WIKIPEDIA

Eiffel (programming language)

Eiffel is an object-oriented programming language designed by Bertrand Meyer (an object-orientation proponent and author of *Object-Oriented Software Construction*) and Eiffel Software. Meyer conceived the language in 1985 with the goal of increasing the reliability of commercial software development;[5] the first version becoming available in 1986. In 2005, Eiffel became an ISO-standardized language.

The design of the language is closely connected with the Eiffel programming method. Both are based on a set of principles, including design by contract, command–query separation, the uniform-access principle, the single-choice principle, the open–closed principle, and option–operand separation.

Many concepts initially introduced by Eiffel later found their way into Java, C#, and other languages.[6] New language design ideas, particularly through the Ecma/ISO standardization process, continue to be incorporated into the Eiffel language.

Eiffel

Paradigm	Object-oriented, Class-based, Generic, Concurrent
Designed by	Bertrand Meyer
Developer	Eiffel Software
First appeared	1986[1]
Stable release	EiffelStudio 19.05[2] / 22 May 2019
Preview release	EiffelStudio 20.05[3] / 17 June 2020
Typing discipline	static
Implementation language	Eiffel
Platform	Cross-platform
OS	FreeBSD, Linux, Mac OS X, OpenBSD, Solaris, Windows
License	dual and enterprise
Filename extensions	.e
Website	www.eiffel.org (https://www.eiffel.org/)

Major implementations

EiffelStudio, LibertyEiffel, SmartEiffel, Visual Eiffel, Gobo Eiffel, "The Eiffel Compiler" tecomp

Influenced by

Contents

Characteristics

Design goals
- Background
- Implementations and environments
- Specifications and standards

Syntax and semantics
- Overall structure
  - Scoping
  - "Hello, world!"
- Design by contract
- Void-safety
- Features: commands and queries
- Overloading
- Genericity
- Inheritance basics
- Deferred classes and features
- Renaming
- Tuples
- Agents
- Once routines
- Conversions

Exception handling
Concurrency
Operator and bracket syntax, assigner commands
Lexical and syntax properties
Style conventions
Interfaces to other tools and languages
References
External links

Ada, Simula, Z
Influenced
Ada 2012, Albatross, C#, D, Java, Racket, Ruby,[4] Sather, Scala

Characteristics

The key characteristics of the Eiffel language include:

- An object-oriented program structure in which a class serves as the basic unit of decomposition.
- Design by contract tightly integrated with other language constructs.
- Automatic memory management, typically implemented by garbage collection.
- Inheritance, including multiple inheritance, renaming, redefinition, "select", non-conforming inheritance (http://docs.eiffel.com/book/method/et-inheritance#Non-conforming_inheritance), and other mechanisms intended to make inheritance safe.
- Constrained and unconstrained generic programming
- A uniform type system handling both value and reference semantics in which all types, including basic types such as INTEGER, are class-based.
- Static typing
- Void safety, or static protection against calls on null references, through the attached-types mechanism.
- Agents, or objects that wrap computations, closely connected with closures and lambda calculus.
- *Once* routines, or routines evaluated only once, for object sharing and decentralized initialization.
- Keyword-based syntax in the ALGOL/Pascal tradition but separator-free, insofar as semicolons are optional, with operator syntax available for routines.
- Case insensitivity
- Simple Concurrent Object-Oriented Programming (SCOOP) facilitates creation of multiple, concurrently active execution vehicles at a level of abstraction above the specific details of these vehicles (e.g. multiple threads without specific mutex management).

Design goals

Eiffel emphasizes declarative statements over procedural code and attempts to eliminate the need for bookkeeping instructions.

Eiffel shuns coding tricks or coding techniques intended as optimization hints to the compiler. The aim is not only to make the code more readable, but also to allow programmers to concentrate on the important aspects of a program without getting bogged down in implementation details. Eiffel's simplicity is intended to promote simple, extensible, reusable, and reliable answers to computing problems. Compilers for computer programs written in Eiffel provide extensive optimization techniques, such as automatic in-lining, that relieve the programmer of part of the optimization burden.

Background

Eiffel was originally developed by Eiffel Software, a company founded by Bertrand Meyer. *Object-Oriented Software Construction* contains a detailed treatment of the concepts and theory of the object technology that led to Eiffel's design.[7]

The design goal behind the Eiffel language, libraries, and programming methods is to enable programmers to create reliable, reusable software modules. Eiffel supports multiple inheritance, genericity, polymorphism, encapsulation, type-safe conversions, and parameter covariance. Eiffel's most important contribution to software engineering is design by contract (DbC), in which assertions, preconditions, postconditions, and class invariants are employed to help ensure program correctness without sacrificing efficiency.

