

# Creating objects

## 20.1 OVERVIEW

The dynamic model, whose major properties were reviewed in the preceding presentations, is highly flexible; your systems may create objects and attach them to entities at will, according to the demands of their execution. The following discussion explores the two principal mechanisms for producing new objects: the **Creation\_instruction** and its less frequently encountered sister, the **Creation\_expression**.

A closely related mechanism — **cloning** — exists for duplicating objects. This will be studied separately, with the mechanism for copying the contents of an object onto another.

The creation constructs offer considerable flexibility, allowing you to rely on language-defined initialization mechanisms for all the instances of a class, but also to override these defaults with your own conventions, to define any number of alternative initialization procedures, and to let each creation instruction provide specific values for the initialization. You can even instantiate an entity declared of a generic type — a non-trivial problem since, for  $x$  declared of type  $G$  in a class  $C [G]$ , we don't know what actual type  $G$  denotes in any particular case, and how one creates and initializes instances of that type.

In using all these facilities, you should never forget the methodological rule governing creation, as expressed by the following principle.



### Creation principle

Any execution of a creation operation must produce an object that satisfies the invariant of its generating class.

Such is the theoretical role of creation: to make sure that any object we create starts its life in a state satisfying the corresponding invariant. The various properties of creation, reviewed next, are designed to ensure this principle.

## 20.2 FORMS OF CREATION: AN OVERVIEW



You may use a **Creation\_instruction** to produce a totally new object, initialize its variable fields to preset values, and attach it to a **Variable** entity called the **target** of the creation and named in the instruction.

The examples which follow assume that the target is of a reference (non-expanded) type. As will be seen below, the **Creation\_instruction** is also applicable to expanded types, although with a less interesting effect.

See [20.8, page 534](#) below, about *Creation instructions applied to expanded types*.

Syntactically, a **Creation\_instruction** always begins with the keyword **create**, followed by the target. Here are some examples:



<b>create</b> <i>account1</i>	[1]
<b>create</b> <i>point1</i> . <i>make_polar</i> (1, Pi / 4)	[14]
<b>create</b> { <i>SAVINGS_ACCOUNT</i> } <i>account1</i>	[15]
<b>create</b> { <i>SEGMENT</i> } <i>figure1</i> . <i>make</i> ( <i>point1</i> , <i>point2</i> )	[16]

The respective targets are *account1*, *point1*, *account1*, *figure1*.

With form [1] you create an object of the type declared for *account1*, initialize it to default values, and attach it to *account1*. The default initialization is language-defined, although you can override it for any class.

With form [14] you create an object of the type declared for *point1*, apply the standard default initialization, complement the initialization by calling *make\_polar* (a procedure of the class, designated as one of its “creation procedures”) with the given arguments, and attach the object to *point1*.

Cases [15] and [16] are respectively similar to the first two, but specify an explicit type, in braces, for the new object. So if *account1* is of type *ACCOUNT*, form [1] creates an instance of that class, but form [15] creates an instance of *SAVINGS\_ACCOUNT*. This requires *SAVINGS\_ACCOUNT* to be a descendant of *ACCOUNT*. Similarly, in form [16], *SEGMENT* must be a descendant of the type, say *FIGURE*, declared for *figure1*.

--- ADD INTRO TO CREATION EXPRESSIONS ---

Since the run-time effect of a creation instruction or expression is essentially the same, it is convenient to have a name covering both:

### Creation operation

A **creation operation** is a creation instruction or expression.

## 20.3 BASIC FORM OF CREATION INSTRUCTIONS

Even though example [1] shows the most concise variant, a better place to start studying the `Creation_instruction` is the more general variant illustrated by [14]: `create x.creation_procedure (...)`. Its effect is, in order, to:

- 1 • Create a new object — a direct instance of the type  $T$  of  $x$ .
- 2 • Initialize all the variable fields of that object to default values.
- 3 • Call `creation_procedure` on the object, with the arguments given, to complete its initialization.
- 4 • Attach  $x$  to the object.

The default initialization values used in step 2 are adapted to the type of each field corresponding to a variable attribute: zero for numbers, false for booleans, void for references and so on. The full rule will appear later.

→ “[Default Initialization rule](#)”, page 508.

