**Eiffel Guide**

Contents

[Introduction 2](#_Toc368687634)

[Basics 5](#_Toc368687635)

[Hello World and basic class syntax 5](#_Toc368687636)

[Routine syntax 7](#_Toc368687637)

[Standard types 7](#_Toc368687638)

[Operators 11](#_Toc368687639)

[Arithmetic operators 12](#_Toc368687640)

[Relational operators 13](#_Toc368687641)

[Boolean operators 13](#_Toc368687642)

[Free operators 14](#_Toc368687643)

[Precedence 15](#_Toc368687644)

[Control structures 15](#_Toc368687645)

[Conditional 15](#_Toc368687646)

[Multi-branch 15](#_Toc368687647)

[Loop 16](#_Toc368687648)

[Extended types and constructions 18](#_Toc368687649)

# Introduction

Eiffel is an ISO-standardized, object-oriented programming language designed by Bertrand Meyer (an object-orientation proponent and author of Object-Oriented Software Construction) and Eiffel Software in 1986.

Here is an overview of the facilities supported by Eiffel:

* Completely object-oriented approach. Eiffel is a full-fledged application of object technology, not a "hybrid" of O-O and traditional concepts.
* External interfaces. Eiffel is a software composition tool and is easily interfaced with software written in such languages as C, C++, Java and C#.
* Full lifecycle support. Eiffel is applicable throughout the development process, including analysis, design, implementation and maintenance.
* Classes as the basic structuring tool. A class is the description of a set of run-time objects, specified through the applicable operations and abstract properties. An Eiffel system is made entirely of classes, serving as the only module mechanism.
* Consistent type system. Every type is based on a class, including basic types such as integer, boolean, real, character, string, array.
* Design by Contract. Every system component can be accompanied by a precise specification of its abstract properties, governing its internal operation and its interaction with other components.
* Assertions. The method and notation support writing the logical properties of object states, to express the terms of the contracts. These properties, known as assertions, can be monitored at run-time for testing and quality assurance. They also serve as documentation mechanism. Assertions include preconditions, postconditions, class invariants, loop invariants, and also appear in "check" instructions.
* Exception handling. You can set up your software to detect abnormal conditions, such as unexpected operating system signals and contract violations, correct them, and recover
* Information hiding. Each class author decides, for each feature, whether it is available to all client classes, to specific clients only, or just for internal purposes.
* Self-documentation. The notation is designed to enable environment tools to produce abstract views of classes and systems, textual or graphical, and suitable for reusers, maintainers and client authors.
* Inheritance. You can define a class as extension or specialization of others.
* Redefinition. An inherited feature (operation) can be given a different implementation or signature.
* Explicit redefinition. Any feature redefinition must be explicitly stated.
* Subcontracting. Redefinition rules require new assertions to be compatible with inherited ones.
* Deferred features and classes. It is possible for a feature, and the enclosing class, to be specified -- including with assertions -- but not implemented. Deferred classes are also known as abstract classes.
* Polymorphism. An entity (variable, argument etc.) can become attached to objects of many different types.
* Dynamic binding. Calling a feature on an object always triggers the version of the feature specifically adapted to that object, even in the presence of polymorphism and redefinition.
* Static typing. A compiler can check statically that all type combinations will be valid, so that no run-time situation will occur in which an attempt will be made to apply an inexistent feature to an object.
* Object test ("type narrowing"). It is possible to check at run time whether the type of an object conforms to a certain expectation, for example if the object comes from a database or a network.
* Multiple inheritance. A class can inherit from any number of others.
* Feature renaming. To remove name clashes under multiple inheritance, or to give locally better names, a class can give a new name to an inherited feature.
* Repeated inheritance: sharing and replication. If, as a result of multiple inheritance, a class inherits from another through two or more paths, the class author can specify, for each repeatedly inherited feature, that it yields either one feature (sharing) or two (replication).
* No ambiguity under repeated inheritance. Conflicting redefinitions under repeated inheritance are resolved through a "selection" mechanism.
* Unconstrained genericity. A class can be parameterized, or "generic", to describe containers of objects of an arbitrary type.
* Constrained genericity. A generic class can be declared with a generic constraint, to indicate that the corresponding types must satisfy some properties, such as the presence of a particular operation.
* Garbage collection. The dynamic model is designed so that memory reclamation, in a supporting environment, can be automatic rather than programmer-controlled.
* No-leak modular structure. All software is built out of classes, with only two inter-class relations, client and inheritance.
* Once routines. A feature can be declared s "once", so that it is executed only for its first call, subsequently returning always the same result (if required). This serves as a convenient initialization mechanism, and for shared objects.
* Standardized library. The Kernel Library, providing essential abstractions, is standardized across implementations.
* Other libraries. Eiffel development is largely based on high-quality libraries covering many common needs of software development, from general algorithms and data structures to networking and databases.