Eiffel's design is based on object-oriented programming theory, with only minor influence of other paradigms or concern for support of legacy code. Eiffel formally supports abstract data types. Under Eiffel's design, a software text should be able to reproduce its design documentation from the text itself, using a formalized implementation of the "Abstract Data Type".

Implementations and environments

EiffelStudio is an integrated development environment available under either an open source or a commercial license. It offers an object-oriented environment for software engineering. EiffelEnvision is a plug-in for Microsoft Visual Studio that allows users to edit, compile, and debug Eiffel projects from within the Microsoft Visual Studio IDE. Five other open source implementations are available: "The Eiffel Compiler" tecomp; Gobo Eiffel; SmartEiffel, the GNU implementation, based on an older version of the language; LibertyEiffel, based on the SmartEiffel compiler; and Visual Eiffel.

Several other programming languages incorporate elements first introduced in Eiffel. Sather, for example, was originally based on Eiffel but has since diverged, and now includes several functional programming features. The interactive-teaching language Blue, forerunner of BlueJ, is also Eiffel-based. The Apple Media Tool includes an Eiffel-based Apple Media Language.

Specifications and standards

The Eiffel language definition is an international standard of the ISO. The standard was developed by ECMA International, which first approved the standard on 21 June 2005 as Standard ECMA-367, Eiffel: Analysis, Design and Programming Language. In June 2006, ECMA and ISO adopted the second version. In November 2006, ISO first published that version. The standard can be found and used free of charge on the ECMA site.[8] The ISO version[9] is identical in all respects except formatting.

Eiffel Software, "The Eiffel Compiler" tecomp and Eiffel-library-developer Gobo have committed to implementing the standard; Eiffel Software's EiffelStudio 6.1 and "The Eiffel Compiler" tecomp implement some of the major new mechanisms—in particular, inline agents, assigner commands, bracket notation, non-conforming inheritance, and attached types. The SmartEiffel team has turned away from this standard to create its own version of the language, which they believe to be closer to the original style of Eiffel. Object Tools has not disclosed whether future versions of its Eiffel compiler will comply with the standard. LibertyEiffel implements a dialect somewhere in between the SmartEiffel language and the standard.

The standard cites the following, predecessor Eiffel-language specifications:

- Bertrand Meyer: Eiffel: The Language, Prentice Hall, second printing, 1992 (first printing: 1991)
- Bertrand Meyer: Standard Eiffel (revision of preceding entry), ongoing, 1997–present, at Bertrand Meyer's ETL3 page, and
- Bertrand Meyer: Object-Oriented Software Construction, Prentice Hall: first edition, 1988; second edition, 1997.
- Bertrand Meyer: Touch of Class: Learning to Program Well with Objects and Contracts, Springer-Verlag, 2009 ISBN 978-3-540-92144-8 lxiv + 876 pages Full-color printing, numerous color photographs

The current version of the standard from June 2006 contains some inconsistencies (e.g. covariant redefinitions). The ECMA committee has not yet announced any timeline and direction on how to resolve the inconsistencies.

Syntax and semantics

Overall structure

An Eiffel "system" or "program" is a collection of *classes*. Above the level of classes, Eiffel defines *cluster*, which is essentially a group of classes, and possibly of *subclusters* (nested clusters). Clusters are not a syntactic language construct, but rather a standard organizational convention. Typically an Eiffel program will be organized with each class in a separate file, and each cluster in a directory containing class files. In this organization, subclusters are subdirectories. For example, under standard organizational and casing conventions, `x.e` might be the name of a file that defines a class called X.

A class contains *features*, which are similar to "routines", "members", "attributes" or "methods" in other object-oriented programming languages. A class also defines its invariants, and contains other properties, such as a "notes" section for documentation and metadata. Eiffel's standard data types, such as `INTEGER`, `STRING` and `ARRAY`, are all themselves classes.

Every system must have a class designated as "root", with one of its creation procedures designated as "root procedure". Executing a system consists of creating an instance of the root class and executing its root procedure. Generally, doing so creates new objects, calls new features, and so on.

Eiffel has five basic executable instructions: assignment, object creation, routine call, condition, and iteration. Eiffel's control structures are strict in enforcing structured programming: every block has exactly one entry and exactly one exit.