This form of the instruction is only valid if the base class  $C$  of  $x$ 's type  $T$  lists `creation_procedure` in its `Creators` part.

Such a `Creators` part is permitted only in an effective class (since it makes no sense to create direct instances of a deferred class). We have seen that it comes towards the beginning of a class text — just before `Features` but after `Inheritance` — and consists of at least one `Creation_clause`, each beginning with the keyword `create` followed by a list of zero or more procedures of the class, as in

← “[PARTS OF A CLASS TEXT](#)”, 4.7, page 119.



```
class C ... inherit
    ...
create
    make, execute, ...
feature
    ...
end
```

where `make, execute ...` are procedures of  $C$ . For the moment we are restricting ourselves to just one `Creation_clause` (the vast majority of cases). By including such a clause, the author of  $C$  specifies that any `Creation_instruction` producing direct instances of the class must be of one of the two forms

→ You can use more than one `Creation_clause`; also, each one may restrict clients' creation privileges. See below “[RESTRICTING CREATION AVAILABILITY](#)”, 20.7, page 531 for full details.



```
create x.make (...)
create x.execute (...)
```

which will initialize the new object by calling the specified creation procedure — with actual arguments whose types and number match those of the formal arguments declared for the procedure.



The two creation-related constructs, **Creators** and **Creation\_instruction**, both use the same keyword **create**. This makes things easier to remember than if you had to learn two keywords. No confusion can result since the constructs appear in completely different syntactic contexts.

Creation procedures (also known as “*constructors*” from C++ terminology) serve to apply initializations beyond the default ones if these do not suffice. For example, the author of a class **POINT** in a graphics system may wish to offer a creation mechanism that not only allocates a new object but also initializes its fields according to coordinates provided by the client. Here is an outline of such a class:



```

class POINT inherit
  TRIGONOMETRY
create
  make_polar, make_cartesian
feature -- Access
  ro, theta: REAL
  x, y: REAL
feature -- Element change
  make_polar (r, t: REAL)
    -- Set to polar coordinates r, t.
  do
    ro := r; theta := t
    reset_from_polar
  end
  make_cartesian (a, b: REAL)
    -- Set to cartesian coordinates a, b.
  do
    x := a; y := b
    reset_from_cartesian
  end
  ... Other exported features ...
feature {NONE} -- Implementation
  consistent_attributes: BOOLEAN
    -- Do polar and cartesian attributes
    -- represent same point?
  do
    Result := (x = ro * cos (theta)) and
      (y = ro * sin (theta))
  end

```

*This example assumes a library class **TRIGONOMETRY** offering functions such as *cos* and *sin*. The equality in *consistent\_attributes* should be changed to an approximate equality to account for numerical precision issues.*

```

reset_from_polar
    -- Update cartesian coordinates from polar ones.
do
    x := ro * cos (theta); y := ro * sin (theta)
ensure
    consistent_attributes
end

reset_from_cartesian
    -- Update polar coordinates from cartesian ones.
do
    ...
ensure
    consistent_attributes
end

invariant
    consistent: consistent_attributes
end

```

With this design, the author of class *POINT* provides clients with two creation mechanisms: one initializes a point by its polar coordinates, the other by its cartesian coordinates. Examples of *Creation\_instruction*, assuming that *point1* is a *Variable* entity of type *POINT*, are

```

create point1.make_polar (2, Pi / 4)
create point1.make_cartesian (Sqrt2, Sqrt2)

```

*If Pi and Sqrt2 are real constants with the values suggested by their names, these instructions will have the same effect.*



Names of the form *make\_something* are common practice for creation procedures, although by no means required. When a class has just one creation procedure, or one more fundamental than the others, the convention is to call it just *make* — although if the procedure has no arguments your clients can ignore it altogether, if you use *default\_create* as will now be seen.