It is also useful, as in any design, to list some of what is not present in Eiffel. The approach is indeed based on a small number of coherent concepts so as to remain easy to master. Eiffel typically takes a few hours to a few days to learn, and users seldom need to return to the reference manual once they have understood the basic concepts. Part of this simplicity results from the explicit decision to exclude a number of possible facilities:

* No global variables, which would break the modularity of systems and hamper extendibility, reusability and reliability.
* No union types (or record type with variants), which force the explicit enumeration of all variants; in contrast, inheritance is an open mechanism which permits the addition of variants at any time without changing existing code.
* No in-class overloading, which by assigning the same name to different features within a single context, causes confusions, errors, and conflicts with object-oriented mechanisms such as dynamic binding. (Dynamic binding itself is a powerful form of inter-class overloading, without any of these dangers.)
* No goto instructions or similar control structures (break, exit, multiple-exit loops) which break the simplicity of the control flow and make it harder or impossible to reason about the software (in particular through loop invariants and variants).
* No exceptions to the type rules. To be credible, a type system must not allow unchecked "casts" converting from a type to another. (Safe cast-like operations are available through object test.)
* No side-effect expression operators confusing computation and modification.
* No low-level pointers, no pointer arithmetic, a well-known source of bugs. (There is however a type POINTER, used for interfacing Eiffel with C and other languages.)

Additional info:

<http://en.wikipedia.org/wiki/Eiffel_(programming_language)>

[http://docs.eiffel.com](http://docs.eiffel.com/)

<http://eiffelroom.org/>

# Basics

## Hello World and basic class syntax

When discovering any approach to software construction, however ambitious its goals, it is reassuring to see first a small example of the big picture -- a complete program to print the famous "Hello World" string. Here is how to perform this fascinating task in the Eiffel notation.

**note**

description: "Root for trivial system printing a message"

author: "Elizabeth W. Brown"

**class**

HELLO

**create**

make

**feature**

make

-- Print a simple message.

**do**

io.put\_string ("Hello World")

io.put\_new\_line

**end**

**end** -- class HELLO

The two versions perform identically; the following comments will cover the more complete second one.

Note the absence of semicolons and other syntactic clatter or clutter. You may in fact use semicolons to separate instructions and declarations. But the language's syntax is designed to make the semicolon optional (regardless of text layout) and it's best for readability to omit it, except in the special case of successive elements on a single line.

The **note** clause does not affect execution semantics; you may use it to associate documentation with the class, so that browsers and other indexing and retrieval tools can help users in search of reusable components satisfying certain properties. Here we see two notes, labeled description and author.

The name of the class is HELLO. Any class may contain "features"; HELLO has just one, called make. The **create** clause indicates that make is a "creation procedure", that is to say an operation to be executed at class instantiation time. The class could have any number of creation procedures.

The definition of make appears in a feature clause. There may be any number of such clauses (to separate features into logical categories), and each may contain any number of feature declarations. Here we have only one.

The line starting with -- (two hyphen signs) is a comment; more precisely it is a "header comment", which style rules invite software developers to write for every such feature, just after the point at which the feature is named. The tools of EiffelStudio know about this convention and use it to include the header comment in the automatically generated class documentation.

The body of the feature is introduced by the **do** keyword and terminated by **end**. It consists of two output instructions. They both use io, a generally available reference to an object that provides access to standard input and output mechanisms; the notation io.f, for some feature f of the corresponding library class (STD\_FILES, in this case), means "apply f to io". Here we use two such features:

put\_string outputs a string, passed as argument, here "Hello World".