Scoping

Unlike many object-oriented languages, but like Smalltalk, Eiffel does not permit any assignment into attributes of objects, except within the features of an object, which is the practical application of the principle of information hiding or data abstraction, requiring formal interfaces for data mutation. To put it in the language of other object-oriented programming languages, all Eiffel attributes are "protected", and "setters" are needed for client objects to modify values. An upshot of this is that "setters" can, and normally do, implement the invariants for which Eiffel provides syntax.

While Eiffel does not allow direct access to the features of a class by a client of the class, it does allow for the definition of an "assigner command", such as:

```
    some_attribute: SOME_TYPE assign set_some_attribute

    set_some_attribute (v: VALUE_TYPE)
                -- Set value of some_attribute to `v'.
            do
                some_attribute := v
            end
```

While a slight bow to the overall developer community to allow something looking like direct access (e.g. thereby breaking the Information Hiding Principle), the practice is dangerous as it hides or obfuscates the reality of a "setter" being used. In practice, it is better to redirect the call to a setter rather than implying a direct access to a feature like `some_attribute` as in the example code above.

Unlike other languages, having notions of "public", "protected", "private" and so on, Eiffel uses an exporting technology to more precisely control the scoping between client and supplier classes. Feature visibility is checked statically at compile-time. For example, (below), the "{NONE}" is similar to "protected" in other languages. Scope applied this way to a "feature set" (e.g. everything below the 'feature' keyword to either the next feature set keyword or the end of the class) can be changed in descendant classes using the "export" keyword.

```
feature {NONE} -- Initialization
    default_create
            -- Initialize a new `zero' decimal instance.
        do
            make_zero
        end
```

Alternatively, the lack of a {x} export declaration implies {ANY} and is similar to the "public" scoping of other languages.

```
feature -- Constants
```

Finally, scoping can be selectively and precisely controlled to any class in the Eiffel project universe, such as:

```
feature {DECIMAL, DCM_MA_DECIMAL_PARSER, DCM_MA_DECIMAL_HANDLER} -- Access
```

Here, the compiler will allow only the classes listed between the curly braces to access the features within the feature group (e.g. `DECIMAL, DCM_MA_DECIMAL_PARSER, DCM_MA_DECIMAL_HANDLER`).

"Hello, world!"

A programming language's look and feel is often conveyed using a "Hello, world!" program. Such a program written in Eiffel might be:

```
class
    HELLO_WORLD
create
    make
feature
    make
        do
            print ("Hello, world!%N")
        end
end
```

This program contains the class `HELLO_WORLD`. The constructor (create routine) for the class, named `make`, invokes the `print` system library routine to write a `"Hello, world!"` message to the output.

Design by contract

The concept of Design by Contract is central to Eiffel. The contracts assert what must be true before a routine is executed (precondition) and what must hold to be true after the routine finishes (post-condition). Class Invariant contracts define what assertions must hold true both before and after any feature of a class is accessed (both routines and attributes). Moreover, contracts codify into executable code developer and designers assumptions about the operating environment of the features of a class or the class as a whole by means of the invariant.

The Eiffel compiler is designed to include the feature and class contracts in various levels. EiffelStudio, for example, executes all feature and class contracts during execution in the "Workbench mode." When an executable is created, the compiler is instructed by way of the project settings file (e.g. ECF file) to either include or exclude any set of contracts. Thus, an executable file can be compiled to either include or exclude any level of contract, thereby bringing along continuous levels of unit and integration testing. Moreover, contracts can be continually and methodically exercised by way of the Auto-Test feature found in EiffelStudio.

The Design by Contract mechanisms are tightly integrated with the language and guide redefinition of features in inheritance:

- Routine precondition: The precondition may only be weakened by inheritance; any call that meets the requirements of the ancestor meets those of the descendant.
- Routine postcondition: The postcondition can only be strengthened by inheritance; any result guaranteed by the ancestor is still provided by the descendant.
- Class invariant: Conditions that must hold true after the object's creation and after any call to an exported class routine. Because the invariant is checked so often, it makes it simultaneously the most expensive and most powerful form of condition or contract.

In addition, the language supports a "check instruction" (a kind of "assert"), loop invariants, and loop variants (which guarantee loop termination).

Void-safety

Void-safety, like static typing, is another facility for improving software quality. Void-safe software is protected from run time errors caused by calls to void references, and therefore will be more reliable than software in which calls to void targets can occur. The analogy to static typing is a useful one. In fact, void-safe capability could be seen as an extension to the type system, or a step beyond static typing, because the mechanism for ensuring void-safety is integrated into the type system.