## 20.4 OMITTING THE CREATION PROCEDURE

In some common cases you can avoid specifying a creation procedure. This gives the simplest possible form of *Creation\_instruction*, illustrated by the first of our initial examples:

```

create x

```

This form is applicable when the base class *C* of *x*'s type does *not* have a **Creators** part. This is particularly useful for simple classes which do not need particularly flexible creation mechanisms, but just provide clients with a standard way to create instances without providing any specific information. These instances will all be initialized in the same way. A simple example is



```
note
  description: "[%Binary trees with nodes containing
                information of type G%]"
class BINARY_TREE [G]... feature -- Access
  item: G
    -- Node information
  left, right: BINARY_TREE [G]
    -- Left and right children
feature -- Element change
  ... Features to set node information and attach children...
end
```

Here a creation instruction, for *bt* of type *BINARY\_TREE* [*SOME\_TYPE*], will simply be



```
create bt
```

and will set all the fields of the resulting object to their default values: void references for *left* and *right*, the default value of the actual generic parameter (whatever it may be) for *item*.

This simple form of the **Creation\_instruction** is appropriate when the object-creating client is happy to rely on a standard initialization. But even in this case you may need more fine-tuning, because the language-defined default initializations might not suit all classes. Consider



```
class EMPLOYEE inherit
  PERSON
  feature -- Access
    Unknown_marital_status, Single, Widowed, Divorced:
      INTEGER -- !!!!! REDO EXAMPLE !!!!!
    marital_status: INTEGER
  feature
    ... Other features ...
  invariant
    meaningful_marital_status:
      marital_status >= Unknown_marital_status and
      marital_status <= Divorced
  end
```

We require, as expressed by the invariant, that *marital\_status* have one of the values listed. Because this attribute is of type *INTEGER*, the universal default initializations would set it to zero — not compatible with the invariant! Remember the **Creation principle**: it is creation's responsibility to ensure that every new object satisfies the invariant.

*Creation principle:*  
page 515.

One solution is to use a creation procedure:



```
class EMPLOYEE inherit
  PERSON
  create
    make
  feature -- Initialization
    make
      -- Initialize by setting marital status to "Unknown".
    do
      marital_status := Unknown_marital_status
    end
  feature -- Access
    ... Other features and invariant as before ...
  end
```

Since the class now has a **Creators** part, the abbreviated form **create emp** (for *emp* of type *EMPLOYEE*) is no longer valid: we are back to the previous technique and must write



```
create emp.make
```

This approach works but is a bit tedious for the clients since they must specify a creation procedure for no clear benefit: only one such procedure is available, *make*, and it takes no argument.

In such a case — providing a standard initialization, but not necessarily the universal language-defined one — you can still make the simple creation form **create** *x* valid for your clients. Do not include a **Creators** part; just redefine the procedure *default\_create* which, coming from class *ANY*, is a feature of all classes. This redefinition will specify your desired initializations.

This technique relies on a simple convention: any class *C* without a **Creators** part is treated as if it had one of the form

```
create
  default_create
```

(If *default\_create* has been renamed, this should use the new name instead.) In other words, a class which doesn't list any creation procedures is considered to have just one — its version of *default\_create*.

Correspondingly, a **Creation\_instruction** of the form **create** *x*, which doesn't specify a creation procedure, is treated as a shorthand for

```
create x.default_create
```

for *x* of a type based on *C* (again with the understanding that, if *default\_create* has been renamed, this unfolded form uses the new name).

With this technique we can adapt class *EMPLOYEE* so that its clients can create instances by writing just



```
create emp
```

with no creation procedure. The new form of the class is almost the same as the last one seen, but instead of a specific creation procedure *make* we don't include any **Creators** part and just redefine *default\_create*:



```

class EMPLOYEE inherit
  PERSON
  redefine default_create end

feature -- Initialization
  default_create
    -- Initialize by setting marital status to "Unknown".
    do
      marital_status := Unknown_marital_status
    end

feature -- Access
  ... Other features and invariant as before ...
end

```



Because such a class redeclares a feature *default\_create* which it inherits in non-deferred form, it must state **redefine** *default\_create* in some **Inheritance** part. Here *EMPLOYEE* inherits from *PERSON*, so we just stick this clause into the corresponding **Inheritance** part. If the class didn't have any **Inheritance** part — meaning that it only has an implicit parent, *ANY* — we would have to use the standard idiom enabling such a class to redefine a feature coming from *ANY*: include an **Inheritance** part making *ANY* an explicit rather than implicit parent. This would give:



```

class EMPLOYEE inherit
  -- Here we make ANY an explicit parent:
  ANY
  redefine default_create end

feature -- Initialization
  ... Feature clauses and invariant as before ...
end

