality)
<>	comparison (inequality)

3.2.1 Assignment (:=)

The *assignment* (:=) assigns an object reference to a variable (variables *always* store object references). The object itself is not copied.

Example:

Consider variables a and b which are declared to be of some user defined class:

```
var
    a: Myclass
    b: Myclass
```

Then the code segment

```
    a := create Myclass
    b := a
```

causes *a* and *b* to reference the *same* object. Changes to *a* will be visible using *b*.

3.2.2 Comparison (=, <>)

The *comparison* compares two values (variables or expression results). Values are always object references.

Example:

```
var
    a: myclass
    b: myclass
    result: Boolean

    …
    result := b = a
```

result will be true if *a* and *b* reference the same object. It will *not* be true, if *a* and *b* reference different objects in identical states.

a <> b is true, if *a* and *b* do not reference the same object.

3.3 Predefined Types

The predefined types and their operations are as follows. Many of the operations mentioned here are aliases. The full list of operations and their aliases are given in the individual sections for each type below.

Integer	**Real**	**Boolean**	**String**	**Array**
n + *m*	*x* + *y*	not *b*	s.length	a [*n*]
n - *m*	*x* - *y*	*a* and *b*	str (s1, s2)	a.size
-*n*	-*x*	*a* or *b*	s.substring (*n*,*m*)	a.setSize (*n*)
n * *m*	*x* * *y*		s[*n*]	a.init (*v*)
n div *m*	*x* / *y*		s1.find (s2,*n*)	
n mod *m*				
n ^ *m*	*x* ^ *y*			
	sqrt(*x*)			
n > *m*	*x* > *y*		s1 < s2	
n < *m*	*x* < *y*		s1 > s2	
n >= *m*	*x* >= *y*		s1 <= s2	
n <= *m*	*x* <= *y*		s1 >= s2	
str (*n*)	str (*x*)	str (*b*)		

Integer, *Real*, *Boolean*, and *String* are manifest classes. *Array* is a dynamic class.

The predefined types are now discussed in more detail. For a full class interface of the predefined types see Appendix D.

3.3.1 Integer

The class Integer stores whole numbers. Integer is a manifest class.

The operations defined on integers are:

alias	*operation*	*return type*	*meaning*
n + *m*	*n*.add (*m*)	Integer	addition
n - *m*	*n*.sub (*m*)	Integer	subtraction
-*n*	*n*.neg	Integer	negation
n * *m*	*n*.mult (*m*)	Integer	multiplication
n div *m*	*n*.div (*m*)	Integer	Integer division
n mod *m*	*n*.mod (*m*)	Integer	remainder
n ^ *m*	*n*.pow (*m*)	Integer	power
n > *m*	*n*.greater (*m*)	Boolean	greater than
n < *m*	*n*.less (*m*)	Boolean	less than
n >= *m*	*n*.greaterEq (*m*)	Boolean	greater or equal
n <= *m*	*n*.lessEq (*m*)	Boolean	less or equal
str (*n*)	*n*.toString	String	conversion to String

In addition to these operations, integer expressions can always be used where real expressions are expected. An implicit conversion to Real takes place in that case.

Integer literals are written as usual:

```
42
-99
```

Constants are predefined for the smallest and largest available integer value in every specific implementation:

MAXINT	largest representable integer
MININT	smallest representable integer

The actual value of MAXINT and MININT are implementation dependent.

3.3.2 Real

The class Real stores floating point numbers. Real is a manifest class.

The operations defined on reals are:

alias	*operation*	*return type*	*meaning*
x + *y*	*x*.add (*y*)	Real	addition
x - *y*	*x*.sub (*y*)	Real	subtraction
-*x*	*x*.neg	Real	negation
x * *y*	*x*.mult (*y*)	Real	multiplication
x / *y*	*x*.div (*y*)	Real	division
x ^ *y*	*x*.pow (*y*)	Real	power
	x.sqrt	Real	square root
	x.trunc	Integer	truncation to integer
	x.round	Integer	round to closest Integer
x > *y*	*x*.greater (*y*)	Boolean	greater than
x < *y*	*x*.less (*y*)	Boolean	less than
x >= *y*	*x*.greaterEq (*y*)	Boolean	greater or equal
x <= *y*	*x*.lessEq (*y*)	Boolean	less or equal
str (*x*)	*x*.toString	String	conversion to String

Real literals are written as:

```
2.0
0.111
-0.001
```

Note that an implicit conversion takes place when an integer is provided where a real value is expected. Thus it is legal to write 2 instead of 2.0, where the value of 2 will be implicitly converted to the real value 2.0.

3.3.3 Boolean

Variables of class Boolean store boolean values (i.e. *true* or *false*). Boolean is a manifest class.

The operations defined on booleans are:

alias	*operation*	*return type*	*meaning*
not *b*	*b*.invert	Boolean	negation
a and *b*	*a*.and (*b*)	Boolean	logical and
a or *b*	*a*.or (*b*)	Boolean	logical or
str (*b*)	*b*.toString	String	conversion to String

The boolean literals are:

```
true
false
```

Note that, since "true" and "false" are keywords, and the case of keywords is insignificant, variation of case such as True or TRUE are also recognised.

Partial Evaluation

Boolean expressions are incrementally evaluated. Evaluation starts from the left (subject to operator precedence rules) and stops as soon as the result can be determined.

Examples:

Consider the expression

```
a and b
```

If *a* evaluates to *false*, the evaluation of the expression will terminate and return the result *false*. *b* is only evaluated if *a* evaluates to *true*.

Likewise, in the expression

a or b

b is only evaluated if *a* is *false*. If *a* is true, the result of the expression will be determined without evaluating *b*.

This allows the following code segments to be written:

```
if obj<>nil and obj.size>N then ...
```

or

```
if n<=a.size and a[n]>0 then ...
```

3.3.4 String

Strings hold sequences of characters. Strings are manifest. Thus a String variable can reference any possible String. There is no notion of a specific (maximum) number of characters or a specific memory section associated with a String variable. String literals are written using double quotes (").

Example:

```
s1, s2: String
...
s1 := "Da steh' ich nun, ich armer Tor"
s2 := s1
```

The effect of class String being a manifest type is that it behaves like having value semantics. In this example, subsequent changes to *s1* will not affect *s2*. (A change of *s1* only makes *s1* reference another string.)

Characters and substrings can be extracted by using indices. String indices always start at 1.

There is no character type in Blue. Characters are strings with length one.

The most common operations on strings are:

alias	*operation*	*return type*	*meaning*
	s.length	Integer	number of characters in s
str (s1, s2)	s1.concat(s2)	String	returns concatenation of s1 and s2
	s.substring (*n*,*m*)	String	substring of s starting at *n* with length *m*
s[*n*]	s.substring (*n*,1)	String	character in s at position *n*
	s2.find (s1, *n*)	Integer	position of s1 in s2 after *n*
	s1.insert (s2, *n*)	String	inserts s2 into s1 at pos. *n*
	s.delete (*n*, *m*)	String	delete *m* characters, starting at *n*
s1 < s2	s1.less (s2)	Boolean	s1 less than s2 ?
s1 > s2	s1.greater (s2)	Boolean	s1 greater than s2 ?
s1 <= s2	s1.lessEq (s2)	Boolean	s1 less or equal s2 ?
s1 >= s2	s1.greaterEq (s2)	Boolean	s1 greater or equal s2 ?
str (s)	s.toString	String	identity
	s.toUpper	String	convert to upper case
	s.toLower	String	convert to lower case
	s.caseEqual (s2)	Boolean	equality (ignore case)

Special characters

A number of escape sequences are provided to specify non-printable characters:

"\t"	tab character
"\n"	end-of-line character
"\\"	"\"
"*nnn*"	ISO character *nnn* (decimal)

Long strings

If a string is too long to be written in one line in the editor, it can be broken into sevaral parts:

```
s :=  "This is one string"
      " (and this is still the same string)"
```

String literals that are separated by whitespace only are processed as one string. The neighbouring quotes and the whitespace in between are ignored. The strings

```
"one "  "two "  "three"
```

and

```
"one "
"two "
"three"
```

and

```
"one two three"
```

are identical. Every string literal (every part of the string) must end on the same line it starts on.

For a full interface of class String, see Appendix D.

The decision to make the class String a manifest class may surprise initially. Programmers tend to think about strings as memory buffers holding characters. Objects of manifest classes can not change - this contradicts the model programmers used to some other languages have of strings: they can change. After careful examination, however, it turns out that we usually think of strings not as different objects. Making strings a dynamic class would lead to the effect that in the code fragment

a := "marvin"

if a = "marvin" then ...

the expression a = "marvin" *results in* false *(because a literal string is then an object creator and* a *and* "marvin" *are two distinct objects). Intuitively, we do not think about strings as independent objects, where two distinct objects with the same value can exist. If they have the same value, then they* are *the same. And this is exactly the behaviour of manifest objects.*

All string functions that change a string (such as toUpper*) do not change the current string, but rather return a reference to another string that contains the required text.*

3.3.5 Array

Arrays are the only predefined dynamic class. Therefore array objects need to be explicitly created.

Arrays are sequences of homogenous elements. The class specification is generic. (For details about generic classes, see section 17.) The element type is specified when declaring a variable; the index type is always Integer. The size of an array is dynamic. The boundaries are not statically fixed and are not part of the type.

Example:

```
var a: Array <Boolean>
...
a := create Array<Boolean> (20)
```

This example creates an array of 20 booleans. Array indices always start at 1, so this array contains elements from a[1] to a[20].

Arrays can then be assigned like all dynamic classes.

Operation on arrays:

alias	*operation*	*return type*	*meaning*
	create Array<*t*> (*n*)	Array<*t*>	create a new array with *n* elements of type *t*
a [*n*]	a.getElem (*n*)	*t*	element at index *n*
a [*n*]	a.putElem (*n*, *t*)	–	set element at index *n*
	a.init (*v*)	–	set all elements in a to *v*
	a.size	Integer	number of elements in a
	a.setSize (*n*)	–	change array size to *n*. If n > old n, the new elements are *undefined*.

Apart from these operations, literal Array constructors can be specified.

Example:

```
var a: Array <Integer>
...
a := [23, 2, 42]
```

This code fragment creates an integer array of size three with a[1]=23, a[2]=2, a[3]=42 and assigns a reference to that array to *a*.

Elements of an array can be of any type. In particular, arrays of arrays are possible:

```
var
    matrix: Array <Array<Integer>>
    i1, i2: Integer
...
    i1 := matrix [2][2]
```

The array bracket alias is special in that it can appear on the right or the left side of an assignment. Dependent on its position, it translates to either *getElem* or *putElem*.

num := a [i] *translates to* num := a.getElem (i)
a [i] := num *translates to* a.putElem (i, num)

The bracket alias therefore is the only alias that is not a simple macro.

3.4 User Defined Types

User defined classes can be either *general classes* (user defined dynamic classes) or *enumeration classes* (user defined manifest classes).

3.4.1 General classes

The definition of a general class is the closest thing to a "program" in procedural languages. A class typically consists of some internal data (encapsulated), internal routines, a creation routine and interface routines. The interface cannot contain variables.

BNF:

```
class-decl          ::= class identifier [ "<" generics-list ">"] is
                            class-definition

generics-list       ::= gen-param { , gen-param }
gen-param           ::= identifier [ is identifier ]

class-definition    ::= [ identifier ] general-class-decl
                      | ...

general-class-decl  ::= class-comment
                        uses [ ident-list ]
                        [ internal
                            [ const const-decls ]
                            [ var var-decls ]
                            [ routines routine-decls ] ]
                        interface
                            [ creation [ "(" parameter-list ")" ]
                              routine-body ]
                            [ routines routine-decls ]
                        [ invariant condition ]
                        end class
```

Example:

```
class Rectangle is
    == Author:   M. Kölling
    == Date:     April 1995
    == Version:  1.0
    == Short:    Graphical representation of a rectangle.
    ==
    ==   Class Rectangle represents a rectangle with specified
    ==   coordinates and colour that can be ...

  uses Point, Colour

internal

  var
    top_left: Point
    bottom_right: Point
```

```
        fill_col: Colour
        border_col: Colour
    interface
        creation (tl: Point, br: Point) is
            == Create rectangle at coordinates defined by tl (top-left)
            == and br (bottom-right).  Default colours are: fill white,
            == border black.
        do
            top_left := tl
            bottom_right := br
            fill_col := white
            border_col := black
        end creation

        routines
            move (dx: Integer, dy: Integer) is
                == Move rectangle by distance defined by dx, dy.
            do
                top_left.move (dx, dy)
                bottom_right.move (dx, dy)
            post
                size = old size
            end move

            size -> (width: Integer, height: Integer) is
                == Return size of rectangle in width and height.
            do
                width := bottom_right.x - top_left.x
                height := bottom_right.y - top_left.y
            end size
    invariant
        (top_left.x < bottom_right.x) and
        (top_left.y < bottom_right.y)
    end class
```

There is no separate interface definition for a class. The interface of a class is a restricted view of the class attributes (the internals are hidden) and it is presented to the user as a special view that is produced by a tool in the Blue Programming Environment from the full class definition.

For details on generic classes, see section 17.