The guard against void target calls can be seen by way of the notion of attachment and (by extension) detachment (e.g. detachable keyword). The void-safe facility can be seen in a short re-work of the example code used above:

```
    some_attribute: detachable SOME_TYPE

    use_some_attribute
                -- Set value of some_attribute to `v'.
            do
                if attached some_attribute as l_attribute then
                    do_something (l_attribute)
                end
            end
        do_something (a_value: SOME_TYPE)
                -- Do something with `a_value'.
            do
               ... doing something with `a_value' ...
            end
```

The code example above shows how the compiler can statically address the reliability of whether `some_attribute` will be attached or detached at the point it is used. Notably, the `attached` keyword allows for an "attachment local" (e.g. `l_attribute`), which is scoped to only the block of code enclosed by the if-statement construct. Thus, within this small block of code, the local variable (e.g. `l_attribute`) can be statically guaranteed to be non-void (i.e. void-safe).

Features: commands and queries

The primary characteristic of a class is that it defines a set of features: as a class represents a set of run-time objects, or "instances", a feature is an operation on these objects. There are two kinds of features: queries and commands. A query provides information about an instance. A command modifies an instance.

The command-query distinction is important to the Eiffel method. In particular:

- Uniform-Access Principle: from the point of view of a software client making a call to a class feature, whether a query is an attribute (field value) or a function (computed value) should not make any difference. For example, `a_vehicle.speed` could be an attribute accessed on the object `a_vehicle`, or it could be computed by a function that divides distance by time. The notation is the same in both cases, so that it is easy to change the class's implementation without affecting client software.
- Command-Query Separation Principle: Queries must not modify the instance. This is not a language rule but a methodological principle. So in good Eiffel style, one does not find "get" functions that change something and return a result; instead there are commands (procedures) to change objects, and queries to obtain information about the object, resulting from preceding changes.

Overloading

Eiffel does not allow argument overloading. Each feature name within a class always maps to a specific feature within the class. One name, within one class, means one thing. This design choice helps the readability of classes, by avoiding a cause of ambiguity about which routine will be invoked by a call. It also simplifies the language mechanism; in particular, this is what makes Eiffel's multiple inheritance mechanism possible.[10]

Names can, of course, be reused in different classes. For example, the feature `plus` (along with its infix alias `"+"`) is defined in several classes: `INTEGER`, `REAL`, `STRING`, etc.

Genericity

A generic class is a class that varies by type (e.g. LIST [PHONE], a list of phone numbers; ACCOUNT [G->ACCOUNT_TYPE], allowing for ACCOUNT [SAVINGS] and ACCOUNT [CHECKING], etc.). Classes can be generic, to express that they are parameterized by types. Generic parameters appear in square brackets:

```
class LIST [G] ...
```

G is known as a "formal generic parameter". (Eiffel reserves "argument" for routines, and uses "parameter" only for generic classes.) With such a declaration G represents within the class an arbitrary type; so a function can return a value of type G, and a routine can take an argument of that type:

```
item: G do ... end
put (x: G) do ... end
```

The `LIST [INTEGER]` and `LIST [WORD]` are "generic derivations" of this class. Permitted combinations (with `n: INTEGER`, `w: WORD`, `il: LIST [INTEGER]`, `wl: LIST [WORD]`) are:

```
n := il.item
wl.put (w)
```

`INTEGER` and `WORD` are the "actual generic parameters" in these generic derivations.

It is also possible to have 'constrained' formal parameters, for which the actual parameter must inherit from a given class, the "constraint". For example, in

```
   class HASH_TABLE [G, KEY -> HASHABLE]
```

a derivation `HASH_TABLE [INTEGER, STRING]` is valid only if `STRING` inherits from `HASHABLE` (as it indeed does in typical Eiffel libraries). Within the class, having `KEY` constrained by `HASHABLE` means that for `x: KEY` it is possible to apply to `x` all the features of `HASHABLE`, as in `x.hash_code`.

Inheritance basics

To inherit from one or more others, a class will include an `inherit` clause at the beginning:

```
class C inherit
   A
   B

-- ... Rest of class declaration ...
```

The class may redefine (override) some or all of the inherited features. This must be explicitly announced at the beginning of the class through a `redefine` subclause of the inheritance clause, as in

```
class C inherit
    A
        redefine f, g, h end
    B
        redefine u, v end
```

See[11] for a complete discussion of Eiffel inheritance.

Deferred classes and features

Classes may be defined with `deferred class` rather than with `class` to indicate that the class may not be directly instantiated. Non-instantiatable classes are called abstract classes in some other object-oriented programming languages. In Eiffel parlance, only an "effective" class can be instantiated (it may be a descendant of a deferred class). A feature can also be deferred by using the `deferred` keyword in place of a `do` clause. If a class has any deferred features it must be declared as deferred; however, a class with no deferred features may nonetheless itself be deferred.