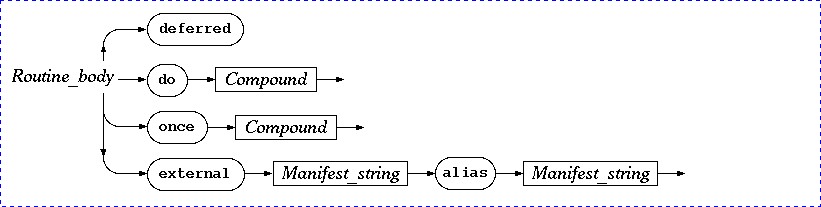
put\_new\_line terminates the line.

Rather than using a call to put\_new\_line, the first version of the class simply includes a new-line character, denoted as %N (the percent sign is used to introduce codes for special characters), at the end of the string. Either technique is acceptable.

You may have noticed another difference between the two versions. The first version uses a call to print where the second uses io.put\_string . Here too, the effect is identical and either technique is acceptable. In the next section, you will begin to see how things like io and print become available for use in a class like HELLO.

## Routine syntax

## D:\DOSE2013\SyntaxDiagramOf_Routine.jpg



## Standard types

Each of the standard types like INTEGER, REAL, etc. is represented by an Eiffel class written in a class file (e.g. integer\_32.e) in the kernel library.

The standard types consist of the basic types (INTEGER, REAL, ... ) and some other types (like STRING) with clearly defined standardized semantics.

The basic types available in Eiffel are:

BOOLEAN

CHARACTER, CHARACTER\_8, CHARACTER\_32

INTEGER, INTEGER\_8, INTEGER\_16, INTEGER\_32, INTEGER\_64

NATURAL, NATURAL\_8, NATURAL\_16, NATURAL\_32, NATURAL\_64

REAL, REAL\_32, REAL\_64

The basic types are all expanded. I.e. an entity of type INTEGER represents an integer value (i.e. an integer object) and not a reference to an integer object. All expanded types have copy semantics, i.e. assignment causes a copy of the value and not just the assignment of a reference.

The basic types are not just expanded, they are also immutable. It is not possible to change the value of an INTEGER, i.e. with a variable i of type INTEGER, there is no operation like increment.

All basic types have default values. They need not be initialized explicitely. The default values are zero, False or the character with code zero for INTEGERs/REALs, BOOLEANs and CHARACTERs respectively.

CHARACTER, INTEGER, NATURAL and REAL have the above written sized variants. CHARACTER, INTEGER, NATURAL and REAL are not individual types, they are just synonyms for one of their sized variants.

The sized variant can be chosen by a compiler option. The usual default is CHARACTER\_8, INTEGER\_32, NATURAL\_32 and REAL\_32.

A BOOLEAN can hold the truth values True and False.

An INTEGER\_n represents a signed integer value in the range -2n-1 ..2n-1 - 1, where n is either 8, 16, 32 or 64. I.e.

* INTEGER\_8: -128..127
* INTEGER\_16: -32768..32767
* INTEGER\_32: -2147483648..2147483647
* INTEGER\_64: -9223372036854775808..9223372036854775807

The NATURALs are unsigned are represent values in the range 0..2n-1

* NATURAL\_8: 0.. 255
* NATURAL\_16: 0 .. 65535
* NATURAL\_32: 0 .. 4294967295
* NATURAL\_64: 0 .. 18446744073709551615

The REALs are floating point number in IEEE format. There is a 32 bit REAL\_32 and a 64 bit REAL\_64 floating point number.

An INTEGER constant is any sequence of decimal digits within the range of INTEGER (remember INTEGER is either a synonym for INTEGER\_32 or INTEGER\_64). For better readability underscores can be used to group the digits (recommendation: groups of three decimal digits).

INTEGER constants can be given in hexadecimal (base 16, prefix 0x), octal (base 8, prefix 0c) and binary (base 2, prefix 0b) as well. Eiffel uses prefixes to indicate the number base.