- *The name is always first in all declarations. The reason for this is that a class definition is mostly used to look up interface features (members) and their characteristics (parameters, types). The feature should be easy to find. This is achieved by placing the name first, making it easy to read down a column of identifiers. In C-type syntax (type and some keywords first) the identifier is shifted somewhere towards the middle of the line, which makes it harder to find. The name is followed by the access*

definition (type, parameters) showing the syntax for usage, and the implementation definition (routine body), which is class internal information.

- *Creation is separate to make clear it is not a normal routine. Create looks similar to a routine but has a special syntax and semantics. It is separated from normal interface procedures and put into a special position in the class declaration to make it easy to find by enforcing a certain position for it and to symbolise its special meaning.*
- *Comments are compulsory. Writing comments is not seen as a luxury but as part of programming. Therefore the compiler also deals (as well as it can) with proper commenting. This includes a class comment, containing certain keywords, used by the library browser, and comments for routines.*
- *Eliminating separate interface files avoids the danger of inconsistency and repetition of code.*

3.4.2 Enumeration classes

Enumeration classes are the only user defined manifest classes. In a manifest class all objects exist automatically and no other objects of this class can be created. Every object is referenced by a named constant.

class-decl	::=	**class** ***identifier*** ["<" *generics-list* ">"] **is** *class-definition*
class-definition	::=	**Enumeration** *enum-class-decl* | ...
enum-class-decl	::=	*class-comment* **manifest** *ident-list* **end class**
ident-list	::=	*identifier* {"," *identifier* }

Example:

```
class Colour is Enumeration
    == Simple enumeration type for basic colours
    manifest  red, white, blue
end class
```

This example defines and creates three objects which are referenced through the constant identifiers *red*, *white* and *blue*. In another class a variable can be defined of class *Colour*, and one of the constant references can be assigned to that variable.

Example:

```
var col1, col2: Colour
...
```

```
col := red
col2 := blue
```

Note that, because no new objects are created at any time, after assigning *red* to both *col1* and *col2*, *(col1 = col2)* is true.

Enumeration classes inherit from the predefined class *Enumeration*. *Enumeration* defines two routines *pred* and *succ* and the standard routine *toString*, which can then be applied to all enumeration types. It also provides a routine *ord* for conversion to Integer. For a full interface definition of *Enumeration*, see appendix D.

No further attributes or routines can be defined for enumeration objects. The only characteristics of enumerations are: they exist, they are distinct, and they are ordered.

Qualified Enumeration Constants

In some situation it may be necessary to qualify enumeration values with their type. This situation arises when a class uses two enumeration types that define one or more common enumeration constants. Consider the following two enumeration classes:

```
class Colour is Enumeration
    == Simple enumeration type for basic colours
    manifest  red, white, blue
end class
class Program is Enumeration
    == Simple enumeration type for basic colours
    manifest  emacs, vi, blue
end class
```

If a third class now uses both *Colour* and *Program*, the constant *blue* is ambiguous. Trying to use it will result in a compile time error. This conflict can be resolved by preceding the enumeration value with its class name and an exclamation mark (!):

Example:

```
var    col : Colour
       prog : Program
…
    col := Colour!blue
    prog := Program!blue
```

3.5 Operator Precedence

(to be written)

4. Identifiers

Identifiers are used as names for classes, variables, constants and routines. Identifiers are strings that consist only of letters, digits and the underscore (_), where the first character is a letter or an underscore.

Example of legal identifiers are:

```
abc
Num23
addAtEnd
add_at_beginning
_count
```

Examples of illegal identifiers are

```
hit cnt          -- error: space in identifier
23add            -- error: digit at start of identifier
#elem      -- error: illegal symbol (#) in identifier
```

Case of Characters

Blue identifiers are case sensitive. That means that *abc* and *ABC* are two different identifiers.

By convention, class names are often written with a leading capital, whereas routine names start with a lowercase character.

5. The Uses Clause

If a class A declares variables or parameters of another class B (we then say *"A uses B"*), and B is not one of the predefined classes, then A must declare the use of B.

BNF:

```
class-decl          ::= class identifier [ "<" generics-list ">"] is
                            class-definition

class-definition    ::= [ identifier ] general-class-decl
                        | ...

general-class-decl  ::= class-comment
                        uses [ ident-list ]
                        ...
                        end class
```

The following example shows a class *Rectangle* that uses two classes *Point* and *Colour*.

Example:

```
class Rectangle is
      == ...
   uses Point, Colour
   ...
end class
```

The effect of the *uses* clause is that the classes *Point* and *Colour* are known inside the class definition of *Rectangle* and can then be used as types for variables, parameters and return values. The classes *Point* and *Colour* must exist in the current project.

Uses clauses are inherited. If a superclass lists a class in its uses clause, the subclass does not need to (and indeed must not) repeat the same class in its uses clause. A superclass itself is automatically known to the subclass – it does not need to be listed as a used class. For more details about inheritance see section 16.

The *uses* keyword itself is not optional. If a class does not use other classes the uses list is empty, but the keyword must appear in the source.

The predefined classes (Integer, Real, Boolean, String, Array) do not need to be listed in the uses clause. They are automatically known in every class without being explicitly mentioned as being used. This is, in fact, the meaning of the term *predefined* – those classes are always considered used by every class.

6. Variables

Variables can be declared of any class. This is done in separate variable declaration sections. Such a section exists for the whole class (instance variables) and once for each routine (local variables) and is preceded by the keyword *var*. The scope for instance variables is the class definition, the scope for local variables is the routine they are declared in.

The section declaring the instance variables must be in the *internal* section and must precede the internal routine declarations. No variables are allowed in the class interface.

The section declaring local variables follows the preconditions and constant declarations in a routine declaration and precedes the routine statement block.

BNF:

var-decls	::=	*var-decl* { *var-decl* }
var-decl	::=	*ident-list* "**:**" *type-def* [*initialisation*]
type-def	::=	*class-ident* ["<" *ident-list* ">"]
class-ident	::=	***identifier***
initialisation	::=	"**:=**" *expression*

Examples:

```
var
    num1, num2: Integer
    count : Integer := 0
    int_arr: Array <Integer>
    f: Figure
    s: Stack <Figure>

printList (l: List) is
        == print all list elements

    var
        cnt: Integer
        nm: String := ""

    do
        ...
    end printList
```

Lifetime

The lifetime of local variables is the time of a routine execution. In other words: scope and lifetime are connected (as in Pascal). There is no mechanism to extend the lifetime of local variables beyond the procedure execution (such as the "static" construct for local variables in C).

The lifetime of instance variables is the time of existence of the object they are part of.

States_and_Values

Variables are a combination of type, state and value. The type of a variable is specified statically and never changes. At any time, each variable is in one of two states: *undefined* or *defined*. If a variable is *defined*, it also has a value. The value of a variable is always a reference to an object of its type (or *nil*, see below).

A variable can be initialised on declaration by any legal expression, using the assignment instruction (see example above). If a variable is not initialised at its declaration, its initial state is *undefined*. It is an error to use an undefined variable. An attempt to do so results in a runtime error.

No function exists to check for the state *undefined*. It is the responsibility of the programmer at the time of writing the program to ensure that variables are not used in the undefined state. (A good programmer will always know when a variable is potentially undefined. Parameters can never be undefined – that would have resulted in a runtime error at the time of the call.)

Assignment of a value to a variable changes the state to *defined*. After a variable leaves the state *undefined*, it never returns to that state.

Nil

A special value called *nil* exists. The value *nil* indicates that the variable does not reference any object. A variable of any type can hold the value *nil*. *nil* can be assigned, passed as a parameter and used in comparisons. Trying to use the object held by a nil variable (e.g. in an object call) is an error, since the variable does not refer to an object.

Encapsulation

Variables are encapsulated and cannot appear in the interface of a class. This means that a class cannot directly access a variable of another class. (It can indirectly access it if that class offers a function that returns or changes the value of a variable, but this is the choice of the class owning the variable.)

The only variables directly accessible are local variables of the current routine and instance variables of the current class.

Note, however, that encapsulation is *class based*, not *object based*. As a result, a class *can* access variables in another object, if that object is of the same class.

Example:

```
class Point is
    …
    var
        x, y: Integer
    …
    add (other : Point) is
            == Add another point to this one
        do
            x := other.x
            y := other.y
        end add
```

This technique is useful in situations where access to the internals is needed, and no interface access functions are defined. (An object clone function is such a case where an object might want to create a new object of its own class and set all its instance variables to the same values as itself. We would not want to make all internal variables publicly accessible only to be able to provide a clone function.)

The possibility of accessing instance variables of another object, even though they are of the same class, seems to some people to go against the information hiding (or encapsulation) principle. And in some respects it does. To understand the reason for this access to be allowed, we must understand why information hiding is a good idea in the first place.

Information hiding is a software engineering technique, that decouples the implementation of different modules (here: classes) from each other. Accessing variables directly in arbitrary places is a maintenance dϑisaster, because changes to the implementation of one module might require changes in unexpected other places in the program. This is why we do not allow direct access of instance variables.

If the access comes from the same class, though, (even if it is from another object) the argument does not hold anymore. If I change the implementation of that class, I will change all accesses to a variable within that class. So allowing access to instance variables of other objects from within the same class is no problem.

7. Constants

Constants in Blue can be literals or named constants. Literals are available for the predefined manifest classes. Named constants can be created from every legal expression.

7.1 Literals

Literals can be written for all predefined types. They have been introduced in the sections describing the individual types. Here, we just provide a brief summary of possible literal values for the predefined types. For details, refer to the section introducing each type (sections 3.3.1 to 3.3.5).

Literals:

Class:	Integer	Real	Boolean	String
Examples:	17	2.0	true	"hello"
	0	0.0	false	""
	-5	-0.122	TRUE	"ab\n"

7.2 Named Constants

Named constants are defined in a separate constant definition section, which is preceded by the reserved word *const*. The constant definition section can precede every variable section. Scope rules are the same as those for variables, and the syntax of the declaration is the same as for variables, except that the initialisation is not optional.

BNF:

const-decls	::=	*const-decl* { *const-decl* }
const-decl	::=	*ident-list* "**:**" *type-def initialisation*
initialisation	::=	"**:=**" *expression*

Examples:

```
const
    size: Integer := 99
    default_title : String := "untitled"

printList (l: List) is
        == print all list elements
    const
        empty : Boolean := l.isEmpty (false)
    var
        ...
    do
        ...
    end printList
```

Note that literals as well as arbitrary function calls may be used for the initialisation of constants. Instance variables cannot be used for the initialisation of instance constants — they are still uninitialised. Care must be taken with function calls: If the function used for initialisation uses instance variables, a runtime error will be generated for using an uninitialised variable. The initialisation of instance constants and variables takes place on object creation *before* the creation routine is executed.

The exact order of object initialisation is:

- Initialisation of instance constants
- Initialisation of instance variables
- Execution of the creation routine, including (in this order)
  - Initialisation of local constants
  - Initialisation of local variables
  - Execution of routine body

All constants and variables are initialised sequentially in the order in which they are defined. Later definitions may use preceding ones.

7.3 Set Constants

Blue allows sets of literals to be specified. These sets can only appear as literals. No variables of these set types can be created. The only available operator on these sets is *in*, testing a value for membership in a given set. The *in*-operation returns a boolean value. (See section 10.4 for more details on the *in* operation.) Constant sets are also used for *case* statements (section 11.2).

BNF:

set_expr	::=	"**{**" [*set_elem* { "**,**" *set_elem* }] "**}**"
set_elem	::=	*expression* | *subrange*
subrange	::=	*expression* "**..**" *expression*

Examples:

```
if choice in {1, 2, 3 } then ...

if result in {0..9} then ...

whitespace := ch in {" ", "\n", "\t"}

if ch in {"a".."z", "A".."Z", "_"} then ...
```

As the examples show, sets can be created by enumeration of values by comma separated lists, subranges, or a combination of both.

Subranges can only be taken from ordinal types. The ordinal types are Integer, String and Enumeration.

This construct does not replace a general purpose set type. It is merely a convenience notation that abbreviates relational comparisons. A general purpose set class can be found in the Blue Standard Collection Library.

8. Routines and Parameters

There are two types of routines: procedures and functions. Both have an optional parameter list. Functions return one or more values.

All parameters are passed by value. (Note that variables always hold references, so a *reference* is passed by value, resulting in semantics of pass by reference for the objects themselves.)

Parameters cannot be used to return values directly. (There are no "var" parameters.) The only way to return a value is via a function result. Thus procedures do not return any values (but they can change the state of an object they reference). A function can return more than one result.

