This is a pre-publication version of an article that appeared in the JOOP Eiffel column in 1999. Citation reference: Bertrand Meyer, *Extension Season*, in *Journal of Object-Oriented Programming* (JOOP), vol. 12, no. 3, June 1999.

Extension season

Bertrand Meyer

For the first time in many years, Eiffel is undergoing significant changes, all meant to increase expressibility without renouncing simplicity and safety.

Eiffel, more than a language, is a method of building software, and previous articles in this column have accordingly concentrated on principles and applications of the method rather than on the language itself. Language issues are important too, so this column and the next one turn their attention to recent changes. For the first time in many years, the Eiffel language is indeed undergoing some significant extensions. They have been discussed extensively on the Internet and within the Nonprofit International Consortium for Eiffel (NICE); but this will be their first description in print, as befits the JOOP Eiffel column.

Although the discussion is more language-specific than in previous, the idea is, as usual, to have something for everyone, Eiffel programmers as well as others. So even if you are programming in another language and have only a superficial knowledge of Eiffel I hope you will find this column useful, because the goals addressed by recent changes are not just Eiffel goals: they are software design issues, and I am sure you will be able to relate them to your language of choice.

Eiffel programmers are not, by instinct, favorably disposed towards language extensions. They chose Eiffel because they like it just the way it is, and when anyone — original language designer included — wants to add or change something, their first reaction usually is: "hands off!". Partly for that reason, Eiffel has enjoyed great stability, as a language as well as a method, since its original design in 1985. During that period most other O-O languages have changed significantly, sometimes dramatically; but Eiffel incurred only one major revision, leading in the years 1990-1991 to the book *Eiffel: The Language* [1] or "ETL", which describes the language level known as Eiffel 3. This is the basis for all current implementations. The only significant addition since then has been the *Precursor* mechanism, the result of a collective design in 1997-1998, facilitating the redefinition of a routine in terms of its original implementation.

In the software world nothing can remain completely static, and the time has come to take a closer look at the language. A number of modifications premiered recently with release of ISE Eiffel 4.3. They all try to follow the principles discussed in detail in the chapter *On Language Evolution* of ETL:

- The general idea is to increase the signal-to-noise ratio of the language: improve the signal (the expressive power available to language users) and decrease the noise (the amount of redundant or useless features).
- When thinking of changes, consider not only adding constructs but also removing superfluous constructs and cleaning up existing ones for simplicity and consistency.
- Every extension must address clear user needs.
- Every extension must be compatible with the spirit and letter of the language.
- Every extension must be easy to explain to any competent user.
- Every extension must have been carefully discussed by users and implementers; the syntax, validity constraints and semantics should be available in the precise form used in ETL.
- Extensions must favor safety and, in particular, support Eiffel's static typing policy and its Design by Contract principles.
- Don't shoot for upward compatibility at all costs. Be considerate to existing users, of course, but you find a better way and it really makes a difference, go for it. Users will follow, as long as you explain the rationale convincingly and provide a migration path. They don't want you to sacrifice consistency and simplicity for the sake of compatibility. There are too many examples around us of languages that are just made of stratified layers of successive sediments.
- The good extensions are those which address *several* issues rather than just one.
- Extensions should not violate the Eiffel principle that for every need there should be *one* good way to express the solution, rather than many competing mechanisms.
- No extension must be accepted definitively until an official compiler release has supported it for several months.

The extensions described here satisfy, I hope, these criteria. They are all part of 4.3, and have already been used in a number of large applications (including by us). Their final incorporation is of course subject further discussion by all involved.

The extensions discussed below are:

- New creation syntax
- Creation expressions
- Generic creation (as you see, the area of object creation has been particularly active)
- Recursive generic constraints
- Tuples
- Generic conformance

This list does not include what is clearly the most spectacular innovation, the *agent* mechanism, which pursues many different goals at once: higher-level operators (functions that manipulate functions that…), iterators, introspection ("reflection"), mechanisms for numerical computation, higher-level contracts and assertions. Agents are important enough to justify a separate article, which will be the next one in this column,

co-signed with the other developers of the mechanism. If you can't wait, take a peek at its description at http://eiffel.com, where you will also find further information about the extensions described below.

