19

Objects, values and entities

19.1 OVERVIEW

The execution of an Eiffel system consists of creating, accessing and modifying **objects**.

The following presentation discusses the structure of objects and how they relate to the syntactical constructs that denote objects in software texts: **expressions**. At run time, an expression may take on various *values*; every value is either an object or a reference to an object.

Among expressions, **entities** play a particular role. An entity is an identifier (name in the software text), meant at execution time to denote possible values. Some entities are **read-only**: the execution can't change their initial value. Others, called **variables**, can take on successive values during execution as a result of such operations as creation and assignment.

The description of objects and their properties introduces the *dynamic model* of Eiffel software execution: the run-time structures of the data manipulated by an Eiffel system.

This chapter and the following one will illustrate the dynamic model through figures representing values and objects. These figures and the conventions only serve explanatory purposes. In particular:

- Although they may suggest the actual implementation techniques used to represent values and objects at run time, they should not be construed as *prescribing* any specific implementation.
- Do not confuse these conventions for representing *dynamic* (that is to say, run-time) properties of systems with the graphical conventions for representing classes, features, the client relation, inheritance, and other *static* properties of software texts.

We saw that it is often convenient, in these representations of the static model, to picture *attribute* features in a form that resembles the representation of objects in the dynamic model. But this should cause no ambiguity since one convention applies to classes and the other to run-time objects.

← *"GRAPHICAL REPRESENTATION", 18.2, page 491*.

19.2 OBJECTS AND THEIR TYPES

During its execution, an Eiffel system will create one or more objects.

There will always be at least one: the root object created on execution start.

← "System execution", page 114

A clear correspondence exists between objects, the dynamic (run-time) notion, and on the other side types and classes, the static (programming-time) notions. Every object proceeds from a type, itself based on a class. The following definitions capture this correspondence:

Type, generating type of an object; generator

Every run-time object is a direct instance of exactly one Class_or_tuple_type of the system, called the **generating type** of the object, or just "the type of the object" if there is no ambiguity.

The base class of the generating type is called the object's **generating class**, or **generator** for short.

← "Direct instance" was defined on page 321.

An object may be an *instance* of many types: if it is an instance of *TC*, it is also an instance of any type *TB* to which *TC* conforms. But it is a *direct* instance of only one type, and so has just one generating type.

← "Instance of a type", page 322.

To obtain the generating type of the object attached to *x*, you may use:

```
x.type
```

whose value is an object denoting a type. The query *type*, which comes from the universal class *ANY*, returns an object denoting a type, with the associated feature; with *x* declared of type *TX*, the type of *x*.*type* itself is

→ "OBJECT PROPERTIES", 35.4, page 919

```
TYPE [TX]
```

based on the library class *TYPE*. More precisely, *TYPE* [*TX*] covers all objects representing types that *conform* to *TX*, including *TX* itself.

19.3 VALUES AND INSTANCES

We saw in the discussion of types that any possible value for an entity is either an object or a *reference*. The notion of reference has a precise definition:

← "Direct instances and values of a type", page 321.

Reference, void, attached, attached to

A **reference** is a value that is either:

- **Void**, in which case it provides no more information.
- **Attached**, in which case it gives access to an object. The reference is said to be **attached to** that object, and the object attached to the reference.

A non-void reference is "attached to" exactly one object, but an object may be attached to several references.

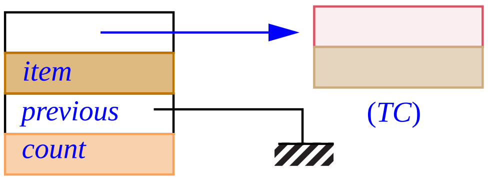
The reserved word *Void* denotes a void reference. To find out if the value of *e* is void, use the boolean expression

See chapter 35 about the features of class ANY. Void may be implemented as an attribute or a once function.

```
e = Void
```

Values of an *expanded* type can never be void.

The following figure shows conventions for representing a reference: by an arrow — more precisely, a blue arrow in this book — if attached to an object, by a special "grounding" symbol if void. Below an object you may write its generating type, here *TC*.