* 0xFF -- decimal value 255
* 0xa -- decimal value 10
* 0x8000\_0000 -- decimal value -2147483648
* 0xffff\_ffff -- decimal value -1
* 0c40 -- decimal value 32 = 4\*8
* 0c77 -- decimal value 63 = 7\*8 + 7
* 0b1\_0000\_0000 -- decimal value 256 = 28
* 0b1111 -- decimal value 15 = 24 - 1

A CHARACTER constant is written as one printable CHARACTER within single quotes.

Non printable characters can be represented by the escape sequences

|  |  |  |
| --- | --- | --- |
| %A | At sign | @ |
| %B | Backspace | -- |
| %C | Circumflex | ^ |
| %D | Dollar | $ |
| %F | Formfeed | -- |
| %H | Backslash | \ |
| %L | Tilde | ~ |
| %N | Newline | -- |
| %Q | Backquote | ` |
| %R | CarriageReturn | CR |
| %S | Sharp | # |
| %T | HorizontalTab | -- |
| %U | NUll | NUL |
| %V | Vertical bar | | |
| %% | Percent | % |
| %' | Single quote | ' |
| %" | Double quote | " |
| %( | Opening bracket | [ |
| %) | Closing bracket | ] |
| %< | Opening brace | { |
| %> | Closing brace | } |

It is also possible to define character constants by its character code in the form '%/code/'. The character code can be given in decimal, hexadecimal, octal or binary form. E.g.

* '%/32/' -- character 32, i.e. blank in decimal,
* '%/0x20/' -- in hexadecimal,
* '%/0c40/' -- in octal,
* '%/0b1\_0000/' -- and in binary notation

Valid REAL constants are

* 1.
* 1.0
* 1e4
* .5
* 0.5

A string constant, or string literal, is a sequence of zero or more characters surrounded by double quotes, as in

"I am a string"

or

"" -- the empty string

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; %" represents the double quote character. Example of a string with embedded escape sequences:

"A string with double quote %" and non printables like %T"

A long string can be line wrapped across several source lines, e.g.

"hello, %

%world"

is equivalent to

"hello, world"

Another possibility is to use verbatim strings. The verbatim string

"{

Hello,

world.

Don't forget me!

}"

is equivalent to

" Hello,%N world.%N Don't forget me!%N"

Since the line sequence delimited by "{ and }" is taken verbatim, the blanks in front of the text on the lines are taken verbatim as well. This is sometimes not wanted. There is a variant which strips off any common initial blanks and tabs which uses the delimiters "[ and ]". The verbatim string

"[

Hello,

world.

Don't forget me!

]"

is equivalent to

"Hello,%Nworld.%NDon't forget me!%N"

Only the indentation common to all lines is stripped off. If one or more lines are indented relative to the others, that indentation is kept. E.g.

"[

Hello,

world.

Don't forget me!

]"

is equivalent to

"Hello,%N world.%NDon't forget me!%N"

Declarations

All entities must be declared. The following example shows typical declarations.

Pi: REAL = 3.14159265358979323846 -- a real constant

Name: STRING = "Joe Cartwright" -- a string constant

ival1, ival2, ival3: INTEGER -- variable attributes

rval: REAL

Nearly all semicolons in Eiffel are optional. They are inserted for better readability if more than one declaration or statement is placed on one line.

Symbolic constants can be declared only at the class level (constant attributes). There are no local and no global symbolic constants. If a class wants access to symbolic constants it either has to declare them as constant attributes in its class text or inherit them as constant attributes from a parent class.

## Operators

In Eiffel operators are just aliases for feature names. The expression a + b \* c is a shorthand for a.plus (b.product (c))

An operator alias is declared like

**class** INTEGER **feature**

...

plus **alias** "+" (other: **like** Current): **like** Current

**do** ... **end**

product **alias** "\*" (other: **like** Current): **like** Current

**do** ... **end**

...

**end**

Operators allow us to write expressions in a more natural manner. Furthermore operators have precedences which allow us to avoid a lot of parentheses and make the source code more readable.

Any class can use any operator for an alias of its features as long as there is no name clash (i.e. different features must have different names and different aliases). The precedence of the operators cannot be changed, the precedence is defined by the language.

### Arithmetic operators

The class INTEGER uses the binary arithmetic operators +,-,\* the integer division //, the real division / and the power operator ^.