New creation syntax

We start with a purely syntactic change; nothing earth-shattering but an improvement of the syntax's clarity. It has always been the goal with Eiffel that even someone who doesn't know the language should be able to grasp immediately the essentials of a class text. Eiffel is often used for analysis and design, so you may have to show Eiffel texts to people who are not experts in the technology. An exception to this rule of "immediate understanding of basic constructs" was introduced by Eiffel 3 (the original version was better in this respect) with its creation syntax, employing exclamation marks. This convention is simple and easy to remember, but it has to be learned. The new variant is more in line with the keyword-oriented style of the rest of the language:

create *x*	-- No creation procedure, implicit type
create *x.make* (*a*, *b*)	-- Initialize object with *make*, with given arguments
create {*T*} *x*	-- Use *T* as the type of the new object
create {*T*} *x.make* (*a*, *b*)	-- Use *T* as type, initialize with *make*

The creation procedure *make* ("constructor" in C++ terminology) must be one of the procedures listed in the **create** (ex **creation** — there is only one keyword left, for ease of learning and remembering) clause at the beginning of the class. In the forms with an explicity type *T*, that type must be a descendant of the type declared for *T*.

For this and subsequent comments on creation mechanisms, you need to understand the role of creation procedures, which the Eiffel method views are more than just a convenience to initialize objects at the time of creation, overriding the language's default initializations. In the Design by Contract approach, creation procedures fulfill a clear role: making sure that every object starts its life in a state satisfying the invariant. That's why you must declare creation procedures explicitly: you should make sure that each of them, starting from the default initializations (zero for numbers, false for booleans and so on) will ensure the invariant.

Creation expressions

The notion of creation expression, well known in other languages (Simula already had it), is new in Eiffel, because so far we have been content with creation instructions of the form shown above. Indeed, in most cases you will still be using creation instructions. But sometimes you want to create an object for the sole purpose of passing it immediately as argument to a routine. Then instead of writing something like

x: *YOUR_TYPE*

…

create *x*

your_routine (*x*)

with its need for a local entity *x* and a separate creation instruction, creation expressions allow you to write just

your_routine (**create** {*YOUR_TYPE*})

or, if you need a creation procedure *make*:

your_routine (**create** {*YOUR_TYPE*}.*make* (…))

We found out that some users had hundreds of such cases, so the simplification is nothing to be scoffed at. Note that this mechanism would seem to violate the principle of "only one good way to do anything", since instead of **create** *x* you can now also write *x* := **create** {*TYPE_OF_x*}, but in fact the principle stands intact since the instruction (the first) form is preferable to the expression form, if only because in most cases you don't need to specify the type, which the expression form requires you to do. The expression form appears useful only in the case discussed above (creating an object as argument to a routine), where the instruction form is clumsy. So there is no redundancy between the two forms; rather, a clear separation of roles.

Generic creation

The next issue is more delicate: how to create objects of a generic type (template class, in C++ terms). Assume you have a class *C* [*G*], parameterized by *G* which represents an arbitrary type, and, within the class, an entity (variable) *x: G*. Now in some routine of *C* you want to create a new instance of *G* and attach it to *x*. How do you do it?

So far, it was impossible, and ETL explains in detail why: because *G*, by nature, represents an arbitrary type, you don't know what creation procedures it may accept! Recall that initialization by a creation procedure is not just a matter of convenience, but a matter of ensuring consistent objects; if we ever got objects that don't satisfy their invariants, we would be unable to reason about our software. (C++ and Java programmers should also note that Eiffel has none of the strange rules about constructor inheritance: each class is, simply and naturally, free to determine how to initialize its own instances, using techniques that are often different from those of its parents.)

ETL also gives some workarounds to create and initialize generic objects, but they are rather heavy. Fortunately, a clever idea (due to Mark Howard of Axa Rosenberg, formerly Rosenberg Institutional Equity Management) solves the problem in a really neat way. Mark's solution involves a variant of genericity known as **constrained** genericity.

Instead of *C* [*G*], you can declare a generic class as *C* [*G* –> *CT*] where *CT* is a type, the "constraining type" for *G*. What this means is that a generic derivation *C* [*SOME_TYPE*] (a "template instantiation" in C++ terms, but we prefer to reserve the word "instantiation" for the run-time creation of objects, instances of a class) is only legal if *SOME_TYPE* conforms, in the sense of inheritance, to *CT*. For example Free EiffelBase [2] contains a class *HASH_TABLE* [*G, H* –> *HASHABLE*] representing hash tables (dictionaries) containing objects of an arbitrary type *G*, identified by keys of an almost arbitrary type *H*. Almost, because types corresponding to *H* must conform to *HASHABLE*, requiring them to have a hash function. *HASHABLE* is a deferred (abstract) class with a few features including *hash_code*. This enables class *HASH_TABLE* to apply these features to anything of type *H*, and hence to provide a proper, type-safe implementation of hash tables. *INTEGER*, *STRING* and the like conform to *HASHABLE*, and you can achieve the same for any of your classes by making it inherit from *HASHABLE* (this is a multiple inheritance world, of course) and providing an implementation of *hash_code*.