Picturing references, attached and void

The four values on the figure are the fields of the object on the left. The first and the third value from the top, labeled *next* and *previous*, are references; *next* is attached to the object on the right, and *previous* is void. The figure gives no information about the values in the expanded fields *item* and *count*, or about the fields of the object on the right.

The following property is essential to the consistency of the Eiffel type system and the dynamic model:

← Following directly from the "Instance principle", page 323.

Object principle

Every non-void value is either an object or a reference attached to an object.

In particular, simple values such as integers, booleans and reals are objects.

19.4 BASIC TYPES

A number of object types come from classes of the Kernel Library: *BOOLEAN*; *CHARACTER* (64-bit) and *CHARACTER_8*; *INTEGER* and its sized variants *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*,*NATURAL_64*; *REAL* and its sized variants *REAL_32* and *REAL_64*; and *POINTER*.

The specification of their direct instances — boolean values, characters, integers, floating-point numbers, and addresses for passing to external software — appears in the chapter on basic types.

→ *Chapter 30.*

The specifications of direct instances appearing in the rest of the present chapter exclude the case of basic types.

19.5 REFERENCE AND COPY SEMANTICS

Object semantics

Every run-time object has either **copy semantics** or **reference semantics**.

An object has copy semantics if and only if it is the result of executing a creation operation whose creation target is of an expanded type, or of cloning such an object.

This property determines the role of the object when used as source of an assignment: with copy semantics, it will be copied onto the target; with reference semantics, a reference will be reattached to it.

19.6 COMPOSITE OBJECTS AND THEIR FIELDS

We will use specific terminology for non-basic types:

Non-basic class, non-basic type, field

Any class other than the basic types is said to be a **non-basic class**. Any type whose base class is non-basic is a **non-basic type**, and its instances are **non-basic objects**.

A direct instance of a non-basic type is a sequence of zero or more values, called **fields**. There is one field for every attribute of the type's base class.

This definition makes no difference between variable and constant attributes. See the end of this section.

Consider a class type *TC*, of base class *C*, and an attribute *a* of class *C*; let *TA* be the type of *a*. The possible values for the field corresponding to attribute *a* in a direct instance of *TC* depend on the nature of *TA*. There are three possible cases for *TA*:

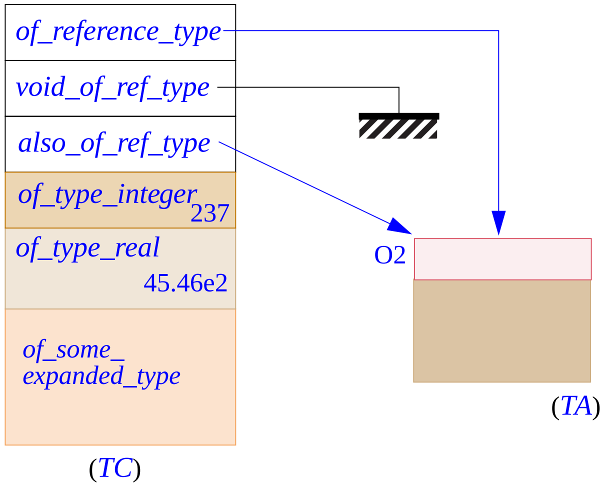
1 • Reference type.

2 • Expanded type.

3 • Formal generic parameter of class *C*.

In case 1, the field corresponding to attribute *a* is a reference. That reference may be void, or it may be attached to an instance of *TA*'s base type — not necessarily a direct instance. In the figure on the following page, the first and third fields from the top are attached to the same object, called O2.

An object and its fields



This represents a partial snapshot taken during the execution of a possible system, illustrating some of the various kinds of field.

In case 2, the field corresponding to attribute *a* is an instance of the expanded type *TA*. That field, then, is itself an object, called a **subobject** of the enclosing object. There are two cases:

- *TA* may be a basic type; then the subobject is a basic object of that type; the figure shows fields of type *INTEGER* and *REAL*.
- If *TA* is a non-basic expanded type, the subobject is itself a non-basic object. This applies to the last field of the left object on the figure. In this case the enclosing object isa **composite** object.