BNF:

routine-decl	::=	***identifier*** ["**(**" *parameter-list* "**)**"] ["**->**" "**(**" *parameter-list* "**)**"] **is** *routine-impl*
parameter-list	::=	*param-decl* {"**,**" *param-decl* }
param-decl	::=	***identifier*** "**:**" *type-def*
routine-impl	::=	**deferred** *routine-spec* | **builtin** *routine-spec* | [**redefined**] *routine-body*
routine-body	::=	*routine-comment* [**pre** *condition*] [**const** *const-decls*] [**var** *var-decls*] **do** *statement-list* [**post** *condition*] **end** ***identifier***
routine-spec	::=	*routine-comment* [**pre** *condition*] [**post** *condition*] **end** ***identifier***

Examples:

A short function is

```
count –> (val: Integer) is
        == Return number of elements
    do
        val := i_count
    end count
```

A simple procedure without parameters, pre or post conditions:

```
show is
    == displays this element on standard output
  do
    print ("Name: ", name, "\n")
    print ("Count: ", count, "\n")
  end show
```

A function with two parameters and two return values:

```
lookup (n: Integer, show: Boolean)
                    -> (nm: String, cnt: String) is
    == returns name and count for entry number n
    ==   n: number of entry to lookup, must be valid
    ==   show: if true, value is also printed to standard out
    ==   nm: returns the name of the entry
    ==   cnt: returns the number of the entry
  pre
    n >= 1
    and n <= nr_of_entries
  do
    nm := entries[n].name
    cnt := entries[n].count
    if show then
        entries[n].show
    end if
  post
    nr_of_entries = old nr_of_entries
  end lookup
```

The comment after the function header is part of the language. Comments for routines are not optional and are part of the routine interface.

Pre- and post conditions are optional.

Values are returned by assigning to the result variable (*nm* and *cnt* in this example). Result variables are initially undefined and must be assigned a value during function execution. It is an error to return from a function with undefined result variables. A function without an assignment statement to a result variable will produce a compile time error; a function that contains but does not execute such a statement results in a runtime error.

Visibility

Routines are visible in the whole class scope. This means that a routine call may precede the routine definition, allowing mutually recursive routines.

Many people hold the view that functions should not have side effects. Suggestions have been made to us that the language should not allow side effects in functions (by inhibiting assignments to instance variables and routine calls to objects held in instance variables). While this is technically possible, it is impractical. We agree that it is a valuable design rule that functions should not have side effects, but side effects in this context can only mean changes to the external state *of the object.*

Sometimes side effects that change the internal state *of the object are desirable. An example is the implementation of a query object with caching. A query (a function) should not change the* external state *of the object, but it may well result in some information being cached inside the object (thus changing the state of the object – a side effect). If this change does not affect the externally visible behaviour (other than in efficiency), we say that the* external state *has not been changed. Our design rule is thus satisfied. The compiler cannot distinguish whether changes of state are internal or external. Thus this valuable design rule remains unenforced by the compiler.*

For details on pre- and postconditions, see section 12.

For details on redefined routines, see section 16.2.

For details on deferred routines, see section 16.4.

8.1 Builtin Routines

Builtin routines are written with the keyword *builtin* after the routine header:

BNF:

routine-decl	::=	**identifier** ["**(**" *parameter-list* "**)**"] ["->" "**(**" *parameter-list* "**)**"] **is** *routine-impl*
routine-impl	::=	**builtin** *routine-spec* | ...
routine-spec	::=	*routine-comment* [**pre** *condition*] [**post** *condition*] **end *identifier***

Example:

```
cursorTo (x: Integer, x: Integer) is builtin
    == Set the terminals cursor to screen position (x,y).
  pre
    x >= 0 and x <= width  and
    y >= 0 and y <= height
  end cursorTo
```

The builtin keyword is used only by system programmers writing classes that are part of the Blue environment. It cannot be used by application programmers. Builtin classes are usually implemented in a system-specific way (often in another language than Blue).

9. Statements

Statements are:

- assignment
- assignment attempt
- procedure call
- return
- assertion
- control structures

Control structures are described in section 11. This section only describes simple (non-compound) statements.

BNF:

statement	::=	*assignment* | *assignment-attempt* | *procedure-call* | *return* | *assertion* | ...

9.1 Assignment

Values can be assigned to variables if the type of the value *conforms to* the type of the variable (see below for definition of *conformance*).

BNF:

assignment	::=	*ident-list* "**:=**" *expression-list*
ident-list	::=	***identifier*** {"**,**" ***identifier*** }
expression-list	::=	*expression* {"**,**" *expression* }

Examples:

```
a := 42
a, b, c := x, y, z        – – multi-assignment (see below)
a, b, c := f()
```

Multi-Assignment

The multi-assignment is defined as follows: Each of the expressions on the right hand side is evaluated to one or more values. (An identifier always evaluates to one value, a function evaluates to the list of its return values.) The order of evaluation is undefined. The values are then assigned to the identifiers on the left hand side. The number of identifiers must be equal to the number of values produced by the expressions on the right hand side. The first value is assigned to the first identifier, the second value to the second identifier, and so on. The evaluation of all expressions on the right hand side

is always completed before any assignment is executed. This allows a statement like

```
a, b := b, a
```

to swap two values.

Conformance

A type *B* conforms to a type *A* if

- *B* is the same type as *A*
- *B* is a (direct or indirect) subclass of *A*

Thus, the code segment

```
var
   x: A
   y: B      – – B is subclass of A
...
y := create B
x := y
```

is legal, because B is a subclass of A, and so B *conforms to* A.

The variable *x* is said to have the *static type* A and the *dynamic type* B.

An assignment

```
a := b
```

is legal, if the static type of *b* conforms to the static type of *a*.

9.2 Assignment Attempt

The assignment attempt allows an assignment from a variable *a* of some class to a variable *b* of a subclass. Such an assignment can be safely executed without risking type errors only if the *dynamic type* of *a* conforms to the static type of *b*.This can not be statically determined and causes a dynamic check to be executed.

BNF:

assignment-attempt	::=	*ident-list* "**?=**" *expression-list*
ident-list	::=	*ident* {"," *ident* }
expression-list	::=	*expression* {"," *expression* }

Example:

```
var
   x: A
   y: B      – – B is subclass of A
...
y ?= x
```

Statically, this assignment cannot be guaranteed to be successful (since the type of *x* does not conform to the type of *y*). *x* could reference an object of the dynamic type *A*. In that case the assignment cannot be executed, because *y*

cannot hold references of type *A*. But *x* could also reference an object of the dynamic type *B*. In that case the assignment can be executed.

The assignment attempt results in a dynamic (runtime) check of the dynamic type of *x*. If the dynamic type of *x* conforms to the static type of *y*, the assignment will be executed, otherwise *y* will be assigned the value *nil*.

An assignment attempt

a ?= b

is statically legal if the static type of *b* conforms to the static type of *a*, or the static type of *a* conforms to the static type of *b* (in other words: if there is any chance that the assignment could be successful).

9.3 Procedure Call

A procedure call is a call to a routine that does not return parameters.

BNF:

```
procedure-call   ::=  call-chain
call-chain       ::=  [ super "!" ] unqualified-call { "." unqualified-call  }
unqualified-call ::=  identifier [ "(" expression-list  ")" ]
```

Examples:

```
proc
proc (a,b)
obj.proc (a, 23+17)
alist.get (22).put ("hello")
```

A procedure call can be a call of either an internal or external procedure. An internal procedure is a procedure that is defined in the same class containing the procedure call. An external procedure is a procedure defined in another class. External procedure calls are preceded by an object identifier, separated from the procedure name by a dot (.). If a procedure has parameters, the call has a list of the actual parameters in parenthesis. If no parameter list exists, no parenthesis are written.

9.4 Return

The return instruction causes a return from a routine.

BNF:

```
return                    ::=  return
```

Example:

```
show is
        == displays this element on standard output
```

```
do
    if name = "" then
        return
    …
end show
```

return does not take any parameters. If the routine is a function, all return values must be assigned before the *return* is executed.

> *Routine exits from anywhere in a routine (even in the middle of a loop!) are seen by some as "unstructured programming". For a good summary of the reasons that show the advantages (in terms of programming and teaching) of this construct, see Roberts, E.,* Loop Exits & Structured Programming: Reopening the Debate, *SIGCSE Bulletin, 27, 1, March 1995, pp. 268-272*

9.5 Assertion

Assertions are used as a tool for correctness assurance and debugging.

BNF:

assertion	::=	**assert** "**(**" *condition* "**)**"
condition	::=	*boolean-expression*

Example:

```
assert (name <> nil)
```

If the boolean expression in the assertion is not true, program execution is interrupted and the user is notified about failing of the assertion. If the expression is true, the statement has no effect.

10. Expressions

Expressions are:

- function call
- equality
- type equality
- *in*-expression (sets)
- create
- this

BNF:

expression	::=	*function-call* | *operator-expression*

| *reference-equality*
| *type-equality*
| *in*
| *create*
| *old*
| **this**

10.1 Function Call

A function call is a call to a routine which returns one or more values.

BNF:

function-call ::= *call-chain*
call-chain ::= *unqualified-call* { "**.**" *unqualified-call* }
unqualified-call ::= ***identifier*** ["**(**" *expression-list* "**)**"]

Examples:

```
x := func
a,b := func (1)
if obj.func (a,b+c) then ...

found, element := list.search (22);
if found then
    print (element)
end if
```

A function call is syntactically similar to a procedure call. Since a function returns values, it is used in an expression rather than as a statement. A function can return one or more values, which can then be assigned to variables in a (multi-)assignment.

If a function returns exactly one value, it can also be used in expressions such as the condition of an *if* statement or in an actual parameter list. Functions returning multiple values can only be used in a multi-assignment statement.

If a function has no parameters, a call to that function consists of its name only (no parentheses are written to indicate an empty parameter list).

10.2 Equality

BNF:

reference-equality ::= *expression comparison expression*
comparison ::= "=" | "<>"

Examples:

```
a = b
found = true
a <> 42
```

See section 3.2.2 (Comparison).

10.3 Type Equality

BNF:

type-equal	::=	*expression* **is** *type*

Examples:

```
var
    f: Figure
    r: Rect    – – inherits from figure
...
if f is Rect then
    r := f
end if
```

The keyword *is* checks whether a variable is of a given type. The result is a boolean value. The expression

ident is Mytype

is true, if the dynamic type of *ident* is *Mytype* or a subclass of *Mytype*. In other words: The expression is true, if the dynamic type of *ident* conforms to *Mytype*.

10.4 In

BNF:

in	::=	*expression* **in** *set_expr*
set_expr	::=	"**{**" [*set_elem* { "**,**" *set_elem* }] "**}**"
set_elem	::=	*expression* | *subrange*
subrange	::=	*expression* "**..**" *expression*

Example:

```
if num in {1, 3, 5, 7} then ...
if  n in {1..10} then ...
valid := x in {" ", "\t", "0".."9" }
```

In expressions are aliases and are defined by combinations of comparisons:

n in {a, b}	is defined as	*n = a or n = b*
n in {a..b}	is defined as	*n>=a and n<=b*
n in {a, b..c}	is defined as	*n = a or (n>=b and n<=c)*

10.5 Create

BNF:

create	::=	*general-create* | *array-create*
general-create	::=	**create** *class-ident* ["**(**" *expression-list* "**)**"]
array-create	::=	"**[**" *expression* { "**,**" *expression* } "**]**"

Example:

```
var
    r : Rectangle
    b : Buffer
...
    r := create Rectangle (p1, p2, c)
    b := create Buffer
```

The *create* keyword creates an object of the specified class and executes its creation routine. Parameters to the creation routine are passed in a parameter list after the class name. If the creation routine has no parameters, the create call has no parameter list. (If the class does not specify a creation routine, a default routine without parameters is generated).

10.6 This

BNF:

expression	::=	**this**

Example:

```
list.add (this)
```

The expression *this* is a reference to the currently active object. It can be used to pass the current object reference to other objects.

11. Control Structures

The remaining group of statements, not described so far, are control structures. Control structures are:

- conditional
- multi branch
- loop

BNF:

statement	::=	…
		| *conditional*
		| *selection*
		| *loop*

11.1 Conditional: The If Statement

BNF:

conditional	::=	**if** *condition* **then**
		statement-list
		{ **elseif** *expression* **then**
		statement-list }
		[**else**
		statement-list]
		end if
condition	::=	*boolean-expression*
statement-list	::=	{ *statement* }

Examples:

```
if val>0 then
    handle_positiv
elseif val=0 then
    handle_zero
else
    print ("The value was negative!")
    handle_error
end if
```

```
if n < 0 then
    n := –n
end if
```

The if statement is similar to conditionals in many other languages. Note that the if and else parts contain statement *lists*. This makes an explicit grouping

symbol at the beginning of the statement list unnecessary. The “end if” is *always* required.

11.2 Selection: The Case Statement

The case statement allows a multi-way branch depending on an expression. It is similar to the case statement in Pascal or the switch statement in C. The case labels are set constants and selection of a case is done by applying an *in* operation to the expression and the case labels. The expression is evaluated and the resulting value is checked for membership in the label sets. The statements at the first matching label are executed. Only the statements between the label and the next label (if any) are executed. Then execution resumes after the **end case** instruction. The **else** part is executed if the value was not a member of any of the label sets. If no set contains the value and the case statement does not contain an else part, no nested instructions are executed and execution continues after the case statement.