```

Let's review the two schemes studied in the previous section and this one:

- 1 • To provide clients with specific creation procedures, which may take arguments, include at the beginning of the class a **Creators** part, of the form **create** *cp1*, *cp2*, ... , where the *cp<sub>i</sub>* are procedures of the class. A **Creation\_instruction** in this case must be of the form **create** *x.cp* (...) where *cp* is one of the specified *cp<sub>i</sub>*.
- 2 • To make the simplified form **create** *x* valid, you do not need to include any **Creators** part: this form is equivalent to the previous case using for *cp* the procedure *default\_create*; and an absent **Creators** part is equivalent to one that lists only that procedure.

At first these two cases may seem incompatible, but if you examine them more closely you will realize they are not. The rule is simply that the simplified form `create x` is valid if and only if `default_create`, in its local version, is one of the creation procedures of the class. You can achieve this property by not listing any creation procedures at all: this is equivalent to listing `default_create` only. But you can also have a **Creators** part, provided it lists `default_create`, possibly among other procedures. This observation yields a third case, combining the previous two:

- 3 • To make both forms of creation instruction valid — the form with an explicit procedure, `create x.cpi (...)` for some `cpi`, and the procedure-less form, `create x` — simply include a **Creators** part that lists both the desired `cpi` and the class's version of `default_create`.

Here is an example of this last scheme, a variation on an earlier class text:

← See the original version on page [518](#).



```
class POINT inherit
  TRIGONOMETRY
create
  make_polar, make_cartesian, default_create
feature
  ... Features as before ...
invariant
  consistent: consistent_attributes
end
```

Then all of the following four creation instructions are valid:

```
[1]
  create your_point.make_polar (2, Pi/4)
[2]
  create your_point.make_cartesian (Sqrt2, Sqrt2)
[3]
  create your_point.default_create
[4]
  create your_point
```

Forms **3** and **4** are exactly equivalent, so there is usually little reason to use **3** except if you insist on including the creation procedure for clarity.



Note that including `default_create` among the creation procedures, hence permitting **4**, makes sense only because the default initializations ensure the invariant `consistent_attributes`, which states that cartesian and polar coordinates agree — true if they are all zero, the default. When thinking about creation, always keep in mind the **Creation principle**.

← Creation principle: page [515](#).

As a variation on this example, assume that you write a class *C* that inherits from a parent *B* a procedure *set* without arguments, and want *C* to offer its clients the procedure-less form **create** *x* so that it will call *set* for initialization. A simple technique is:



```
class C inherit
  B
    rename
      default_create as discarded
    end
  ANY
    rename
      default_create as set
    undefine
      set
    select
      set
    end
feature
  ...
end
```

This uses a **join** to merge two inherited features, undefining *default\_create* along one of the branches so that its joined feature *set* can override its previous implementation. Corresponding creation instructions may be written **create** *x*.

← See “[THE JOIN MECHANISM](#)”, 10.21, page 286.

We can now summarize the basic rule for validity of a creation instruction: the *instruction’s creation procedure* must be one of the *class’s creation procedures*, with the understanding that:

- 1 • Every creation instruction uses a creation procedure — either explicit, as in **create** *x.cp (...)*, or implicit, as in **create** *x*, where the instruction’s creation procedure is *default\_create*.
- 2 • Every class lists a set of creation procedures — either explicit, if the class has a **Creators** part, or implicitly taken to be *default\_create* in the absence of a **Creators** part.

This also suggests, as a special case, what you should do if for some reason you do **not** want clients of a class to create any direct instances of it. Simply include a **Creators** part, but make it empty:



```
class NOT_INSTANTIABLE create
  -- Nothing at all listed here!
feature
  ...
end
```

WARNING: not the recommended style; see next.

This falls under the “explicit” case of observation 1 above, so that under observation 2 a creation instruction could only be valid if it were of the form `create x.cp (...)` where `cp` is a creation procedure of the class; but there is no such `cp` since the **Creators** part, although present, is empty.