Subobject, composite object

Any expanded field of an object is a **subobject** of that object.

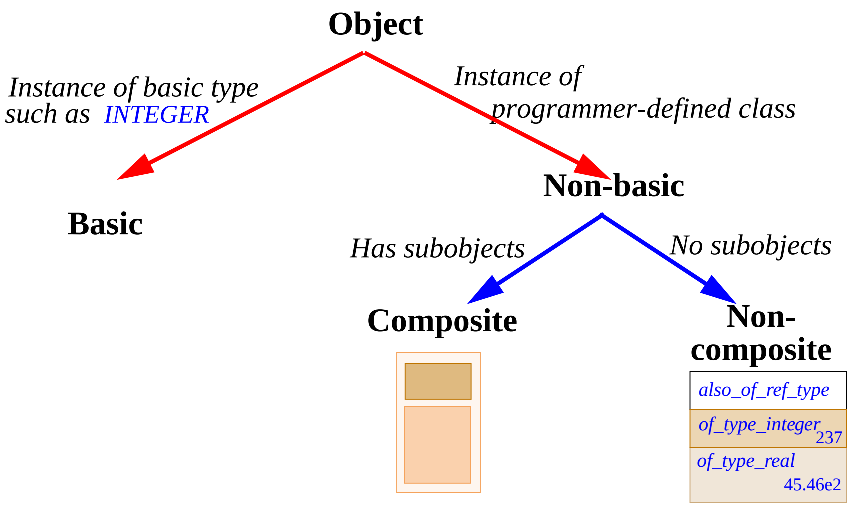
An object that has a non-basic subobject is said to be **composite**.

Finally, in case 3, *TA* is a formal generic parameter of class *C*, the base class of *TC*. Depending on whether the actual generic parameter is a reference type or an expanded type, this will in fact yield either case 1 or case 2.

The above definition of fields makes no difference between constant and variable attributes: an attribute of either kind yields a field in every instance. In a reasonable implementation, fields for constant attributes, being the same value for every instance of a class, will not occupy any run-time space. This indicates again that figures representing objects (such as the ones in this chapter) do not necessary show actual object implementations. This book uses "field" in the precise sense defined above, which does not always imply an actual memory area in an object's representation.

Here is a summary of the classification of objects:

Kinds of object

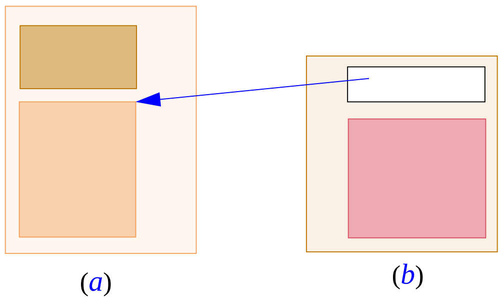


19.7 REFERENCE ATOMICITY

The dynamic model as illustrated above has both composite objects, containing subobjects, and references to objects. How do these notions combine? In particular, can a system produce the run-time situation shown on the following figure, where a reference is attached to a subobject of another object?

Reference to subobject

WARNING: This illustrates an impossible sit-uation.



The answer is no. The semantics of reattachment operations (Assignment, formal-actual argument association) will guarantee that a reference can only become attached to a full object. Although objects themselves are not "atomic", since clients can modify individual fields by calling the appropriate routines, the level of atomicity for attaching *references* is an entire object.

→ See case [2] of attachment semantics table on page 590, and the discussion that follows.

It is possible to conceive of a model that supports references to subobjects, as was in fact the case in ISE Eiffel 2. But this significantly complicates the dynamic model and the implementation, garbage collection in particular, without bringing a clearly useful improvement in expressive power.

19.8 EXPRESSIONS AND ENTITIES

The discussion so far has defined the object structures that can be created during system execution. To denote the objects and their fields in software texts, you may use expressions — specimens of the construct Expression.

There are several forms of expression, which subsequent chapters cover in detail. One form, the simplest, is of immediate interest: entities, which consist of a single name.

Entity, variable, read-only

An **entity** is an Identifier, or one of two reserved words (**Current** and **Result**), used in one of the following roles:

1 • Final name of an attribute of a class.