Deferred classes play some of the same role as interfaces in languages such as Java, though many object-oriented programming theorists believe interfaces are themselves largely an answer to Java's lack of multiple inheritance (which Eiffel has).[12][13]

Renaming

A class that inherits from one or more others gets all its features, by default under their original names. It may, however, change their names through `rename` clauses. This is required in the case of multiple inheritance if there are name clashes between inherited features; without renaming, the resulting class would violate the no-overloading principle noted above and hence would be invalid.

Tuples

Tuples types may be viewed as a simple form of class, providing only attributes and the corresponding "setter" procedure. A typical tuple type reads

```
TUPLE [name: STRING; weight: REAL; date: DATE]
```

and could be used to describe a simple notion of birth record if a class is not needed. An instance of such a tuple is simply a sequence of values with the given types, given in brackets, such as

```
["Brigitte", 3.5, Last_night]
```

Components of such a tuple can be accessed as if the tuple tags were attributes of a class, for example if `t` has been assigned the above tuple then `t.weight` has value 3.5.

Thanks to the notion of assigner command (see below), dot notation can also be used to assign components of such a tuple, as in

```
t.weight := t.weight + 0.5
```

The tuple tags are optional, so that it is also possible to write a tuple type as `TUPLE [STRING, REAL, DATE]`. (In some compilers this is the only form of tuple, as tags were introduced with the ECMA standard.)

The precise specification of e.g. `TUPLE [A, B, C]` is that it describes sequences of *at least* three elements, the first three being of types `A`, `B`, `C` respectively. As a result, `TUPLE [A, B, C]` conforms to (may be assigned to) `TUPLE [A, B]`, to `TUPLE [A]` and to `TUPLE` (without parameters), the topmost tuple type to which all tuple types conform.

Agents

Eiffel's "agent" mechanism wraps operations into objects. This mechanism can be used for iteration, event-driven programming, and other contexts in which it is useful to pass operations around the program structure. Other programming languages, especially ones that emphasize functional programming, allow a similar pattern using continuations, closures, or generators; Eiffel's agents emphasize the language's object-oriented paradigm, and use a syntax and semantics similar to code blocks in Smalltalk and Ruby.

For example, to execute the `my_action` block for each element of `my_list`, one would write:

```
    my_list.do_all (agent my_action)
```

To execute `my_action` only on elements satisfying `my_condition`, a limitation/filter can be added:

```
    my_list.do_if (agent my_action, agent my_condition)
```

In these examples, `my_action` and `my_condition` are routines. Prefixing them with `agent` yields an object that represents the corresponding routine with all its properties, in particular the ability to be called with the appropriate arguments. So if `a` represents that object (for example because `a` is the argument to `do_all`), the instruction

```
    a.call ([x])
```

will call the original routine with the argument `x`, as if we had directly called the original routine: `my_action (x)`. Arguments to `call` are passed as a tuple, here `[x]`.

It is possible to keep some arguments to an agent **open** and make others **closed**. The open arguments are passed as arguments to `call`: they are provided at the time of *agent use*. The closed arguments are provided at the time of agent *definition*. For example, if `action2` has two arguments, the iteration

```
    my_list.do_all (agent action2 (?, y))
```

iterates `action2 (x, y)` for successive values of `x`, where the second argument remains set to `y`. The question mark `?` indicates an open argument; `y` is a closed argument of the agent. Note that the basic syntax `agent f` is a shorthand for `agent f (?, ?, ...)` with all arguments open. It is also possible to make the *target* of an agent open through the notation `{T}?` where `T` is the type of the target.

The distinction between open and closed operands (operands = arguments + target) corresponds to the distinction between bound and free variables in lambda calculus. An agent expression such as `action2 (?, y)` with some operands closed and some open corresponds to a version of the original operation *curried* on the closed operands.