BNF:

selection	::=	**case** *expression* **of** { *set_expr* "**:**" *statement-list* } [**else** *statement-list*] **end case**

Examples:

```
case colour of
   {red}:
      label := "danger"
   {yellow}:
      label := "warning"
   {green}:
      label := "okay"
   else
      handle_error
end case


case value of
   {0..33}:
      print ("low")
   {34..66}:
      print ("medium")
   {67..99}:
      print ("high")
   {100}:
      print ("top")
end case
```

```
case ch of
    {" ", "\n", "\t"}:
        handle_whitespace
    {"a".."z", "A".."Z", "_"}:
        handle_letter
    {"0".."9"}:
        handle_number
    else
        handle_error
end case
```

If the sets used as case labels overlap and the test value is a member of more than one of the sets, only the instructions at the first match are executed.

11.3 Iteration: The Loop Statement

There is only one kind of loop structure in Blue. The loop is defined with the keywords **loop** ... **end loop**.

Exit from the loop is explicit by using the **exit on** keyword, which is part of the loop. Every loop must have at least one exit. Multiple exits are possible. By placing the exit at the beginning or end of the loop, behaviour of a pre-test or posttest loop (*while* and *repeat* in Pascal) can be implemented.

iteration ::= **loop**
statement-list
exit on *condition*
statement-list
{ **exit on** *condition*
statement-list }
end loop

Note that a statement list can be empty.

Examples:

The first example shows a loop with the exit instruction at the beginning. This construct has semantics similar to a while loop (where the condition in the while loop would be the negation of the exit condition in this example).

```
list.first
loop
    exit on list.atEnd
    print (list.current)
    list.advance
end loop
```

Similarly, a post-test loop (*repeat* in Pascal) can be constructed by placing the exit instruction at the end of the loop.

In Blue, however, exit instructions can appear at any stage in the loop, increasing flexibility:

```
loop
    readInt (i)
    exit on i>0
    print ("The number has to be greater than 0. Enter again.")
end loop
```

The last example shows the use of multiple exit instructions.

```
loop
    s := getSelection
    exit on s=0
    error := process (s)
    exit on error
end loop
```

A loop without an exit statement causes a compile time error. Exit statements are illegal outside a loop. In nested loops, exit always exits only the innermost loop. It is not possible to exit an outer loop from within a nested loop.

12. Pre/Post Conditions

Preconditions and postconditions are part of routine definitions. They are optional. If present, they are automatically checked at runtime and an error is reported if a condition is not met.

Preconditions are written at the beginning of a routine body (before the variable declarations) and are checked on entry of that routine. It is the responsibility of the caller to ensure that the precondition is met (i.e. that it evaluates to *true* on routine entry). The code inside the routine body can then safely assume that the condition is true.

Postconditions are written at the end of a routine body and are checked on exit of the routine. It is the responsibility of the routine to ensure that the postcondition is true, and the caller can assume that the condition is met on return from the routine.

BNF:

routine-body ::= *routine-comment*
[**pre** [*condition*]
[*comment*]]
...
do
...
[**post** [*condition*]
[*comment*]]
end *identifier*

Pre- and postconditions consist of one condition and/or a comment each. If multiple conditions are to be defined, they can be combined with a logical *and* to form one condition. The comment is intended to express conditions that cannot be expressed in Blue expressions. The comment has no runtime effect

(no error is generated if a comment is false), but it is good programming practice to include documentation about pre- and postconditions even if they cannot be written in code.

Examples:

```
func (n: Integer, m: Integer) -> (res: Integer) is
    pre
        n >= 0 and
        m >= 1
    var
        x : Integer
    do
        …
    post
        res <> nil
end func


printName is
    do
        …
    post
        == the name has been printed on screen
end printName
```

Both pre- and postconditions are part of the interface of a routine. Redefinitions of routines may alter these conditions only in restricted ways: preconditions may be weakened, postconditions may be strengthened. If a redefined routine does not define pre- or postconditions, the conditions of the original (parent) routine apply. If a precondition is redefined, the actual condition tested at runtime is

precondition **or** *parent-precondition*

If a postcondition is redefined, the condition tested at runtime is

postcondition **and** *parent-postcondition*

This ensures that the redefined routine guarantees at least as much as its parent.

Two special expression are available in conditions: `=>` (implies) and the reserved word **old**.

Example:

```
post
    found => index > 0
```

The implies symbol can be read as

if (found) then assert (index > 0)

The **old** expression is available only in postconditions. It is used on exit of a routine to refer to the value that an expression had on entry to that routine.

The following example ensures that the global variable *num* has not been altered in the routine:

Example:

```
doSomething (n: Integer) is
    pre
        n <> nil
    post
        num = old num
end doSomething
```

It is an error to use "=>" outside of a pre- or postcondition or class invariant or to use "**old**" outside a postcondition.

13. Class Invariants

Class invariants are conditions that have to be met by any stable state of an object. Stable states exists before and after every execution of an interface routine from the outside of an object. (Note that when an interface routine is called locally from within an object, it does not have to be in a stable state.)

BNF:

class-decl	::=	**class *identifier*** ["<" *generics-list* ">"] **is** *class-definition*
class-definition	::=	… | [**identifier**] *general-class-decl*
general-class-decl	::=	… [**invariant** *condition*] **end class**

Invariants are checked at runtime before and after every external interface routine call and after execution of the creation routine. A runtime error is generated if an invariant evaluates to *false*.

14. Comments

Blue recognises two types of comments, marked by a double equals sign (==) and a double hyphen (– –). Comments always extends to the end of the line. To have several lines as a comment, every line has to be preceded by the comment symbol.

14.1 Interface Comments

Comments starting with == are part of the class interface and part of the language definition. They are used to describe the class itself and routine semantics, and are allowed to appear only in strictly defined locations. These locations are:

- after the class header
- after a routine header

Example

```
class Rectangle is
   == Author:    M. Kölling
   == Date: April 1995
   == Version:   1.1
   == Short:     Graphical representation of a rectangle.
   ==
   ==Class Rectangle represents a rectangle with
   ==specified coordinates and colour that can be ...
   ==…
   …
interface
   creation (tl: Point, br: Point) is
         == Create rectangle at coordinates defined by tl (top-left)
         == and br (bottom-right).  Default colours are: fill white,
         == border black.
      do
         …
      end creation

   routines
      move (dx: Integer, dy: Integer) is
            == Move rectangle by distance defined by dx, dy.
         do
            …
         end move

      size -> (width: Integer, height: Integer) is
            == Return size of rectangle in width and height.
         do
            …
         end size
invariant
   …
end class
```

The example shows interface comments for the class and for each routine. Interface comments may not appear anywhere else. They are part of the class interface and therefore displayed in interface view (see [1] for a description of the environment, including the *interface view* of classes). They are also used by the class browser. (By convention, some lines of the class comments begin with certain keywords which are recognised by the class browser. These lines define the author, version number, date of creation and a short description of the class. See the documentation of the browser for details.)

14.2 Implementation Comments

All other comments, starting with the symbol – – (double hyphen), are implementation comments. They may appear anywhere and are not included in the interface view. Implementation comments are ignored by the compiler.

Note that implementation comments may appear as part of the routine comment. If they do, they should describe the routine implementation and will not be displayed as part of the interface.

Example:

```
class WidgetManager is
    …
    add (new_widget: Widget) is
        == Add 'new_widget' to the set of managed widgets.
        – – This is done by adding 'new_widget' to 'wl', the
        – – internal widget list. Note that 'wl' may be undefined
        – – if this is the first widget to be added. We have to test for
        – – that and possibly create the list before adding to it.
    do
        if wl = nil then                    – – test whether list exists
            wl := create List <Widget>
        end if
        wl.add (new_widget)                 – – add widget to widget list
    end add
    …
end class
```

The interface of this routine is

```
    add (new_widget: Widget)
== Add 'new_widget' to the set of managed widgets.
```

There is no block comment in Blue (every comment is for one line only). Instead, the Blue environment provides tools to add or remove comments on every line in a block of text.

15. I/O

I/O is implemented in Blue by providing a number of standard classes and objects that can be called to perform input/output operations. There are two groups of I/O classes: one group for text based (or "standard") I/O and one group for more sophisticated graphical user interface building. The graphical user interface classes are found in the Blue GUI library and are not described here. The following sections describe what is known as standard I/O, a simple, text based I/O facility.

15.1 Standard I/O

Blue provides four environment variables for standard I/O: *input*, *output*, *terminal* and *format*. These variables are predefined and accessible in all Blue classes.

It also provides two predefined objects: one object of class *TextTerminal* and one object of class *OutputFormat*. The predefined variables *input*, *output* and *terminal* typically all refer to the predefined TextTerminal object, the *format* variable refers to the OutputFormat object. It is possible, though, to bind *input* or *output* to other objects, for instance *output* to an object representing a printer or *input* to a file. This can be done externally before a routine is executed, or dynamically by the program.

The predefined variables are of the following classes:

```
var
    input : IO_Channel
    output : IO_Channel
    terminal : TextTerminal
    format : OutputFormat
```

IO_Channel is a superclass of *TextTerminal*, so the variables *input* and *output* provide a view onto the predefined TextTerminal object as an IO_Channel object.

The variables *input* and *output* exist to provide a means do redirect input or output to and from other channels (such as a text file, which is also a subclass if IO_Channel).

The variable *terminal* exists so that a program can take advantage of TextTerminal specific operations that are not part of IO_Channel (such as positioning the cursor on screen, etc.)

For a full description and interface, see the "Blue Standard Library Manual" [2].

15.1.1 Standard Output

Output to the terminal is usually done using the alias *print*.

Examples:

```
i : Integer
m : Myclass
```

```
print (i)
print (m)
print ("The value of ", i, " is ", m, "\n")
```

The alias *print* is defined as

print (a, b, ...) <=> output.write (str (a, b, ...))

str itself is an alias, which is defined as

str (a, b, ...) <=> a.toString.concat (b.toString.concat (...))

The function *concat* is defined in class String and returns a string that is the concatenation of two other strings. The function *toString* is defined for all predefined classes (except Arrays) and may be defined for user defined classes. It returns a printable representation of an object in a string. If a user defined class wants to define a *toString* routine, it has to conform to the following signature:

```
toString -> (s: String)
    == Return a string representation of this object.
```

Every object that defines a function *toString* conforming to this specification, can be printed using the alias *print*.

Furthermore, apart from being used for output, the *str* alias can be used for conversion to string from another type and for concatenation of string representations of any type, including strings.

Examples:

```
s, s1, s2 : String
i : Integer
m : Myclass

s := str (i)                        -- conversion Integer to String
s := str ("The value is ", i)       -- conversion and concatenation
s := str (s1, s2)                   -- concatenation of Strings
```

15.1.2 Output Formatting

The *toString* routines of the predefined classes *Integer, Real*, *Boolean* and *Enumeration* use the predefined *format* object to determine their output format. (*format* is a predefined constant referring to a standard object of class OutputFormat.) Calls to *format* may be made to influence the formatting of number, Boolean and enumeration output. The (incomplete) interface of OutputFormat is as follows:

```
class interface OutputFormat is

  setWidth (n : Integer)
    == Set field width for output. Default is 0. If the string
    == representation of a value consists of more characters than
    == the field width, the width is ignored.
    ==
    == Used by Integer, Real, Boolean and Enumeration.
    ==
    == Note that the class String does not use the format object for
    == output. String output is unaffected by the field width. To
```

```
        == align strings, use the 'fill' function in class String.
        pre
            n >= 0
    alignRight (right : Boolean)
        == Specify alignment of output in the field defined by 'setWidth'.
        == The default is 'left' (spaces will be added to the right of the
        == output as appropriate).  If 'right' is set to true, output will
        == be right-aligned within the specified field width.
        ==
        == Used by Integer, Real, Boolean and Enumeration.
        pre
            right <> nil
    scientific (useScientific : Boolean)
        == If useScientific is true, scientific notation is used for output
        == of real numbers (e.g. 1.23456e+02).  Otherwise fixed point
        == notation is used (e.g. 123.456).  The default is
        == useScientific=false.
        ==
        == Used by Real.
        pre
            useScientific <> nil
    roundTo (n : Integer)
        == Round output of fixed point real numbers to n digits after the
        == decimal point.  Default is 6.  Used by Real.
        pre
            n >= 0
end class
```

The *format* object may also be used by user defined classes to determine the behaviour of the *toString* routine (so that user defined output can, for instance, honour field width or justification). For a full interface of *OutputFormat*, see Appendix D8.

15.1.3 Standard Input

Input from standard input is defined for integers, reals, booleans and strings. It is done using a group of aliases which map to routine calls to the standard object *input*. The aliases are:

```
readInt     <=>  input.readInt
readReal    <=>  input.readReal
readStr     <=>  input.readStr
readChar    <=>  input.readChar
```

All of these routines are functions returning objects of the according type.