2 • Local variable of a routine or Inline_agent, including **Result** for a query.

3 • Formal argument of a routine or inline agent.

4 • Object Test local.

5 • **Current**, the predefined entity used to represent a reference to the current object (the target of the latest not yet completed routine call).

Names of non-constant attributes and local variables are **variable** entities, also called just **variables**. Constant attributes, formal arguments, Object Test locals and **Current** are **read-only** entities.

→ See 8.6, page 221, about local variables and Result. Inline agents are an described in chapter 27 and Object Test locals in 24.3, page 650.

Two kinds of operation, creation and reattachment, may modify the value of a variable (a non-constant attribute, part of category 1, or local variable, category 2. In the other four cases — constant attributes, formal arguments (3), Object Test locals (4) and **Current** (5) — you may not directly modify the entities, hence the name *read-only* entity.

→ Creation: chapter 20; reattachment: chapter 22.

The term "*constant* entity" wouldn't do, not so much because you can modify the corresponding objects but because read-only entities (other than constant attributes) do change at run time: a qualified call reattaches **Current**, and any routine call reattaches the formal arguments.

→ "Current object, current routine", page 641

Result appearing in the Postcondition of a constant attribute cannot be changed at execution time, but for simplicity is considered part of local variables in all cases anyway.

Here is the corresponding syntax specification:

Entities and variables

Entity ≜ Variable | Read_only

Variable ≜ Variable_attribute | Local

Variable_attribute ≜ Feature_name

Local $\triangleq$ Identifier | **Result**

Read_only $\triangleq$ Formal | Constant_attribute | **Current**

Formal $\triangleq$ Identifier

Constant_attribute $\triangleq$ Feature_name

The constraint on entities indicates that an entity must be of one of the five forms listed above. In addition, local variables, formal arguments and Object Test locals are only permitted in certain contexts:

Entity rule *VEEN*

An occurrence of an entity *e* in the text of a class *C* (other than as the feature of a qualified call) is valid if and only if it satisfies one of the following conditions:

1 • *e* is **Current**.

2 • *e* is the final name of an attribute of *C*.

3 • *e* is the local variable **Result**, and the occurrence is in a Feature_body, Postcondition or Rescue part of an Attribute_or_routine text for a query or an Inline_agent whose signature includes a result type.

4 • *e* is **Result** appearing in the Postcondition of a constant attribute's declaration.

5 • *e* is listed in the Identifier_list of an Entity_declaration_group in a Local_declarations part of a feature or Inline_agent *fa*, and the occurrence is in a Local_declarations, Feature_body or Rescue part for *fa*.

6 • *e* is listed in the Identifier_list of an Entity_declaration_group in a Formal_arguments part for a routine *r*, and the occurrence is in a declaration for *r*.

7 • *e* is listed in the Identifier_list of an Entity_declaration_group in the Agent_arguments part of an Agent *a*, and the occurrence is in the Agent_body of *a*.

8 • *e* is the Object-Test Local of an Object_test, and the occurrence is in its scope.

→ *"Scope of an Object-Test Local", page 653.*

"Other than as feature of a qualified call" excludes from the rule any attribute, possibly of another class, used as feature of a qualified call: in *a*.*b* the rule applies to *a* but not to *b*. The constraint on *b* is the General Call rule, requiring *b* to be the name of a feature in *D*'s base class.

→ *The General Call rule is on page 673.*

A related rule defines what it means for an entity to be a Variable:

Variable rule *VEVA*

A Variable entity *v* is valid if an only if it satisfies one of the following conditions:

1 • *v* is the final name of a variable attribute of *C*.

2 • *v* is the final name of a local variable of the immediately enclosing routine or agent.

This will determine whether you may use *e* as the target of an Assignment. Note that *v* in clause 2 has to be a local variable (including, as usual *Result*) of the *immediately* enclosing routine or agents. Routines may not be nested, but an agent appears in a routine (and possibly in another agent); only the local variables of the immediately enclosing scope are assignable.