Integer division truncates the fractional part (i.e. 5//2 = 2), the expression

x \\ y

produces the remainder when x is divided by y, and thus is zero when y divides x exactly.

E.g., a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 are leap years. Therefore

**local**

year: INTEGER

**do**

...

**if** year \\ 4 = 0 **and** year \\ 100 /= 0 **or** year \\ 400 = 0 **then**

print ( year.out + " is a leap year%N" )

**else**

print ( year.out + " is a leap year%N" )

**end**

...

**end**

This example already shows that the binary arithmetic operators have precedence over the relational operators (=, /=, ~, /~, ...). The relational operators have precedence over the boolean binary operators (**and**, **or**, ...) and **and** takes precedence over **or** (detailed precedence table see below)

Real division / applied to INTEGERs returns a REAL (i.e. 1/2 = 0.5).

Division by zero (all numeric types) results in an exception.

For negative operands the direction of the truncation of the integer division a //b and the sign of the result of a\\b is undefined.

Overflow during arithmetic operations is not detected by the runtime. Addition and substraction is done with circular arithmetic (i.e. Largest\_integer + 1 = Smallest\_integer). Multiplication on n-bit INTEGER/NATURALs is done as if it were done with 2n bit size and the result truncated to n bits (i.e. the most significant n bits removed).

The INTEGERs/NATURALs have a power operator ^ to do the exponentiation a^b. The exponent must not be negative. The exponentiation a^b returns the same result as the repeated multiplication a\*a\*...\*a (b times, with b>=0).

The REALs have an exponentiation operator as well. The exponentiation a^b with REALs evaluates to a^b = exp(b\*log(a)), were exp(x) is the exponential function and log(x) is the natural logarithm. For a <= 0 the runtime throws an arithmetic exception.

The operators // and \\ are not defined for REALs.

### Relational operators

The relational operators are

< <= > >= = /~

They all have the same precedence and are not associative. Expression like

a < b < c -- invalid expression

or

a = b = c -- invalid expression

are invalid and rejected by the parser.

### Boolean operators

The boolean operators are

**not** -- unary

**and or xor** -- binary strict

**and then or else implies** -- binary semistrict

The binary operators **and**, **or** and **xor** are strict. In exp1 and exp2 both expressions exp1, exp2 are evaluted and then the boolean value of exp1 and exp2 will be evaluated.

The operators **and then**, **or else** and **implies** are semistrict. Evaluation stops as soon as the truth or falsehood of the result is known. Therefore in some cases only the first operand will be evalutated by the runtime. We get the semantics

a **and then** b -- evaluate a; if a is false the result is false

-- if a is true, the result is the value of b

a **or else** b -- evaluate a; if a is true the result is true

-- if a is false, the result is the value of b

a **implies** b -- evaluate a; if a is false the result is true

-- if a is true, the result is the value of b

Note: Parentheses are necessary, because the relational operator = has higher precedence than the boolean operators.

The relative precedence of the boolean operators is

**not** -- highest

**and and then**

**or xor or else**

**implies** -- lowest

All binary boolean operators associate left to right. This is inline with general practice in most modern programming language. The only unusual thing might be that

a **implies** b **implies** c

is equivalent to

( a **implies** b ) **implies** c

because implies is an operator which is not available in most other programming languages.

### Free operators

In Eiffel you can define free operators like e.g.

!-!

@

|>

<|

-|->

<-|-

==>

<==

++

You can form free operators by a sequence of the operator symbols

: \ ? = ~ / ! # $ % & \* + - / < > @ ^ ` |

but you are not allowed to clash with sequences which have already a defined meaning. Some examples of invalid free operators

-- -- -- initiates a comment

--> -- -- initiates a comment

? -- ? alone is a placeholder for agents, combinations ?/? are valid

+ -- + is a standard operator and not a free operator

<= -- <= is a standard operator for "less equal"

= -- = is the standard identity operator

/= -- /= is the standard not identity operator

-> -- -> already used for constraints of formal generics

### Precedence

The following table summarizes all precedence and associativity rules. Note that the rules are not complicated and in line with common practice. In order to minimize parentheses and maximize readability it is worthwhile to know these rules.

|  |  |  |
| --- | --- | --- |
| **precedence** | **associativity** | **operators** |
| 10 |  | old not + - (unary) all free unary operators |
| 9 |  | all free binary operators |
| 8 | right to left | ^ |
| 7 | left to right | \* / // \\ |
| 6 | left to right | + - (binary) |
| 5 |  | .. |
| 4 |  | = /= ~ /~ < > <= >= |
| 3 | left to right | **and and then** |
| 2 | left to right | **or xor or else** |
| 1 | left to right | **implies** |

## Control structures

### Conditional

A conditional instruction has the form

**if** ... **then**

...