The agent mechanism also allows defining an agent without reference to an existing routine (such as `my_action`, `my_condition`, `action2`), through inline agents as in

```
my_list.do_all (agent (s: STRING)
     require
         not_void: s /= Void
     do
         s.append_character (',')
     ensure
         appended: s.count = old s.count + 1
     end)
```

The inline agent passed here can have all the trappings of a normal routine, including precondition, postcondition, rescue clause (not used here), and a full signature. This avoids defining routines when all that's needed is a computation to be wrapped in an agent. This is useful in particular for contracts, as in an invariant clause that expresses that all elements of a list are positive:

```
    my_list.for_all (agent (x: INTEGER): BOOLEAN do Result := (x > 0) end)
```

The current agent mechanism leaves a possibility of run-time type error (if a routine with *n* arguments is passed to an agent expecting *m* arguments with $m < n$). This can be avoided by a run-time check through the precondition `valid_arguments` of `call`. Several proposals for a purely static correction of this problem are available, including a language change proposal by Ribet et al.[14]

Once routines

A routine's result can be cached using the `once` keyword in place of `do`. Non-first calls to a routine require no additional computation or resource allocation, but simply return a previously computed result. A common pattern for "once functions" is to provide shared objects; the first call will create the object, subsequent ones will return the reference to that object. The typical scheme is:

```
shared_object: SOME_TYPE
    once
        create Result.make (args)
             -- This creates the object and returns a reference to it through `Result'.
    end
```

The returned object—`Result` in the example—can itself be mutable, but its reference remains the same.

Often "once routines" perform a required initialization: multiple calls to a library can include a call to the initialization procedure, but only the first such call will perform the required actions. Using this pattern initialization can be decentralized, avoiding the need for a special initialization module. "Once routines" are similar in purpose and effect to the singleton pattern in many programming languages, and to the Borg pattern used in Python.

By default, a "once routine" is called *once per thread*. The semantics can be adjusted to *once per process* or *once per object* by qualifying it with a "once key", e.g. `once ("PROCESS")`.

Conversions

Eiffel provides a mechanism to allow conversions between various types. The mechanisms coexists with inheritance and complements it. To avoid any confusion between the two mechanisms, the design enforces the following principle:

(Conversion principle) A type may not both conform and convert to another.

For example, `NEWSPAPER` may conform to `PUBLICATION`, but `INTEGER` converts to `REAL` (and does not inherit from it).

The conversion mechanism simply generalizes the ad hoc conversion rules (such as indeed between `INTEGER` and `REAL`) that exist in most programming languages, making them applicable to any type as long as the above principle is observed. For example, a `DATE` class may be declared to convert to `STRING`; this makes it possible to create a string from a date simply through

```
    my_string := my_date
```

as a shortcut for using an explicit object creation with a conversion procedure:

```
    create my_string.make_from_date (my_date)
```

To make the first form possible as a synonym for the second, it suffices to list the creation procedure (constructor) `make_from_date` in a `convert` clause at the beginning of the class.

As another example, if there is such a conversion procedure listed from `TUPLE [day: INTEGER; month: STRING; year: INTEGER]`, then one can directly assign a tuple to a date, causing the appropriate conversion, as in

```
        Bastille_day := [14, "July", 1789]
```

Exception handling

Exception handling in Eiffel is based on the principles of design by contract. For example, an exception occurs when a routine's caller fails to satisfy a precondition, or when a routine cannot ensure a promised postcondition. In Eiffel, exception handling is not used for control flow or to correct data-input mistakes.

An Eiffel exception handler is defined using the `rescue` keyword. Within the `rescue` section, the `retry` keyword executes the routine again. For example, the following routine tracks the number of attempts at executing the routine, and only retries a certain number of times:

```
connect_to_server (server: SOCKET)
      -- Connect to a server or give up after 10 attempts.
    require
        server /= Void and then server.address /= Void
    local
        attempts: INTEGER
    do
        server.connect
    ensure
      connected: server.is_connected
    rescue
        if attempts < 10 then
            attempts := attempts + 1
            retry
        end
    end
```

This example is arguably flawed for anything but the simplest programs, however, because connection failure is to be expected. For most programs a routine name like `attempt_connecting_to_server` would be better, and the postcondition would not promise a connection, leaving it up to the caller to take appropriate steps if the connection was not opened.

Concurrency

A number of networking and threading libraries are available, such as EiffelNet and EiffelThreads. A concurrency model for Eiffel, based on the concepts of design by contract, is SCOOP, or *Simple Concurrent Object-Oriented Programming*, not yet part of the official language definition but available in EiffelStudio.

CAMEO[15] is an (unimplemented) variation of SCOOP for Eiffel. Concurrency also interacts with exceptions. Asynchronous exceptions can be troublesome (where a routine raises an exception after its caller has itself finished).[16]

Operator and bracket syntax, assigner commands

Eiffel's view of computation is completely object-oriented in the sense that every operation is relative to an object, the "target". So for example an addition such as

```
a + b
```

is conceptually understood as if it were the method call

```
a.plus (b)
```

with target `a`, feature `plus` and argument `b`.