→ Assignment *is discussed in chapter 22.*

19.9 SEMANTICS: EVALUATING AND INITIALIZING ENTITIES

The semantic purpose of an entity is to be ready at execution time to deliver an associated value whenever queried, or **evaluated**. The validity and semantic rules of the language must ensure that whenever this happens the entity denotes *exactly one* value, and to define what that value will be.

For read-only entities this is achieved through simple properties, whose details appear in other chapters:

- A constant attribute has the value specified in its declaration.
- **Current** gets attached to the root object on system start, and at the start of a qualified call *x*.*f* (…) denotes the value of the target *x*.
- On entry to a routine, a formal argument gets attached to the value of the corresponding actual.

→ *29.10, page 803.*

→ *"Current Semantics", page 643.*

→ *"PRECISE CALL SEMANTICS", 23.17, page 643.*

For a variable, the picture is a bit more subtle. The result of the evaluation is a consequence of the operations that may have affected the variable:

- **Initialization**, as it occurs on object creation (for an attribute) or a routine call (for a local variable).
- Any **assignment** using the variable as its target.

Assignment has a well-defined semantics, discussed in detail in the corresponding chapter. But the execution might evaluate the variable before it has been the target of any explicit assignment; it is the task of initialization rules to ensure that even in such a case every variable has one well-defined value.

→ *Chapter 22.*

This is not the case in all programming languages; many leave it to the programmer to ensure that every variable is assigned before use. In Eiffel, it is a language design principle that the rules must be sufficient to deduce, for any evaluation of any variable, a well-defined result.

The value of a variable that hasn't yet been the target of an assignment will be determined by the **initialization rules** that we will now study. These rules determine *which value* a variable will hold prior to assignment, and *when* exactly that value will be set.

---- TO BE REDONE ---There are two possibilities, depending on the type of the variable:

- The most common case covers variables of *basic types* as well as non-attached ones of *reference types*. An attribute of such a type denotes a field in the corresponding objects, and will accordingly be initialized as part of object creation. A local variable (including *Result* for a function) is initialized anew for each call of its routine. In both cases the initial values are language-specified: zero for numbers, false for booleans, null character for characters, and for references — covering all other possibilities — a void reference.

- *Expanded* types raise a special issue because their semantics require variables, when evaluated, always to be attached to an object of the corresponding type. Such an object cannot just follow from the declaration of the variable (like the value 0, in the previous case, follows from the declaration of an *INTEGER* variable); it has to come out of a creation instruction. The rule then is to create an object on *first evaluation* of the variable — meaning for an attribute the first evaluation for any given object, and for a local variable the first evaluation in any given call. The evaluation will cause creation of an object of the appropriate type, using the procedure *default_create*, which must be one of the creation procedures of the type.

This is the gist of the rules. Let now see their precise form. First we name ---- REWRITE our two type categories:

Self-initializing type

A type is **self-initializing** if it is one of:

1 • A detachable type.

2 • A self-initializing formal.

3 • An attached type (including expanded types and, as a special case of these, basic types) whose creation procedures include a version of *default_create* from *ANY*.

A self-initializing type enables us to define a default initialization value:

- Use *Void* for a detachable type (case 1, the easiest but also the least interesting)
- Execute a creation instruction with the applicable version of *default_create* for the most interesting case: 3, attached types, including expanded types. This case also covers basic types, which all have a default value given by the following rule.

A "self-initializing formal" (case 2) is a generic parameter, so we don't exactly know which one of these three semantics will apply; but we do require, through the Generic Derivation rule, that any attached type used as actual generic parameter be self-initializing, meaning in this case that it will provide *default_create*.

In the definition, the "creation procedures" of a *type* are the creation procedures of its base *class* or, for a formal generic parameter, its "constraining creators", the features listed as available for creation in its constraining type.

The more directly useful notion is that of a self-initializing *variable*, appearing below.

The term "self-initializing" is justified by the following semantic rule, specifying the actual initialization values for every self-initializing type.
:

Default Initialization rule

Every self-initializing type *T* has a **default initialization value** as follows:

1 • For a detachable type: a void reference.

2 • For a self-initializing attached type: an object obtained by creating an instance of *T* through *default_create*.