Examples:

```
num := readInt
name := readStr
line [i] := readChar
```

readInt and *readReal* skip whitespace and end-of-line characters. If the next characters can not be interpreted in the required format, they return *nil*. *readStr* reads until an end-of-line character is found, or the end of file is reached. The relevant part of the interface of the class IO_Channel is shown below:

```
class interface IO_Channel is
    . . .

  readStr -> (s : String) is deferred
    == Read a String.  Reads characters until
    == a line break ("\n") is read or an error condition (such as
    == "end-of-file" is encountered).  The resulting String does
    == not include the line break character.

  readChar -> (s : String) is deferred
    == Read the next character.  Every character (including line
    == break characters) are returned as entered. The line break
    == character is returned as "\n".

  readInt -> (i : Integer) is deferred
    == Read an Integer.  Skips white space (spaces, tabs and
    == newlines) before the Integer.  Returns "nil" if the
    == next non-white characters do not represent a number.
    == Numbers are written with an optional - or + sign and digits.

  readReal -> (r : Real) is deferred
    == Read an Real (floating point number).  Skips white space
    == (spaces, tabs and newlines) before the number.  Returns "nil"
    == if the next non-white characters cannot be interpreted as a
    == number. Real numbers are written with an optional - or +
    == sign, digits and a decimal point (.).

  endLine is deferred
    == Read and discard all characters up to (and including) the next
    == NewLine character.

  atEnd  -> (isAtEnd : Boolean) is deferred
    == Return true if the current position is the end of the channel.
    == Some channels might not have an end (and atEnd always
    == returns false) some might reach an end at a specific point (like
    == a file, where the end of the channel might be defined by the
    == end of the file). In some channels the end might be
    == dynamically defined (i.e. when a terminal allows the user to
    == generate and at-end condition via a control key). See the
    == concrete subclasses for their individual handling of the at-end
    == condition.
end class
```

15.1.4 The TextTerminal Class

As mentioned above, the variables *input, output* and *terminal* by default refer to an object of class TextTerminal, which is created by the Blue environment on system startup (a project does not need to create this object explicitly). The TextTerminal class defines additional routines apart from those defined in IO_Channel. Those routines can be called only through the variable *terminal*, since it is the only one to be declared of class TextTerminal. There are no aliases for these routines. They are called using the standard routine call syntax. Fir example, to position the cursor on the screen, a class might include the statement

terminal.cursorTo (4, ln)

See Appendix D7 for the complete interface of class TextTerminal.

15.2 File I/O

Working with files is supported by standard classes, available in the standard library (group *Standard_IO*). These are the classes *FileSystemHandle* and *TextFileHandle*. Objects of both of these classes are *handles* – they give access to an underlying object. This indicates that, for instance, creating a file system handle does not create a file system, but rather creates an object that gives the user access to the underlying filesystem which exists independently from the handle object. Equally, creating a TextFileHandle does not create a text file – it just allows access to a text file that was created earlier.

Class FileSysHandle

The class FileSysHandle is used to access the file system. Access to the file sytem allows operations such as creating or removing files, reading directories, checking for the existence of files, etc.

The (incomplete) definition of the class *FileSysHandle* is

```
class interface FileSysHandle is
  ...

  creation (fileSys : String)
     == Create a handle to a file system.  The parameter "fileSys"
     == identifies the file system to be accessed.  On systems that
     == have only one file system (e.g. Unix), the parameter is
     == ignored.

  routines

  createFile (name : String) -> (done : Boolean)
     == Create a file with file name "name".  If the file exists, its
     == length is truncated to 0.
     ==  ...

  deleteFile (name : String) -> (done : Boolean)
     == Delete the file with file name "name".
     ==  ...
```

```
    copyFile (name : String, newName : String)
                                        -> (done : Boolean)
        == Copy the file with name "name" to a second file with name
        == "newName".
        == ...

    createDirectory (name : String) -> (done : Boolean)
        == Create a directory with name "name".
        == ...

    deleteDirectory (name : String) -> (done : Boolean)
        == Delete the directory with name "name".  The directory must
        == exist and must be empty.
        == ...

    readDirectory (name : String) -> (dir : Array <String>)
        == Read the directory with name "name".  Returns an array with
        == the directory entries.  Each entry is the name of a file or
        == directory.   ...

    exists (name : String) -> (result : Boolean)
        == Return result=true if an entry with name "name" exists in
        == the file system.

    . . .

end class
```

This interface is incomplete. Look at the interface online in the Blue class browser, or have a look at "The Blue Libraries" [2] to see the full interface.

Class TextFileHandle

A file handle is used to access the contents of a file. A file handle is not the file itself. This is an important distinction. Creating a file handle does not create a file. Creation of a file handle rather corresponds to opening a file.

Example:

This example creates a copy of an text file.

```
copy (source: String, target: String) is
        == Copy the file "source" to file "target" character by character.
    var
        filesys : FileSysHandle
        file1 : TextFileHandle
        file2 : TextFileHandle
    do
        filesys := create FileSysHandle (nil)

        – – create the target file

        if (not filesys.createFile (target)) then
            print ("Could not create target file!\n")
            return
```

```
        end if

        – – open both files

        file1 := create TextFileHandle (source)
        file2 := create TextFileHandle (target)

        if (not file1.isReadable) or (not file2.isWritable) then
            print ("Could not open files with necessary access.\n")
            return
        end if

        – – do the copying

        loop
            exit on file1.endOfFile
            file2.write (file1.readChar)
        end loop

        – – close the files

        file1.close
        file2.close
        print ("Done.\n")

    end copy
```

After creating a TextFileHandle, the state of the file handle should be checked. A file is always opened for reading and writing, if the underlying file system allows this access. If access rights in the file system deny read or write access, the file is opened in read-only or write-only mode. The access can be checked with the *isReadable* and *isWritable* functions. If no access is possible at all (either no access was granted by the file system, or the file does not exist) then *isReadable* and *isWritable* are both false, and *isBad* is true. All attempts to read from a file that is not readable, or to write to a file that is not writable result in runtime errors.

The function *endOfFile* can be used to check whether the current file position is at the end of the file. It must be checked *before* a read operation is attempted. Trying to read from a file while *endOfFile* is true is an error.

The (incomplete) definition of class *TextFileHandle* is

```
class interface TextFileHandle is IO_Channel

    creation (fileName : String)
        == Create a handle to a text file. The file "fileName" must exists
        == in the file system.
        ==
        == If a file with the specified name does not exist or if access to
        == the file is denied, no file will be opened and the status of the
        == TextFileHandle will be set to "isBad" (see routine "isBad"
        == below).
        == This routine does not create a file. To create a new file, use
        == the createFile routine in class FileSystemHandle.
        pre
```

fileName <> nil

routines

inherited from IO_Channel

write (s : String) is redefined
== Write 's' to the file.
pre
isWritable

readStr -> (s : String) is redefined
== Read a String from the file. Reads characters until
== a line break ("\n") is read or the end of the file is reached.
== The resulting string does not include the line break character.
== ...
pre
isReadable and not atEnd

readChar -> (s : String) is redefined
== Read the next character from the file. Every character
== (including line break characters) are returned as entered
== The line break character is returned as "\n".
pre
isReadable and not atEnd

readInt -> (i : Integer) is redefined
== Read an Integer from the file. Skips white space (spaces,
== tabs and newlines) before the Integer. Returns "nil" if the
== next non-white characters do not represent a number.
== Numbers are recognised with an optional - or + sign and
== digits.
pre
isReadable and not atEnd

atEnd -> (isAtEnd: Boolean)
== Return true if the end of input has been reached. While input
== from a terminal does not have a natural end, end end-of-input
== condition can be generated by pressing CTRL-D.
== If the next character entered is a CTRL-D, 'atEnd' removes
== the character from the input and returns true. If it is any other
== character, 'atEnd' leaves it pending in the input queue and
== returns false.
pre
not isBad

. . .

new routines

position -> (pos : Integer)
== Return the read/write position in the file.
== The position is the offset from the beginning of the file.
== The first position (start of file) is 0.

```
    pre
        not isBad

**setPosition (pos : Integer)**
    == Set the read/write position in the file to "pos".
    == 0 is the beginning of the file, -1 is the end of the file.
    == Positive numbers count from the beginning of the file,
    == negative numbers count backwards from the end of the file.
    pre
        not isBad and pos <> nil


**close**
    == Close the file associated with this handle.  In general all files
    == should be closed by the user as soon as they are not needed
    == anymore.  Closing files is important, because the number of
    == files a user can have open at the same time is limited.  If files
    == are not closed, opening of further files might fail.  The exact
    == number of files that can be open at any one time is system
    == dependent.
    == Also, output to files is buffered.  Text written to the file is
    == not immediately written to disk.  Only after closing the file or
    == calling an explicit "flush" can the user rely on the file content
    == being visible to other processes.
    ==
    == Files not being closed are automatically closed when the file
    == handle object is garbage collected.  Since the user has no
    == influence on the timing of garbage collection, it is bad practice
    == to rely on this.
    ==
    == After closing, the file handle is not usable any more, all read
    == and write operations cause runtime errors, and "isBad" returns
    == true.

**isReadable -> (readable : Boolean)**
    == Returns true if the file can be read.  If access rights in the
    == file system specify write-only access, isReadable returns
    == false.

**isWritable -> (writable : Boolean)**
    == Returns true if the file can be written.  If access rights in the
    == file system specify read-only access, isWritable returns false.

**isBad -> (bad : Boolean)**
    == "isBad" returns true if this file handle cannot be used to access
    == a file.  This can happen if the file specified in the creation
    == routine does not exist, or if neither read nor write access was
    == permitted.  If "isBad" is true, "isReadable" and "isWritable"
    == are both false.

**end class**
```

This interface is incomplete. Look at the interface online in the Blue class browser, or have a look at "The Blue Libraries" [2] to see the full interface.

16. Inheritance

Inheritance can be used to define specialisations of previously defined classes. These specialisations are called subclasses (or “children”) of their superclasses (or “parents”).

16.1 Defining Subclasses

Classes can be subclasses of at most one other class. The superclass (if any) is listed in the head of the class definition.

BNF:

class-decl	::= **class *identifier*** ["<" *generics-list* ">"] **is** *class-definition*
class-definition	::= [***identifier***] *general-class-decl* | ...

Example

```
class Car is Vehicle
    == Defines a car for a traffic simulation.
    ...
end class
```

A subclass inherits all routines and variables from its parent. All interface routines of the parent are also interface routines of the subclass, and all internals (routines and variables) are available inside the subclass. The creation routine is the only routine that is not inherited by the subclass.

Inherited interface routines cannot be "hidden" (removed from the interface). Inheritance is intended to be used to express real “is-a” relationships. If such a relationship is true, there is no need to hide interface routines of the parent. In all situations where a programmer wants to hide parts of an interface in a child, inheritance should not be used, but a client-server model (“uses” relationship) should be used instead.

16.2 Redefinition

Children may redefine the *implementation* of an inherited routine. This is done by providing a routine implementation in the child with the keyword *redefined* in the routine header.

BNF:

routine-decl	::=	**identifier** ["**(**" parameter-list "**)**"] ["->" "**(**" parameter-list "**)**"] **is** routine-impl
routine-impl	::=	[**redefined**] routine-body | ...

Example:

```
move (dest: Location) is redefined
      == Move the position of this ...
   do
      …
end move
```

The explicit keyword *redefined* protects a programmer from accidental reuse of a routine name of the parent. The parameter list in the redefined function must be the same as the one of the original function. Changing of parameter lists (covariance or contravariance) is illegal.

Only the implementation of routines may be redefined. It is not allowed to redefine the parameter or result list of an existing routine.

16.3 Calling Superclass Functions

Sometimes is is desirable to call the original of a function that has been redefined (usually as part of the redefinition of the function). Consider the following example:

Example:

```
class Person is
   …
   print is
      == Write this person's details to the screen
   do
      print (name, "\n")
      print ("date of birth: ", date, "\n")
   end print
end class


class Student is Person
```

```
    …
    print is redefined
        == Write this student's details to the screen
    do
        super!print                                  – – write Person details
        print ("student number: ", stud_num, "\n")
    end print
end class
```

The class *Person* has an interface routine *print* that prints out that person's details. The class *Student*, which inherits from *Person*, redefines print to print out the details for the student, which include the details for a person. *Student's* print routine can print the person details by calling the original print routine in its superclass. This is done by preceding the routine call with the keyword **super** and an exclamation mark (**!**).

BNF:

procedure-call	::=	*call-chain*
call-chain	::=	[**super** "**!**"] *unqualified-call* { "**.**" *unqualified-call* }
unqualified-call	::=	***identifier*** ["**(**" *expression-list* "**)**"]

This construct is often used to call the superclass's creation routine from a childs creation routine.

Example:

```
class Student is Person
    …
    interface
        creation (name: String, dob: Date, sid: Integer) is
            == Create student instance
        do
            super!creation (name, dob)
            student_num := sid
        end creation
    …
end class
```

I have often been asked why we use an exclamation mark rather than the usual dot notation in this situation. In other words: why do we write

super!creation

instead of

super.creation

The answer is that, although similar on first glance, these are two very different instructions. The dot notation specifies an object before the dot. The routine call is performed on that other object. The exclamation mark

(or "bang notation") specifies the execution of a routine that was defined in another class on this object. *The* super *keyword is a* scope specifier, *not an object specifier. We regard this distinction as so important that we want the programmer to be aware of the difference. The different syntax is an attempt to emphasise this difference.*

16.4 Deferred Routines

Routine implementations can be deferred. Deferred routines are marked with the keyword *deferred*.