Of course, the former is the conventional syntax and usually preferred. Operator syntax makes it possible to use either form by declaring the feature (for example in `INTEGER`, but this applies to other basic classes and can be used in any other for which such an operator is appropriate):

```
plus alias "+" (other: INTEGER): INTEGER
        -- ... Normal function declaration...
    end
```

The range of operators that can be used as "alias" is quite broad; they include predefined operators such as "+" but also "free operators" made of non-alphanumeric symbols. This makes it possible to design special infix and prefix notations, for example in mathematics and physics applications.

Every class may in addition have *one* function aliased to "[]", the "bracket" operator, allowing the notation `a [i, ...]` as a synonym for `a.f (i, ...)` where `f` is the chosen function. This is particularly useful for container structures such as arrays, hash tables, lists etc. For example, access to an element of a hash table with string keys can be written

```
   number := phone_book ["JILL SMITH"]
```

"Assigner commands" are a companion mechanism designed in the same spirit of allowing well-established, convenient notation reinterpreted in the framework of object-oriented programming. Assigner commands allow assignment-like syntax to call "setter" procedures. An assignment proper can never be of the form `a.x := v` as this violates information hiding; you have to go for a setter command (procedure). For example, the hash table class can have the function and the procedure

```
item alias "[]" (key: STRING): ELEMENT         [3]
      -- The element of key `key'.
      -- ("Getter" query)
    do
        ...
    end

put (e: ELEMENT; key: STRING)
      -- Insert the element `e', associating it with the key `key'.
      -- ("Setter" command)
    do
```

```
        ...
    end
```

Then to insert an element you have to use an explicit call to the setter command:

```
    [4] phone_book.put (New_person, "JILL SMITH")
```

It is possible to write this equivalently as

```
    [5] phone_book ["JILL SMITH"] := New_person
```

(in the same way that `phone_book ["JILL SMITH"]` is a synonym for `number := phone_book.item ("JILL SMITH")`), provided the declaration of `item` now starts (replacement for [3]) with

```
    item alias "[]" (key: STRING): ELEMENT assign put
```

This declares `put` as the assigner command associated with `item` and, combined with the bracket alias, makes [5] legal and equivalent to [4]. (It could also be written, without taking advantage of the bracket, as `phone_book.item ("JILL SMITH") := New_person`.

note: The argument list of a's assigner is constrained to be: (a's return type;all of a's argument list...)

Lexical and syntax properties

Eiffel is not case-sensitive. The tokens `make`, `maKe` and `MAKE` all denote the same identifier. See, however, the "style rules" below.

Comments are introduced by `--` (two consecutive dashes) and extend to the end of the line.

The semicolon, as instruction separator, is optional. Most of the time the semicolon is just omitted, except to separate multiple instructions on a line. This results in less clutter on the program page.

There is no nesting of feature and class declarations. As a result, the structure of an Eiffel class is simple: some class-level clauses (inheritance, invariant) and a succession of feature declarations, all at the same level.

It is customary to group features into separate "feature clauses" for more readability, with a standard set of basic feature tags appearing in a standard order, for example:

```
class HASH_TABLE [ELEMENT, KEY -> HASHABLE] inherit TABLE [ELEMENT]

    feature -- Initialization
         -- ... Declarations of initialization commands (creation procedures/constructors) ...

    feature -- Access
         -- ... Declarations of non-boolean queries on the object state, e.g. item ...

    feature -- Status report
         -- ... Declarations of boolean queries on the object state, e.g. is_empty ...

    feature -- Element change
         -- ... Declarations of commands that change the structure, e.g. put ...

    -- etc.
end
```

In contrast to most curly bracket programming languages, Eiffel makes a clear distinction between expressions and instructions. This is in line with the Command-Query Separation principle of the Eiffel method.

Style conventions

Much of the documentation of Eiffel uses distinctive style conventions, designed to enforce a consistent look-and-feel. Some of these conventions apply to the code format itself, and others to the standard typographic rendering of Eiffel code in formats and publications where these conventions are possible.

While the language is case-insensitive, the style standards prescribe the use of all-capitals for class names (`LIST`), all-lower-case for feature names (`make`), and initial capitals for constants (`Avogadro`). The recommended style also suggests underscore to separate components of a multi-word identifier, as in `average_temperature`.

The specification of Eiffel includes guidelines for displaying software texts in typeset formats: keywords in bold, user-defined identifiers and constants are shown in *`italics`*, comments, operators, and punctuation marks in `Roman`, with program text in `blue` as in the present article to distinguish it from explanatory text. For example, the "Hello, world!" program given above would be rendered as below in Eiffel documentation:

```
class
    HELLO_WORLD
create
    make
feature
    make
       do
          print ("Hello, world!")
       end
end
```

Interfaces to other tools and languages

Eiffel is a purely object-oriented language but provides an open architecture for interfacing with "external" software in any other programming language.