How does constrained genericity address creation of generics? The idea is to permit a declaration of the form *C* [*G* –> *CT* **create** *make, ...* **end**] with the rule that the names listed, here *make* etc., must be procedures of *CT*, although not necessarily *creation* procedures of that class. Then for the generic derivation *C* [*SOME_TYPE*] to be legal you must use a *SOME_TYPE* that is not only a descendant of *CT*, as before, but also such that its versions of *make* and co. are creation procedures in *CT* (i.e. listed in its **create** clause at the beginning of the class). Then it is permitted in the body of *C*, for *x* of type *G*, to use creation instructions or expressions on *x*, with one of the listed creation procedures.

Part of the beauty of the scheme is that because we only require the listed names to denote procedures of *CT*, not creation procedures at that level, *CT* may be a deferred class. So a whole category of elegant applications opens up.

Recursive generic constraints

It was not explicitly possible, previously, to write a generically constrained class

C [*G* –> *CT, H* –> *ARRAY* [*G*]]

Which makes legal such derivations as *C* [*INTEGER*, *ARRAY* [*INTEGER*]]. This is now permitted through careful rewording of the corresponding validity rules. You can even write something like *C* [*G* –> *H, H* –>*G*], which means that the only legal generic derivations are of the form *C* [*A, A*] for arbitrary *A*. This last example doesn't seem very useful, but the first one is typical of a practical scheme that may often be convenient. You can see how Eiffel addresses the needs of a truly typed object-oriented language, with the powerful combination of resulting mechanisms, involving both genericity (parametric polymorphism, as it's sometimes called) and inheritance with its many facets.

Generic conformance

The next change is not a language change at all, but in fact the correction of a difficult but. The language specification states that *C* [*U*], for a generic class *C*, conforms to *C* [*T*]

if and only if U conforms to T. In previous release, this was not properly enforced for reference types (the only major language rule not yet implemented). 4.3 now has the rule fully implemented. To the non-expert this will seem obvious, but the correction actually represented a major architectural effort.

As sometimes happens in such circumstances, we know that some systems took "advantage" of the bug and will have to be correspondingly adapted. But the correct type rule must of course be supported.

Tuples

The last change to be discussed here is a notion of anonymous class, or tuple. A tuple is a sequence of values of arbitrary (although specified) types. You can write a tuple type as

TUPLE [*X, Y, ...*]

For arbitrary types *X, Y, ...*This looks like a generically derived type, but the number of parameters is arbitrary. To obtain a generalized notion of generic type with an arbitrary number of parameters for a type other than *TUPLE*, use constrained genericity, as in *VARTYPE* [*G –> TUPLE*], which you can then use as *VARTYPE* [*TUPLE*], *VARTYPE* [*TUPLE* [*INTEGER*]], *VARTYPE* [*TUPLE* [*INTEGER, REAL*]] and so on.

A tuple expression may be written simply as [*value1*, *value2*, …], with appropriate types for the successive *values*. This replaces the previous (and still supported for the moment) "manifest arrays", and gives them a proper type; manifest arrays were the only kind of expression whose type status was a bit murky.

Tuples give us an incredible array of facilities. They enable us to manipulate sequences of values; to define functions with multiple results without writing superfluous classes; and to define agents (the topic of the next article) in a type-safe way. I should add that there is actually more to the tuple mechanism than described here, but what I have presented is what has been implemented so far as part of 4.3, and I will continue to abide with the principle of not talking about any change until there is a working implementation. None of the extra facilities invalidates what has been said so far.

As you can see, Eiffel design work is alive and even vibrant. All the ideas presented here have been thoroughly discussed and tested; only a tiny number of the proposed changes survive the process. The result will continue to support the Eiffel principles of simplicity, clarity and safety, while offering Eiffel developers an ever increased power of expression.

References

[1] *Eiffel: The Language*, Bertrand Meyer, Prentice Hall, 1991 (second printing).

[2] *EiffelBase goes public*, Bertrand Meyer, JOOP, November 1998 See also http://eiffel.com/products/base.