BNF:

routine-decl	::=	**identifier** ["**(**" *parameter-list* "**)**"] ["**->**" "**(**" *parameter-list* "**)**"] **is** *routine-impl*
routine-impl	::=	**deferred** *routine-spec* | ...
routine-spec	::=	*routine-comment* [**pre** *condition*] [**post** *condition*] **end *identifier***

Example:

```
move (dest: Location) is deferred
    == Move the position of this ...
  pre
    dest <> nil
  end move
```

If a routine is deferred, the class does not provide an implementation of the routine. Classes that contain one or more deferred routines are called *abstract classes*. No instances can be created of abstract classes. Children of abstract classes can provide implementations for deferred routines. If they do not provide implementations for all deferred routines, they remain abstract themselves.

Implementations for deferred routines are provided by redefining the routine in the child. See section 16.2 for a description of redefinition.

17. Genericity

Classes may be generic. Genericity (often called *parametric polymorphism*) allows classes to include unspecified types in their class description, which are instantiated when an object of this class is created. This allows a more

general form of code reuse. A list, for instance, can be defined to hold elements of some type, where the type is not specified at the time the class is implemented. The actual type of the list elements is then specified when an object of class list is created. Several instances of the list can exist, one holding elements of type Integer, another one elements of type String, and so on.

17.1 Unconstrained Genericity

Most of the BNF rules involved in genericity have been mentioned above. Here we repeat the relevant parts:

BNF:

class-decl	::=	**class** ***identifier*** ["<" *generics_list* ">"] **is** *class-definition*
generics-list	::=	*generics-decl* { "**,**" *generics-decl* }
generics-decl	::=	*formal-generic-param* [**is** *class-type*]
formal-generic-param	::=	***identifier***
var-decl	::=	*ident-list* "**:**" *type* [*initilisation*]
type	::=	*class-type* | *formal-generic-param* | ...
class-type	::=	***identifier*** ["<" *actual-generics-list* ">"]
actual-generics-list	::=	*type* { "**,**" *type* }

The following example shows part of a class definition for a generic stack and code fragments declaring and using the stack class, instantiating it once as an Integer Stack and once as a Stack of Figures.

Example:

```
class Stack <ELEM_TYPE> is
internal
    var
        st: Array <ELEM_TYPE>

interface
        ...
    routines
        push (elem: ELEM_TYPE) is
            == push elem on stack
        pop -> (elem: ELEM_TYPE) is
            == pop top of stack and return it in elem
    ...
```

```
end class

class client
    uses Stack, Figure
  internal
    var
      s1: Stack <Integer>
      s2: Stack <Figure>
      f: Figure
    …
      s1.push (42)
      s2.push (f)
end class
```

Within the generic class (here: *Stack*) the formal generic parameter (*ELEM_TYPE*) is a valid type. It can be used for the declaration of variables, parameters and return values.

Classes may have more than one generic parameter. The list of types in the instantiation (in the variable declaration), the *actual generic parameters*, must have the same number of elements as the list of identifiers in the class header (the *formal generic parameters*).

Example:

```
class List <T1, T2> is
  …
end class

class Client
    uses List
  internal
    var   l: List <String, Integer>
  …
end class
```

17.1.1 Classes and Types

It is time to revisit the relationship between classes and types. Up to now, we did not distinguish very carefully between classes and types. This was no problem because, as long as no generic classes are involved, each class corresponds to exactly one type, and the type is referred to by using the class's name. Consider

```
var
  p : Person
```

We sometimes said "p is of class Person" or "p is of type Person". A more accurate (but rather clumsy) way to describe this situation is "p is declared of the type that is defined by class Person".

This now changes with generic classes.

Consider the class *Stack* shown above. *Stack* is a class, but because it is generic, it does not correspond to one type, but potentially to many types. Since variable declarations need types (not classes), the following declaration is wrong:

```
var
   s : Stack            – – ERROR   ERROR   ERROR !
```

Instantiating the generic parameters of a generic class generates a type. In other words: *Stack* is not a type, but *Stack<Integer>* is. The following declaration is legal:

```
var
   s : Stack <Integer>
```

Almost everywhere in Blue code where we have seen class names so far, it is actually a type that is expected. (The BNF has always shown this correctly.) For example, the actual generic parameters themselves are types, not classes. Thus the following declaration is an error:

```
var
   n: Array <Stack>             – – ERROR!
```

The declaration should read

```
var
   n: Array <Stack <Integer>>
```

The same is true for a supertype of a class, e.g.

```
class PersonList is List <Person> ...
```

and for type specifications in the *is* expression (see section 10.3).

The only place where class names, rather than a type, are specified is in the *uses* clause of a class.

17.1.2 Operations on Formal Generic Types

Since the dynamic type (the real type at runtime) of an object held in a variable declared of a formal generic type is not known, no special operation can be allowed on that object.

Consider:

```
class List <ELEM_TYPE> is
   …
   var
      head : ELEM_TYPE
   …
end class
```

At compile time of class *List* the dynamic type of *head* cannot be known. Only operations that are allowed on all objects of any class are allowed on variables of this type. These operations are listed in section 3.2. They are assignment and equality check (:=, =, <>).

17.2 Constrained Genericity

The generic type in generic classes can be constrained.

Example:

```
class List <T is Comparable> is
   ...
end class
```

In this example, all actual generic parameters for *List* in instantiations of this class have to be of a subclass of *Comparable* (or of class *Comparable* itself).

As a result of this, operations defined for *Comparable* can be used within the class *List* on variables declared of type *T*.

18. Concepts not included in Blue

This section discusses some concepts included in some other modern object-oriented languages that are not included in Blue.

These concepts are not explained in detail. The discussion rather focuses on their relationship to Blue and explains the motivation for their exclusion from the Blue specification.

18.1 Multiple Inheritance

While being a valuable concept for serious production languages, multiple inheritance is considered not important on the beginners' level. It is not a concept we want to teach in the first year, and its introduction into the language would have raised a series of related problems that would have considerably complicated the language and its implementation (such as repeated inheritance, solving of name clashes, etc.)

18.2 Routine Parameters

Passing routines as parameters is, while sometimes nice for elegant algorithms, not compatible with object-oriented programming. It assumes that pieces of code have an existence of their own, independent from classes.

In a pure object-oriented language code does only exist as part of a class. Passing a routine can be achieved (with a bit of overhead) by defining a class with that routine and passing an object of that class. Also, often constrained genericity offers an alternative solution to problems that can be solved by passing routines.

18.3 User Defined Infix Operators

User defined infix operators serve only for convenience in writing – they add no functional value. More importantly, they can be misused and can easily lead to code that is more confusing and less readable than code written without them. Reading and correctness is more important than writing. While there are a few (frequently mentioned) examples where they can sensibly be used, there seem to be many more where their use is tempting but confusing.

18.4 Function Overloading

In Blue, function overloading is included only for the predefined arithmetic operators (+, -, *, /), which are applicable to both Real and Integer numbers.

User defined overloading is not allowed.

The argument is similar as for user defined infix operators: while there are some nice applications, the gain through the use of function overloading seems to be too small and infrequent to make up for the added complexity and potential source of confusion and misunderstanding.

18.5 Union Type

A union type is not needed since similar constructs can be expressed with inheritance and supertypes.

18.6 Explicit Blocks

Explicit (user defined) blocks are not needed since in an object-oriented language functions are expected to be relatively short. Explicit blocks serve mainly as scope restrictions for variables and should be superfluous in that case.

Appendix A: EBNF

```
class-decl          ::= class identifier [ "<" generics_list ">"] is
                            class-definition

class-definition    ::= Enumeration enum-class-decl
                        | [ class_type ] general-class-decl

enum-class-decl     ::= class-comment
                            manifest ident-list
                        end class

general-class-decl  ::= class-comment
                        uses [ ident-list ]
                        [ internal
                            [ const const-decls ]
                            [ var var-decls ]
                            [ routines routine-decls ]
                        interface
                            [ creation [ "(" parameter-list ")" ] is
                                routine-impl ]
                            [ routines routine-decls ]
                        [ invariant condition ]
                        end class
```

```
generics-list       ::= generics-decl { "," generics-decl }
generics-decl       ::= formal-generic-param [ is class-type ]
formal-generic-param ::= identifier
```

```
const-decls         ::= const-decl { const-decl }
const-decl          ::= ident-list ":" type initialisation
```

```
var-decls           ::= var-decl { var-decl }
var-decl            ::= ident-list ":" type [ initilisation ]

ident-list          ::= identifier {"," identifier }
initilisation       ::= ":=" expression
```

```
type                    ::= class-type
                            | formal-generic-param

class-type              ::= identifier [ "<" actual-generics-list ">" ]
actual-generics-list    ::= type { "," type }
```

```
routine-decls           ::= routine-decl { routine-decl }
routine-decl            ::= identifier [ "(" parameter-list ")" ]
                                    [ "->" "(" parameter-list ")"] is
                                routine-impl

parameter-list          ::= param-decl {"," param-decl }
param-decl              ::= identifier ":" type

routine-impl            ::= deferred routine-spec
                            | builtin routine-spec
                            | [ redefined ] routine-body

routine-body            ::= routine-comment
                            [ pre condition ]
                            [ const const-decls ]
                            [ var var-decls ]
                            do
                                statement-list
                            [ post condition ]
                            end identifier

routine-spec            ::= routine-comment
                            [ pre condition ]
                            [ post condition ]
                            end identifier
```

```
statement               ::= assignment
                            | assignment-attempt
                            | procedure-call
                            | return
                            | assertion
                            | conditional
                            | selection
                            | loop
```

```
assignment              ::= indexed-ident-list ":=" expression-list
```

```
assignment-attempt      ::= indexed-ident-list "?=" expression-list
```

indexed-ident-list ::= *indexed-ident* {"**,**" *indexed-ident* }
expression-list ::= *expression* {"**,**" *expression* }

procedure-call ::= *call-chain*

return ::= **return**

assertion ::= **assert** "**(**" *condition* "**)**"

conditional ::= **if** *condition* **then**
statement-list
{ **elseif** *expression* **then**
statement-list }
[**else**
statement-list]
end if

condition ::= *boolean-expression*
statement-list ::= { *statement* }

selection ::= **case** *expression* **of**
{ *set_expr* "**:**" *statement-list* }
[**else** *statement-list*]
end case

loop ::= **loop**
statement-list
exit on *condition*
statement-list
{ **exit on** *condition*
statement-list }
end loop

boolean-expression	::=	*expression*
expression	::=	*function-call* | *operator-expression* | *reference-equality* | *type-equality* | *in* | *create* | *old* | **this**

function-call	::=	*call-chain*

call-chain	::=	[**super** "**!**"] *unqualified-call* { "**.**" *unqualified-call* }
unqualified-call	::=	***identifier*** ["**(**" *expression-list* "**)**"]

operator-expression	::=	. . . | *expression* "=>" *expression* | *entity*

reference-equality	::=	*expression comparison expression*
comparison	::=	"=" | "<>"

type-equality	::=	*expression* **is** *type*

in	::=	*expression* **in** *set_expr*
set_expr	::=	"**{**" [*set_elem* { "**,**" *set_elem* }] "**}**"
set_elem	::=	*expression* | *subrange*
subrange	::=	*expression* "**..**" *expression*

create	::=	*general-create* | *array-create*
general-create	::=	**create** *class-ident* ["**(**" *expression-list* "**)**"]
array-create	::=	"**[**" *expression* { "**,**" *expression* } "**]**"

old	::=	**old** *expression*

entity	::=	*indexed-ident* | *manifest-constant*
indexed-ident	::=	***identifier*** { "**[**" *expression* "**]**" }

manifest-constant	::=	***integer-constant*** | ***real-constant*** | ***boolean-constant*** | ***string-constant*** | *enum-constant*
enum-constant	::=	***identifier*** | *qualified-ident*
qualified-ident	::=	*class-ident* "**!**" ***identifier***

Appendix B: Complete List of Aliases

alias	original	applicable to
n + m	n.add (m)	Integer, Real
n – m	n.sub (m)	Integer, Real
– n	n.neg	Integer, Real
n * m	n.mult (m)	Integer, Real
i1 div i2	i1.div (i2)	Integer
n mod m	n.mod (m)	Integer
r1 / r2	r1.div (r2)	Real
n ^ m	n.power (m)	Integer, Real
a < b	a.greater (b)	Integer, Real, String
a > b	a.less (b)	Integer, Real, String
a <= b	a.greaterEq (b)	Integer, Real, String
a >= b	a.lessEq (b)	Integer, Real, String
not a	a.invert	Boolean
a or b	a.or (b)	Boolean
a and b	a.and (b)	Boolean
s[i]	s.substring (i,1)	String
a[i]	a.getElem (i)	Array*
a[i] := ...	a.putElem (i, ...)	Array*
str (a, b, ...)	a.toString.concat (b.toString.concat (...))	*any type*
print (a, b, ...)	output.write (str (a, b, ...))	*any type*

** context sensitive*

Appendix C: Implementation-Dependent Definitions

This Appendix lists all characteristics of Blue that are explicitly allowed to be different in different implementations. Blue programs should not rely on specific definitions for these features.

The implementation dependent features are

- The values of MAXINT and MININT.