**elseif** ... **then**

...

**else**

...

**end**

The **elseif** ... **then** ... part (of which there may be more than one) and the **else** ... part are optional. After **if** and **elseif** comes a boolean expression; after **then** and **else** come zero or more instructions.

### Multi-branch

A multi-branch instruction has the form

**inspect**

exp

**when** v1 **then**

inst

**when** v2 **then**

inst2

...

**else**

inst0

**end**

where the **else** inst0 part is optional, exp is a character or integer expression, v1, v2, ... are constant values of the same type as exp, all different, and inst0, inst1, inst2, ... are sequences of zero or more instructions.

The effect of such a multi-branch instruction, if the value of exp is one of the vi, is to execute the corresponding insti. If none of the vi matches, the instruction executes inst0, unless there is no **else** part, in which case it triggers an exception.

*Note:* Raising an exception is the proper behavior, since the absence of an **else** indicates that the author asserts that one of the values will match. If you want an instruction that does nothing in this case, rather than cause an exception, use an **else** part with an empty inst0. In contrast, **if** c **then** inst **end** with no **else** part does nothing in the absence of an **else** part, since in this case there is no implied claim that c must hold.

### Loop

The loop construct provides a flexible framework for iterative computation. Its flexibility lies in how the complete form can be tailored and simplified for certain purposes by including or omitting optional parts.

Here's one example:

my\_list: LINKED\_LIST [STRING]

**from**

my\_list.start

**until**

my\_list.off

**loop**

print (my\_list.item)

my\_list.forth

**end**

.

What is happening in this loop? Let's dissect it and see.

First there is the initialization part:

**from**

my\_list.start

The first thing to occur in the execution of the base loop is the execution of any instructions in the initialization part (it is permissible for the initialization part to be empty of instructions, but the keyword **from** must be present to distinguish the base loop form). In our example, the feature start is applied to my\_list which will attempt to set the list cursor to the first element in my\_list.

The Exit condition part:

**until**

my\_list.off

The exit condition part of the loop construct defines the conditions under which the loop body (explained below) should no longer be executed. In our example, the loop will no longer execute if the cursor is "off", that is, there is no current item. So, if the list is empty, the loop body will not execute at all.

The loop body part:

**loop**

print (my\_list.item)

my\_list.forth

The loop body part contains the sequence of instructions to be executed during each iteration. In the example, that includes printing the current list item and then advancing the cursor. At some point, the cursor will pass the last item in the list, causing the exit condition to become true and stop the loop's execution. So, at the risk of stating the obvious, the key to loops that always complete is to ensure that there is something in the loop body that is guaranteed always to cause the exit condition eventually to become true. Loop correctness will discussed in more detail later.

And finally, there's the End part:

**end**

## Arrays and strings

Container classes in EiffelBase use standard names for basic container operations:

is\_empty : BOOLEAN

has (v : G ): BOOLEAN

count : INTEGER

item : G

make

put (v : G )

remove (v : G )

wipe\_out

start, finish

forth, back

In designing container structures, avoid hardwired limits! “Don’t box me in”: EiffelBase is paranoid about hard limits. Most structures conceptually unbounded .Even arrays (bounded at any particular time) are resizable. When a structure is bounded, the maximum number of items is called capacity.

How do we handle variants of a container class distinguished only by the item type?

Solution: *genericity* allows explicit type parameterization consistent with static typing

Container structures are implemented as generic classes:  
  
LINKED\_LIST [G]

pl : LINKED\_LIST [PERSON]  
 sl : LINKED\_LIST [STRING]  
 al : LINKED\_LIST [ANY]