It is possible for example to program machine- and operating-system level operations in C. Eiffel provides a straightforward interface to C routines, including support for "inline C" (writing the body of an Eiffel routine in C, typically for short machine-level operations).

Although there is no direct connection between Eiffel and C, many Eiffel compilers (Visual Eiffel is one exception) output C source code as an intermediate language, to submit to a C compiler, for optimizing and portability. As such, they are examples of transcompilers. The Eiffel Compiler tecomp can execute Eiffel code directly (like an interpreter) without going via an intermediate C code or emit C code which will be passed to a C compiler in order to obtain optimized native code. On .NET, the EiffelStudio compiler directly generates CIL (Common Intermediate Language) code. The SmartEiffel compiler can also output Java bytecode.

References

1. "Eiffel in a Nutshell" (https://archive.eiffel.com/eiffel/nutshell.html). *archive.eiffel.com*. Retrieved 24 August 2017.
2. "EiffelStudio 19.05 is now available!" (https://www.eiffel.org/blog/eiffelstudio_19_05). *Eiffel.org*.
3. "EiffelStudio 20.05 Releases" (https://dev.eiffel.com/EiffelStudio_20.05_Releases). *Eiffel.org*.

4. Cooper, Peter (2009). *Beginning Ruby: From Novice to Professional*. Beginning from Novice to Professional (2nd ed.). Berkeley: APress. p. 101. ISBN 978-1-4302-2363-4. "To a lesser extent, Python, LISP, Eiffel, Ada, and C++ have also influenced Ruby."
5. "Eiffel — the Language" (http://www.berenddeboer.net/eiffel/archive/halstenbach_eiffel_history.html). Retrieved 6 July 2016.
6. Formal Specification Languages: Eiffel, Denotational Semantics, Vienna Development Method, Abstract Machine Notation, Petri Net, General Books, 2010
7. *Object-Oriented Software Construction*, Second Edition, by Bertrand Meyer, Prentice Hall, 1997, ISBN 0-13-629155-4
8. ECMA International: Standard ECMA-367 —Eiffel: Analysis, Design and Programming Language 2nd edition (June 2006); available online at www.ecma-international.org/publications/standards/Ecma-367.htm (http://www.ecma-international.org/publications/standards/Ecma-367.htm)
9. International Organisation for Standardisation: Standard ISO/IEC DIS 25436, available online at [1] (http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=42924&ICS1=35&ICS2=60&ICS3=)
10. Bertrand Meyer: Overloading vs Object Technology, in Journal of Object-Oriented Programming (JOOP), vol. 14, no. 4, October–November 2001, available online (http://se.ethz.ch/~meyer/publications/joop/overloading.pdf)
11. "9 INHERITANCE" (http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial-10.html). Archive.eiffel.com. 1997-03-23. Retrieved 2013-07-08.
12. "Multiple Inheritance and Interfaces" (http://www.artima.com/intv/abcs.html). Artima.com. 2002-12-16. Retrieved 2013-07-08.
13. "Multiple Inheritance Is Not Evil" (https://c2.com/cgi/wiki?MultipleInheritanceIsNotEvil). C2.com. 2007-04-28. Retrieved 2013-07-08.
14. Philippe Ribet, Cyril Adrian, Olivier Zendra, Dominique Colnet: *Conformance of agents in the Eiffel language*, in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pp. 125-143. Available on line from the JOT article page (http://www.jot.fm/issues/issue_2004_04/article7)
15. Brooke, Phillip; Richard Paige (2008). "Cameo: An Alternative Model of Concurrency for Eiffel" (https://hal.archives-ouvertes.fr/hal-00534917/file/PEER_stage2_10.1007%252Fs00165-008-0096-1.pdf) (PDF). *Formal Aspects of Computing*. Springer. **21** (4): 363–391. doi:10.1007/s00165-008-0096-1 (https://doi.org/10.1007%2Fs00165-008-0096-1).
16. Brooke, Phillip; Richard Paige (2007). "Exceptions in Concurrent Eiffel" (http://www.jot.fm/issues/issue_2007_11/article4/). *Journal of Object Technology*. **6** (10): 111–126. doi:10.5381/jot.2007.6.10.a4 (https://doi.org/10.5381%2Fjot.2007.6.10.a4).

External links

- Eiffel Software (http://www.eiffel.com/) web site of the company that introduced Eiffel, was Interactive Software Engineering (ISE).