Appendix D: Interfaces of Predefined Classes

D.1 Integer

class interface **Integer** is

```
================================================
== Author:    Michael Kölling
== Version:   1.0
== Date:      9 October 1996
== Short:     Blue standard Integer class
==
==  "Integer" is a standard class of the Blue language.  It is used to store
==  integer numbers.  "Integer" is predefined in Blue and thus does not
==  have to be imported (i.e. explicitly listed in the "uses" clause).  It
==  is automatically known in all classes.
==
==  User defined classes cannot inherit from "Integer".
==
================================================
```

routines

neg -> (result: Integer) is
== Return the negative value of this number.
== Alias: – (prefix) (e.g. *–4*)

add (other: Integer) -> (sum: Integer) is
== Return the sum of this number and the number *other*.
== Alias: + other (e.g. *3 + 4*)

sub (other: Integer) -> (diff: Integer) is
== Return the difference of this number and the number *other*.
== Alias: – other (e.g. *3 – 4*)

mult (other: Integer) -> (prod: Integer) is
== Return the product of this number and the number *other*.
== Alias: * other (e.g. *3 * 4*)

div (other: Integer) -> (quot: Integer) is
== Return the integer part of the quotient of this number and *other*.
== Alias: div other (e.g. *3 div 4*)

mod (other: Integer) -> (rem: Integer) is
== Return the remainder of the integer division of this number and
== *other*.
== Alias: mod other (e.g. *3 mod 4*)

pow (exp: Integer) -> (result: Integer) is
== Return this number raised to the power specified by *exp*.
== Alias: ^ other (e.g. *3 ^ 4*)

greater (other: Integer) -> (result: Boolean) is
== Return *true* if this number is greater than *other*, otherwise *false.*
== Alias: > other (e.g. *3* > *4*)

greaterEq (other: Integer) -> (result: Boolean) is
== Return *true* if this number is greater or equal than *other*, otherwise
==*false.*
== Alias: >= other (e.g. *3* >= *4*)

less (other: Integer) -> (result: Boolean) is
== Return *true* if this number is less than *other*, otherwise *false.*
== Alias: < other (e.g. *3* < *4*)

lessEq (other: Integer) -> (result: Boolean) is
== Return *true* if this number is less or equal than *other*, otherwise
== *false*.
== Alias: <= other (e.g. *3* <= *4*)

toString -> (s: String) is
== Return a string representation of this number. The standard format
== object (of class OutputFormat) is used to determine details of the
== appearance.
== Alias: str ()

end class

D.2 Real

class interface **Real** is

```
==================================================
== Author:    Michael Kölling
== Version:   1.0
== Date:      9 October 1996
== Short:     Blue standard Real class
==
==  "Real" is a standard class of the Blue language.  It is used to store
==  floating point numbers.  "Real" is predefined in Blue and thus does not
==  have to be imported (i.e. explicitly listed in the "uses" clause).  It
==  is automatically known in all classes.
==
==  User defined classes cannot inherit from "Real".
==
==================================================
```

creation
== Never used explicitly. "Real" is a manifest class, and all
== real values exist automatically during every execution.
== No further reals can be created at runtime.

routines

neg -> (res: Real)
== Negation. Return the real that represents the value

```
== "-this".
== Alias: -
    post
res <> nil

   add (other: Real) -> (res: Real)
== Addition.  Return the real that represents the value
== "this + other".
== Alias: +
    pre
other <> nil
    post
res <> nil

   sub (other: Real) -> (res: Real)
== Subtraction.  Return the real that represents the value
== "this - other".
== Alias: -
    pre
other <> nil
    post
res <> nil

   mult (other: Real) -> (res: Real)
== Multiplication.  Return the real that represents the value
== "this * other".
== Alias: *
    pre
other <> nil
    post
res <> nil

   divide (other: Real) -> (res: Real)
== Division.  Return the real that represents the value
== "this / other".
== Alias: /
    pre
other <> nil  and
other <> 0.0
    post
res <> nil

   power (n: Real) -> (res: Real)
== Power.  Return the real that is represents the value
== "this ^ n" ("this to the power of n").
== Alias: ^
    pre
n <> nil
    post
res <> nil

   sqrt -> (res: Real)
== Square root.  Return the real that is represents the value
== "sqrt (this)" ("square root of this").
    post
res <> nil
```

```
   trunc -> (res: Integer)
== Truncate to integer.  Return the integer that is represents the
== whole number part of this number (i.e. "this" is rounded towards
== zero).
     post
res <> nil

   round -> (res: Integer)
== Round to integer.  Return the integer that is nearest to the
== value of "this".  (Fractions of value 0.5 or greater will be
== rounded  away from zero, fractions less than 0.5 will be rounded
== towards zero.)
     post
res <> nil

   greater (other: Real) -> (res: Boolean)
== Greater than.  Return true if the value of this real is greater
== than the value of "other".  Return false otherwise.
== Alias: >
     pre
other <> nil
     post
res <> nil

   greaterEq (other: Real) -> (res: Boolean)
== Greater or equal.  Return true if the value of this real is
== greater than or equal to the value of "other".  Return false
== otherwise.
== Alias: >=
     pre
other <> nil
     post
res <> nil

   less (other: Real) -> (res: Boolean)
== Less than.  Return true if the value of this real is less
== than the value of "other".  Return false otherwise.
== Alias: <
     pre
other <> nil
     post
res <> nil

   lessEq (other: Real) -> (res: Boolean)
== Less or equal.  Return true if the value of this real is
== less than or equal to the value of "other".  Return false
== otherwise.
== Alias: <=
     pre
other <> nil
     post
res <> nil

   toString -> (s: String)
== Conversion to String.  Returns a string with a printable
== representation of this real number.
```

```
== Alias: str ()
    post
s <> nil

end class
```

D.3 Boolean

```
class interface Boolean is

=================================================
== Author:    Michael Kölling
== Version:   1.0
== Date:      8 October 1996
== Short:     Blue standard Boolean class
==
==  "Boolean" is a standard class of the Blue language.  It is used to store
==  truth values ("true" and "false").  "Boolean" is predefined in Blue and
==  thus does not have to be imported (i.e. explicitly listed in the "uses"
==  clause).  It is  automatically known in all classes.
==
==  User defined classes cannot inherit from "Boolean".
==
=================================================

   creation
== Never used explicitly.  "Boolean" is a manifest class, and the
== boolean values exist automatically during every execution.
== No further boolean values can be created at runtime.

 routines

   invert -> (res: Boolean)
== Negation.  Return "not this", i.e. "false" if this is "true",
== or "true" if this is "false".
== Alias: not
    post
res <> nil

   and (other: Boolean) -> (res: Boolean)
== Logical and.  Return the boolean value "this and other".
== Alias: and
    pre
other <> nil
    post
res <> nil

   or (other: Boolean) -> (res: Boolean)
== Logical or (inclusive).  Return the boolean value
== "this or other".
== Alias: or
    pre
other <> nil
    post
res <> nil
```

```
   toString -> (s: String)
== Conversion to String.  Returns a string with a printable
== representation of this boolean value.  The string returned is
== "true" for true and "false" for false.
== Alias: str ()
    post
s <> nil

end class
```

D.4 String

```
class interface String is

================================================
== Author:    Michael Kölling
== Version:   1.0
== Date:      24.9.96
== Short:     Blue standard String class
==
==  This class implements the standard String type for Blue.
==  Strings are a sequence of characters.  The first character in a string
==  has the index 1, the last index is equal to length(string).
==
==  Blue does not have a separate character type.  Characters are represented
==  by a string of length 1.
==
================================================

creation is

== Never used - strings are created by writing string literals
== in double quotes.
== Example: "This is a string"


routines

   length -> (l: Integer) is
== Return the length of the string (number of characters).

   concat (s: String) -> (newstring: String) is
== Return the string which is the concatenation of this string
== and 's'.  This string and 's' remain unchanged.
== Alias: str ()

   substring (start: Integer, len: Integer) -> (s: String) is
== Return the substring from this string which starts at 'start'
== with length 'len'.
== If "len" is nil, the substring from "start" to the end of the
== string is returned.
    pre
      (start >= 1) and (start <= length)
  and
(len <> nil => (start+len <= length+1))
```

```
   find (s: String, n: Integer) -> (pos: Integer) is
== Return the position of 's' in this string, starting the search at
== 'n'.  'pos' is the index where 's' was found or nil if not found.
     pre
       (s <> nil) and (n >= 1) and (n <= length)

   insert (s: String, pos: Integer) -> (newstring: String) is
== Return a string that is like this string with 's' inserted into this
== string at position 'pos'.
     pre
       (s <> nil) and (pos >= 1) and (pos <= length)

   delete (start: Integer, cnt: Integer) -> (newstring: String) is
== Return a string that is like this string with 'cnt' characters deleted,
== starting at position 'start'.
== If "cnt" is nil, all characters from "start" to the end of the
== string are deleted.
     pre
(start >= 1) and (cnt >= 0) and (cnt <= length)

   less (s: String) -> (is_less: Boolean) is
== Returns true, if this string is less than 's', else false.
== See class comment (above) for ordering of strings.
        == Alias:  <
    pre
s <> nil

   greater (s: String) -> (is_greater: Boolean) is
== Returns true, if this string is greater than 's', else false.
== See class comment (above) for ordering of strings.
== Alias:  >
     pre
s <> nil

   lessEq (s: String) -> (is_less_eq: Boolean) is
== Returns true, if this string is less or equal to 's', else
== false.
== See class comment (above) for ordering of strings.
== Alias:  <=
     pre
s <> nil

   greaterEq (s: String) -> (is_greater_eq: Boolean) is
== Returns true, if this string is greater or equal to 's', else
== false.
== See class comment (above) for ordering of strings.
== Alias:  >=
     pre
s <> nil

   toString -> (s: String)
== Conversion to String.  This function returns the string itself.
== It is an identity function.  There is usually no need to call
== this function explicitly.  It is provided to make the "str()"
== alias work with all standard classes, including string.
== Alias: str ()
```

```
    post
s <> nil

  toUpper -> (s: String)
== Return a string that is a copy of this string with all lower case
== letters replaced by their upper case equivalent.  Other character are
== unchanged.

  toLower -> (s: String)
== Return a string that is a copy of this string with all upper case
== letters replaced by their lower case equivalent.  Other character are
== unchanged.

  strip -> (s: String)
== Return a string that is a copy of this string with leadfing and
== trailing whitespace removed.  Whitespace are spaces, TAB
== characters ("\t") and newlines ("\n").

  fill (fill_char : String, front : Boolean, length : Integer) -> (s: String)
== Return a copy of this string that has characters added to the
== front or back.  'fill_char' is a character that is added to the
== front (if 'front'=true) or back (if 'front'=false) of this string
== until the string has length 'length'.  If the length of the
== initial string is already equal to or greater than length, the
== string remains unchanged.
==
== This function can be used to align string output.  E.g.
==
==            print (s.fill (" ", true, 20))
==
== prints the string s right aligned in a field 20 characters wide
    pre
(fill_char.length = 1) and (length >= 0)

  caseEqual (other: String) -> (is_equal: Boolean)
== "caseEqual" is a case-insensitive string equality test.
== It returns true, if this string and "other" are equal except
== for possible differences in the case of letters.

  ord -> (val: Integer)
== Return the ordinal value of the first character of this string.
== The ordinal value of a character is its internal byte
== representation, usually its ASCII (or, more correctly ISO) code.
== If the length of the string is 0, the result is 0.

  hash (limit: Integer) -> (hash_val: Integer)
== Return a hash value for a string.  The value is between 0
== and limit.
    pre
      limit > 0
    post
hash_val >= 0  and hash_val < limit

end class
```

D.5 Array

class interface **Array** <ELEM_TYPE> is

```
================================================
== Author:    Michael Kölling
== Version:   1.0
== Date:      14 November 1996
== Short:     Blue standard Array class
==
==  "Array" is a Blue standard class that is predefined in the Blue language.
==  Array objects can be used to store a number of objects of the same type.
==  Arrays are mainly a means for implementation of higher level collection
==  classes (such as sets, sequences, lists, etc.) but they can also be used
==  directly in user classes.
==
==  Arrays are best used in situations where the number of elements is known
==  in advance and does not change very often.  Resizing an array can be
==  relatively expensive in terms of both time and space required.  If the
==  number of elements changes regularly dynamically, consider using
== another collection (List, Set, etc.) from the standard collection library.
==
================================================
```

creation (size: Integer)
== Creation an array with 'size' elements.

routines

getElem (pos: Integer) -> (elem: ELEM_TYPE)
== Return the element at position 'pos'.
pre
pos >= 1 and
pos <= size

putElem (pos: Integer, elem: ELEM_TYPE)
== Assign 'elem' to position 'pos'.
pre
pos >= 1 and
pos <= size

init (val: ELEM_TYPE)
== Set all array elements to 'val'.

size -> (sz: Integer)
== Return the current size of the array.
post
sz >= 0

setSize (sz: Integer)
== Set the size of the array to 'sz'. If the current size is larger
== the elements at positions greater than 'sz' will be lost. If the
== current size is smaller, the elements at the new positions will
== be undefined.
pre
sz >= 0

```
    post
size = sz

end class
```

D.6 Enumeration

class interface **Enumeration** is

```
================================================
== Author:   Michael Kölling
== Version:  1.0
== Date:     30.11.96
== Short:    Abstract superclass for enumerations.
==
==  This class serves as a superclass for all enumeration classes.  It is
==  abstract — no objects can be created of this class directly.
==
================================================
```

routines

pred -> (previous: *SelfType*) is
    == Return the predecessor in this enumeration type. If there is no
    == predecessor, return *nil*.

succ -> (next: *SelfType*) is
    == Return the successor in this enumeration type. If there is no
    == successor, return *nil*.

ord -> (position: Integer) is
    == Return the ordinal position of this element in the enumeration list.
    == The first element has the ordinal 1, the next is 2, and so on.

toString -> (s: String) is
    == Conversion to String. Returns the name of this enumeration value
    == as a string
    == Alias: str ()

end class

Note that "SelfType" is no legal type in Blue. A Blue class like this cannot normally be written. The meaning of the word "SelfType" in the routine interfaces above indicates that for every concrete subclass of Enumeration that routine returns the type of the subclass. For an Enumeration "Colour", for example, the routine pred returns a result of type Colour.