The style guideline in such a case is actually to write

```
class NOT_INSTANTIABLE create {NONE}

feature

end ...
```

which has exactly the same effect but emphasizes the creation ban by listing `NONE` as the single creation (rather, non-creation) client, based on conventions, seen below, for restricting creation availability. → [“RESTRICTING CREATION AVAILABILITY”, 20.7, page 531.](#)

Another way to make a class non-instantiable is to declare it as deferred. But you might want to prohibit instantiation of a class even if it is effective. Then you can use the technique just seen.

## 20.5 CREATORS AND INHERITANCE



(This section is a discussion of the *absence* of dependency between two language concepts, so it introduces no new mechanism; it is a “comment” and “methodology” section meant to dispel a possible confusion, which might in particular follow from experience with other languages.)

You may have been wondering what effect the inheritance structure has on the creation procedures of a class. The short answer is: *no effect*. Each class is free to choose the procedures it wants to offer to its clients for creation, regardless of its parents’ choices. The creation mechanism does of course take full advantage of inheritance: creation procedures may be obtained from parents and adapted through the usual inheritance mechanisms of redefinition, renaming, effecting and so on. And in some cases a class’s choice of creation procedures is directly connected to its parents’ choices:

- A class may list as creation procedures (in its **Creators** part) some or even all of a parent’s own creation procedures.
- A redefined creation procedure may need, as part of its execution, to call the parent’s version, usually through the **Precursor** mechanism.

But all this is optional, not required, and neither theoretical analysis nor analysis of practical examples suggests an obligatory connection. Counterexamples indeed abound. Just think of a class `POLYGON`, where a typical creation procedure will take a list of vertices; for its heir `RECTANGLE` this is most likely inappropriate, as we might use a center, an orientation and two side lengths; then for a grandchild `SQUARE` we will again need something different since we can dispense with one of these lengths.

So the set of creation procedures of a class is entirely determined by its **Creators** clause (or lack thereof, as we have seen), without interference from the parents' own clauses. This yields a simple semantics and avoids confusion. Based on the needs of each class, you decide what creation privileges you award to *your* clients; you may reuse the parents' creation procedures, unchanged or extended, but only if you find them useful for your own needs.

Eiffel's policy on relating *creation status* to inheritance is similar to its policy on relating *export status* to inheritance. There too every class is free to make its own decisions for inherited features, regardless of its parents' choices. The only difference is the default: inherited features retain their original export status unless the heir explicitly overrides it (through a **New\_exports** clause); in contrast, a creation procedure loses its creation status unless the heir explicitly reaffirms it (by listing the procedure in its own **Creators** part). This difference follows from an analysis of what designers most commonly need, in each case, in the practice of building systems.

← "[Adapting the export status of inherited features](#)", . . . page 200.

## 20.6 USING AN EXPLICIT TYPE

In the variants seen so far, the type of the object created by a creation instruction **create** *x* . . . , with or without an explicit creation procedure, is the type *T* declared for *x*, the instruction's target. You may want to use another type *V* instead; this will be permitted if *V* conforms to *T*. The form of the instruction in this case is one of

```
create {V} x.cp (..)
create {V} x
```

with the first one valid only if *cp* is a creation procedure of *V*, and the second only if *default\_create* is a creation procedure of *V* (in particular if *V*'s base class has no **Creators** part).

### Specifying the creation type



Assume class *SEGMENT* is a descendant of *FIGURE*, and has a creation procedure *make*, with two formal arguments of type *POINT* representing the end points of a segment. The following will be valid:

```
[1]
  fig: FIGURE
  point1, point2: POINT
  ...
  create {SEGMENT} fig.make (point1, point2)
```

and will have exactly the same effect on *fig* as

```
[2]
  fig: FIGURE; seg: SEGMENT
  point1, point2: POINT
  ...
  create seg.make (point, point2)
  fig := seg
```

where the last instruction is a polymorphic assignment, permitted by the [Assignment rule](#) since *seg* conforms to *fig*.

→ The Assignment rule, stating that the type of an assignment's source must conform to that of its target, is on page [582](#).

The explicitly typed form [1](#) brings nothing fundamentally new; it is just an abbreviation for the implicitly typed form [2](#), avoiding the need to introduce intermediate entities such as *seg*.

As a consequence of this new form, we can define the **creation type** of a creation instruction — the type of the object that it will create: in the previous form `create x ...`, the creation type is the type declared for the target, *x*; in the explicit form `create {V} x ...`, the creation type is *V*.