3 • For a self-initializing formal: for every generic derivation, (recursively) the default initialization value of the corresponding actual generic parameter.

4 • For *BOOLEAN*: the boolean value false.

5 • For a sized variant of *CHARACTER*: null character.

6 • For a sized variant of *INTEGER*: integer zero.

7 • For a sized variant of *REAL*: floating-point zero.

8 • For *POINTER*: a null pointer.

9 • For *TYPED_POINTER*: an object representing a null pointer.

This rule is the reason why everyone loves self-initializing types: whenever execution catches an entity that hasn't been explicitly set, it can (and, thanks to the Entity Semantics rule, will) set it to a well-defined default value. This idea gains extra flexibility, in the next definition, through the notion of attributes with an explicit initialization.

The notion generalizes ---- COMPLETE

Self-initializing variable

A variable is **self-initializing** if one of the following holds:

1 • Its type is a self-initializing type.

2 • It is an attribute declared with an Attribute part such that the entity **Result** is properly set at the end of its Compound.

If a variable is self-initializing, we don't need to worry about finding it with an undefined value at execution time: if it has not yet been the target of an attachment operation, automatic initialization can take over and set it to a well-defined default value. That value is, in case 1, the default value for its type, and in case 2 the result of the attribute's own initialization. That initialization must ensure that **Result** is "properly set" as defined next (partly recursively from the above definition) .

'I'

---- EXPLAIN

Evaluation position, precedes

An **evaluation position** is one of:

- In a Compound, one of its Instruction components.
- In an Assertion, one of its Assertion_clause components.
- In either case, a special **end position**.

A position p **precedes** a position q if they are both in the same Compound or Assertion, and either:

- p and q are both Instruction or Assertion_clause components, and p appears before q in the corresponding list.
- q is the end position and p is not.

This notion is needed to ensure that entities are properly set before use.

In a compound *i1; i2; i3* we have four positions; *i1* precedes *i2*, *i3* and the end position, and so on.

The relation as defined only applies to **first-level** components of the compound: if *i2* itself contains a compound, for example if it is of the form **if** *c* **then** *i4; i5* **end**, then *i4* is not an evaluation position of the outermost compound, and so has no "precedes" relation with any of *i1*, *i2* and *i3*.

Setter instruction

A **setter instruction** is an assignment or creation instruction.

If *x* is a variable, a setter instruction is a **setter for** *x* if its assignment target or creation target is *x*.

Properly set variable

At an evaluation position *ep* in a class *C*, a variable *x* is **properly set** if one of the following conditions holds:

1 • *x* is self-initializing.

2 • *ep* is an evaluation position of the Compound of a routine or Inline_agent of the Internal form, one of whose instructions precedes *ep* and is a setter for *x*.

3 • *x* is a variable attribute, and is (recursively) properly set at the end position of every creation procedure of *C*.

4 • *ep* is an evaluation position in a Compound that is part of an instruction *ep'*, itself belonging to a Compound, and *x* is (recursively) properly set at position *ep'*.

5 • *ep* is in a Postcondition of a routine or Inline_agent of the Internal form, and *x* is (recursively) properly set at the end position of its Compound.

6 • *ep* is **Result** in the Postcondition of a constant attribute

The key cases are 2, particularly useful for local variables but also applicable to attributes, and 3, applicable to attributes when we cannot deduce proper initialization from the enclosing routine but find that every creation procedure will take care of it. Case 4 accounts for nested compounds. For assertions other than postconditions, which cannot use variables other than attributes, 3 is the only applicable condition. The somewhat special case 6 is a consequence of our classification of **Result** among local variables even in the Postcondition of a constant attribute.

As an artefact of the definition's phrasing, every variable attribute is "properly set" in any effective routine of a deferred class, since such a class has no creation procedures. This causes no problem since a failure to set the attribute properly will be caught, in the validity rule below, for versions of the routine in effective descendants.

Variable Initialization rule *VEVI*

It is valid for an Expression, other than the target of an Assigner_call, to be also a Variable if it is properly set at the evaluation position defined by the closest enclosing Instruction or Assertion_clause.