D.7 TextTerminal

class interface **TextTerminal** is

```
================================================
== Author:       Michael Kölling
== Version:      1.1
== Date:         December 1997
== Short:        Standard text terminal for the Blue environment
==
==  The TextTerminal provides a simple standard terminal for text I/O.
```

```
==  Input and output is buffered by default.  The output buffer is flushed
==  at the end of each line (when a "\n" is written) and on a call of an
==  input routine.  If single characters have to be written, buffering
==  can be switched off (see below).
================================================
```

creation
== Create the TextTerminal

routines

inherited from IO_Channel:

write (s: String) is redefined
== Write 's' to the terminal

readstr -> (s: String) is redefined
== Read a string from the terminal. Reads characters until
== a line break ("\n") is read. The resulting string does
== not include the line break character.

readChar -> (s: String) is redefined
== Read the next character from the terminal. Every character
== (including line break characters) are returned as entered
== The line break character is returned as "\n".
== The input is buffered by default. (This means that editing the
== input line during input is possible for the user, and the program
== will return from this function only after a whole line was
== entered.) For unbuffered input, see getChar and askChar
== below.
==
== When expecting single character input, note that "readChar" does
== not discard the NewLine character. If the input is, for instance,
== "A<Enter>", then "readChar" will return the "A", while the <Enter>
== remains in the input queue (as a NewLine character). Thus the next
== input operation will immediately read a NewLine. To deal with this
== situation, use readStr instead or call "endLine" after "readChar".

readInt -> (i: Integer) is redefined
== Read an integer from the terminal. Skips white space (spaces,
== tabs and newlines) before the integer. Returns "nil" if the next
== non-white characters do not represent a number.
== Numbers are written with an optional - or + sign and digits.

readReal -> (r: Real) is redefined
== Read a real number from the terminal. Skips white space (spaces,
== tabs and newlines) before the number. Returns "nil" if the next
== non-white characters do not represent a real number.
== Numbers are written with an optional - or + sign, digits and a
== decimal point.

endLine is redefined
== Read and discard all characters up to (and including) the next
== NewLine character.

atEnd -> (isAtEnd: Boolean)
== Return true if the end of input has been reached. While input
== from a terminal does not have a natural end, end end-of-input

```
    == condition can be generated by pressing CTRL-D.
    == If the next character entered is a CTRL-D, 'atEnd' removes the
    == character from the input and returns true. If it is any other
    == character, 'atEnd' leaves it pending in the input queue and
    == returns false.
```

new routines:

```
getChar -> (s: String) is redefined
    == Get the next character from the terminal (unbuffered).
    == This function is similar to "readChar" (see above), but the
    == input is not buffered. The function returns as soon as a
    == character is entered - no line editing is provided.

askChar -> (s: String)
    == Check whether there is a character to be read from the terminal.
    == If so, read it.
    == This function is similar to "getChar" (see above), but the
    == function always returns immediately.  If a character has been
    == entered, that character is returned, otherwise "nil" is returned.

show
    == Show the terminal window.
    == This routine opens the window if it was not open, de-iconifies
    == the window if it was iconified and brings it to the top of the
    == window stack.

hide
    == Hide the terminal window.        The window is closed.

clear
    == Clear the terminal window and set the cursor to 0,0 (the upper
    == left corner)

width -> (columns: Integer)
    == Return the current width of the terminal window in characters.
    == (Note that the terminal currently does not support resizing.)

height -> (lines: Integer)
    == Return the current height of the terminal window in text lines.
    == (Note that the terminal currently does not support resizing.)

cursorTo (x: Integer, y: Integer)
    == Set the cursor to position (x,y). The legal ranges for
    == x and y are:
    ==     x: 0 .. width-1
    ==     y: 0 .. heigth-1
    == If x or y is outside its range, it is set to the nearest
    == legal value.
    == This routine works only after the terminal window has been
    == exposed at least once (otherwise the terminal size is not known).
    pre
        x <> nil  and  y <> nil

cursorOn
    == Switch the screen cursor on. (This is the default.)
```

```
    cursorOff
        == Switch the screen cursor off.

    buffered (buf_on: Boolean)
        == Set the output buffering mode.   If buffering is on (the default)
        == then the output line is buffered.  The buffer is flushed when
        == a newline is written or when an input routine is called.
        == If output should be visible immediately without a newline, switch
        == buffering off. Output without buffering is slower for most
        == purposes.
        pre
            buf_on <> nil

    inputEcho (echo : Boolean)
        == Switch input echo on/off.  The default is "on" (echo=true).
        == When echo is on, characters appear on the screen as they are
        == typed.  When echo is off, characters typed on the keyboard do
        == not appear on screen.
        pre
            echo <> nil

end class
```

D.8 OutputFormat

```
class interface OutputFormat is

==================================================
== Author:          Michael Kölling
== Version:         1.0
== Date:            1.10.1997
== Short:    Class defining output formatting options for terminal output.
==
==  The OutputFormat class defines some output formatting options for
==  numbers and other data.  The Blue system automatically creates an object
==  of this class at startup.  This object is accessible from all Blue classes
==  through the predefined constant "format".
==
==  The following classes use the format object to format their output:
==      Integer, Real, Boolean, Enumeration
==  User defined classes may use this object for the same purpose if they
==  wish.  The class String does not use the format object (see routine
==  "fill" in class String for formatting string output).
==
==  All the classes above use the field width and alignment setting from this
==  class.  The class Real uses, in addition to this, the 'scientific' and
==  rounding settings.
==
==================================================

    creation
        == Create format object.  No parameters needed.

routines
```

```
    setWidth (n : Integer)
        == Set field width for output.  Default is 0.  If the string
        == representation of a value consists of more characters than
        == the field width, the width is ignored.
        ==
        == Used by Integer, Real, Boolean and Enumeration.
        ==
        == Note that the class String does not use the format object for
        == output. String output is unaffected by the field width.  To
        == align strings, use the 'fill' function in class String.
        pre
            n >= 0

    alignRight (right : Boolean)
        == Specify alignment of output in the field defined by 'setWidth'.
        == The default is 'left' (spaces will be added to the right of the
        == output as appropriate).  If 'right' is set to true, output will
        == be right-aligned within the specified field width.
        ==
        == Used by Integer, Real, Boolean and Enumeration.
        pre
            right <> nil

    scientific (useScientific : Boolean)
        == If useScientific is true, scientific notation is used for output
        == of real numbers (e.g. 1.23456e+02).  Otherwise fixed point
        == notation is used (e.g. 123.456).  The default is useScientific=false.
        ==
        == Used by Real.
        pre
            useScientific <> nil

    roundTo (n : Integer)
        == Round output of fixed point real numbers to n digits after the
        == decimal point.  Default is 6.  Used by Real.
        pre
            n >= 0

    getWidth -> (n : Integer)
        == Return the current field width.

    isRightAligned -> (right : Boolean)
        == True if right alignment is currently on.

    isScientific -> (scientific : Boolean)
        == True if scientific notation is currently on.

    getRound -> (n : Integer)
        == Return the current round value.

    reset
        == Reset all values to their defaults

end class
```

Index

–– *See* comment, implementation
! 50
:= 4
<> 5
= 5
== *See* comment, interface
=> *See* implies

—A—

abstract *See* class, abstract
alias **2**
  bracket ([]) 10
  list of 63
  print 39
  readChar 41
  readInt 41
  readReal 41
  readStr 41
  str 40
Array 3, 5, **10**
  boundaries 10
  class interface 73
  example 10
  index type 10
  literal 10
assertion 27
assignment 4, **24**
  multi- 24
assignment attempt 25

—B—

blocks 56
Boolean 3, 4, 5, **7**, 19
  class interface 69
builtin 23

—C—

case 32
  of characters 15
character
  and string 8
  special 9
child *See* inheritance
class **3**
  abstract 51
  and type 2, 53
  comment *See* comment, interface
  dynamic **4**
  enumeration **13**
  example 11, 16
  general **11**
  generic *See* genericity
  interface of predefined 65
  invariant *See* invariant
  manifest **4**, 13
  predefined 3, **5**, 40
  user defined 40
comment **36**
  implementation 38
  interface 37
  routine 22
comparison 5
concat 40
conditional 31
conformance 24, **25**, 29
const *See* constant
constant **19**
  initialisation 20
  named 19
constrained genericity *See* genericity, constrained
control structures 31
conversion
  Integer to Real 6, 7
  to String 40
create **30**
creation routine *See* routine, creation
current object *See* this

—D—

data
  internal 11
deferred **51**
defined **17**
dynamic *See* class, dynamic
dynamic type 25

—E—

EBNF 57
else *See* conditional
encapsulation 17
Enumeration 4, **13**
  class interface 74
  example 13
  qualified 14
environment variables 39
equality **28**
evaluation
  partial 7
exit
  from loop 33
  from routine 33
expression **27**

—F—

FileHandle 44
FileSystem 43
format (predefined constant) 39, 40
format (predfined constant) 41
function 21

function call **2 8**

—G—

genericity **5 1**
  constrained 55
  eaxmple 52
  unconstrained 52
GUI 39

—I—

I/O 39
  file 43
identifier **1 5**
if-statement *See* conditional
implementation dependent 64
implemetation comment *See* comment, impl.
implies 35
in operator 20
infix operators
  user defined 56
information hiding *See* encapsulation
inheritance **4 8**
  and uses 16
  multiple 55
initialisation
  of objects 20
In-operator **29**, 32
  example 29
input 43
  standard 41
input (variable) 39
Integer 3, 4, 5, **6**, 19
  class interface 65
interface 12, 37
  and inheritance 48
  view 12
interface comment *See* comment, interf.
interface routine 11, 48
internal
  data 11
internal routine 11
invariant 36
is-operator *See* type equality
iteration 33

—L—

literal **1 9**
  Array 10
  Boolean 7
  Integer 6
  Real 7
  set 20
  String 8
loop 33
  pretest/posttest 33

—M—

manifest *See* class, manifest
MAXINT 6
MININT 6
multi-assignment **24**, 28

—N—

nil **17**, 26, 42

—O—

old 35
operations
  general **4**
operator
  precedence 14
output
  formatting 40
  standard 39
output (variable) 39
OutputFormat
  class interface 77
OutputFormat 39, 40
overloading 56

—P—

parameter **2 1**
  formal generic 53
parameters
  routines as 55
parent *See* inheritance
pass by value 21
polymorphism *See* genericity
postcondition 22, **3 4**
precondition 22, **3 4**
predefined
  and uses 16
  classes 3, **5**, 40
  objects 39
  variables 39
print *See* alias, print
procedure 21
procedure call **2 6**
program 11
Programming Environment **1**, 12
project 2

—R—

readChar *See* alias, readChar
readInt *See* alias, readInt
readReal *See* alias, readReal
readStr *See* alias, readStr
Real 3, 4, 5, **6**, 19
  class interface 66
redefine 35, **49**, 50, 51
redefinition *See* redefine
repeat loop 33
return **2 6**
return value 21, 27, 28
routine **21**, 48
  builtin *See* builtin
  comment *See* comment, interface
  creation 11, 30, 48
  deferred *See* deferred
  example 21
  interface *See* interface routine
  internal *See* internal routine

visibility 22
runtime error
assertion failure 27
invariant violation 36
missing loop exit 34
pre/postcondition violation 34
undefined result 22
undefined variable 17

—S—

scope
of constants 19
of routines 22
of variables 16
selection 32
set constant **20**, 32
stable state 36
statement **24**
static type 25
str *See* alias, str
String 3, 4, 5, **8**, 19
and characters 8
and manifest 8
class interface 70
long 9
subclass *See* inheritance
super 49, 50
superclass *See* inheritance
calling *See* super
supertype 54
syntax
special *See* alias

—T—

TextTerminal 39, **43**
class interface 74
this **30**
toString 40, 41
type 52
and class 53
dynamic *See* dynamic type
formal generic 54
predefined **5**
static *See* static type
user defined **11**
type equality **29**, 54

—U—

undefined **17**, 22
user interface 39
uses **15**

—V—

variable 4, **16**, 48
and nil 17
encapsulation 17
lifetime 17
state and value 17
visibility
of routines 22

—W—

while loop 33