This is the fundamental requirement guaranteeing that the value will be defined if needed.

Because of the definition of "properly set", this requirement is pessimistic: some examples might be rejected even though a "smart" compiler might be able to prove, by more advanced control and data flow analysis, that the value will always be defined. But then the same software might be rejected by another compiler, less "smart" or simply using different criteria. On purpose, the definition limits itself to basic schemes that all compilers can implement.

If one of your software elements is rejected because of this rule, it's a sign that your algorithms fail to initialize a certain variable before use, or at least that the proper initialization is not clear enough. To correct the problem, you may:

- Add a version of *default_create* to the class, as creation procedure.
- Give the attribute a specific initialization through an explicit Attribute part that sets **Result** to the appropriate value.

Variable setting and its value

A **setting** for a variable *x* is any one of the following run-time events, defining in each case the **value** of the setting:

1 • Execution of a setter for *x*. (*Value*: the object attached to *x* by the setter, or a void reference if none.)

2 • If *x* is a variable attribute with an Attribute part: evaluation of that part, implying execution of its Compound. (*Value*: the object attached to **Result** at the end position of that Compound, or a void reference if none.)

3 • If the type *T* of *x* is self-initializing: assignment to *x* of *T*'s default initialization value. (*Value*: that initialization value.)

As a consequence of case 2, an attribute *a* that is self-initializing through an Attribute part *ap* is *not* set until execution of *ap* has reached its end position. In particular, it is not invalid (although definitely unusual and perhaps strange) for the instructions *ap* to use the value *a*: as with a recursive call in a routine, this will start the computation again at the beginning of *ap*. For attributes as for routines, this raises the risk of infinite recursion (perhaps higher for attributes since they have no arguments) and it is the programmer's responsibility to avoid this by ensuring that before a recursive call the context will have sufficiently changed to ensure eventual termination. No language rule can ensure this (in either the routine or attribute cases) since this would amount to solving the "halting problem", a provably impossible task.

Another consequence of the same observation is that if the execution of *ap* triggers an exception, and hence does not reach its end position, any later attempt to access *a* will also restart the execution of *ap* from the beginning. This might trigger the same exception, or succeed if the conditions of the execution have changed.

Execution context

At any time during execution, the current **execution context** for a variable is the period elapsed since:

1 • For an attribute: the creation of the current object.

2 • For a local variable: the start of execution of the current routine.

Variable Semantics

The value produced by the run-time evaluation of a variable *x* is:

1 • If the execution context has previously executed at least one setting for *x*: the value of the latest such setting.

2 • Otherwise, if the type *T* of *x* is self-initializing: assignment to *x* of *T*'s default initialization value, causing a setting of *x*.

3 • Otherwise, if *x* is a variable attribute with an Attribute part: evaluation of that part, implying execution of its Compound and hence a setting for *x*.

4 • Otherwise, if *x* is **Result** in the Postcondition of a constant attribute: the value of the attribute.

This rule is phrased so that the order of the first three cases is significant: if there's already been an assignment, no self-initialization is possible; and if *T* has a default value, the Attribute part won't be used.

The Variable Initialization rule ensures that one of these cases will apply, so that *x* will always have a well-defined result for evaluation. This property was our main goal, and its achievement concludes the discussion of variable semantics.

The previous rule applies only to variables. We now generalize it to a general rule governing all entities:

Entity Semantics rule

Evaluating an entity yields a **value** as follows:

1 • For **Current**: a value attached to the current object.

2 • For a formal argument of a routine or Inline_agent: the value of the corresponding actual at the time of the current call.

3 • For a constant attribute: the value of the associated Manifest_constant as determined by the Manifest Constant Semantics rule.

4 • For an Object-Test Local: as determined by the Object-Test Local Semantics rule.

5 • For a variable: as determined by the Variable Semantics rule.

← *"Current object, current routine", page 641*.

This rule concludes the semantics of entities by gathering all cases. It serves as one of the cases of the semantics of expressions, since an entity can be used as one of the forms of Expression.

The Object-Test Local Semantics rule appears in the discussion of the Object_test construct.