→ The formal definition will appear on page [543](#).

## Choosing between types

To become really useful the example should include more than one case: after all, if all you ever want to obtain is an instance of *SEGMENT*, then you do not need *fig*; *seg* suffices. Things become more interesting with a scheme of the following kind, using a local variable *fig* of type *FIGURE*:

```
[3]
inspect
  icon_selected_by_user
when Segment_icon then
  create {SEGMENT} fig.make (point1, point2)
when Triangle_icon then
  create {TRIANGLE} fig.make (point1, point2, point3)
when Circle_icon then
  create {CIRCLE} fig.make (point1, radius)
when ...
  ...
end
```

Here *SEGMENT*, *TRIANGLE*, *CIRCLE*, ... are descendants of *FIGURE*, all with specific creation procedures, and *Segment\_icon*, *Triangle\_icon*, *Circle\_icon*, ... are integer constants with different values. Depending on the icon selected by an interactive user, the above instruction creates an object of the appropriate type, and attaches *fig* to it.

Were the explicitly typed form of the creation instruction not available, you could still use the equivalence illustrated by [2](#), rather unpleasant here because you need to declare a temporary entity (*seg*, *tri*, *circ*, ...) for each of the possible icon types.

## Creation and deferred classes



Scheme [3](#) helps understand the role of **deferred classes and types** vis-à-vis creation. A class must be declared as **deferred** if it has at least one deferred feature (introduced in the class itself, or inherited from a parent, and not effected — made effective — in the class). A deferred type is one based on a deferred class. In our example we may assume *FIGURE* to be deferred, but the concrete descendants used in the creation instructions — *SEGMENT* and so one — to be effective. The rule is that:

← Although a class may be declared as **deferred** even without deferred features, the common case is for a deferred class to have one or more deferred features. See [10.11, page 266](#).

- We *never* permit a creation instruction to use a deferred type as creation type. As noted in the last chapter, creating direct instances of a deferred type would be asking for trouble, since clients could then call unimplemented operations on these instances. The creation rules of this chapter exclude this possibility; with *fig* of type *FIGURE*, we are not permitted to write **create** *fig* ... , with or without a creation procedure.
- We may, however, use *fig* as target of a creation instruction such as **create** {*SEGMENT*} *fig*.*make* (*point1*, *point2*) or any of the others above, even though the type of *fig* is deferred: that's fine as long as the **creation type** of the instruction is explicit and effective, like *SEGMENT* here. The instruction will create a direct instance of that type, so everything is in order. Attaching this object to an entity *fig* of a deferred type is also in order: it's simply an application of polymorphism.

“Direct instance” is in fact not even defined for deferred types. See “[INSTANCES AND VALUES](#)”, [11.5, page 321](#).

In summary: we cannot create **objects** of deferred types, but we can have **entities** of such types, which will become attached to instances of conforming effective types.

## Single choice and factory objects



Beyond its applicability to polymorphic entities of deferred types, what makes scheme [3](#) especially interesting is its connection with **dynamic binding**: after executing the above *Multi\_branch* instruction, you normally should never have to discriminate again on the type of *fig*; instead, to apply an operation with different variants for the figures involved, you should use a call of the form

*fig*.*display*

where the operation, here *display*, is redefined in various ways in descendants of *FIGURE*. This will select the appropriate version depending on the exact type of the object to which *fig* is attached, as a result of the variable-type creation achieved by 3.

This example illustrates an important concept of Eiffel software development: the **Single Choice principle**. The principle states that in a software system that handles a number of variants of the same notion (such as the figure types in a graphics system) any exhaustive knowledge of the set of possible variants should be confined to just **one component** of the system. This is essential to prevent future additions and modifications from requiring extensive system restructuring.

*See also 17.6, page 483, on explicit discrimination. For further discussion of these issues see "Object-Oriented Software Construction", in particular the Open-Closed Principle. .*

Often, the component that performs the "Single Choice" will be the one that initially creates instances of the appropriate objects; 3 illustrates one of the possible schemes.

There is a simpler scheme, avoiding any explicit control structure: the *clonable array technique*, implementing what the Design Pattern literature calls the **Factory Pattern**, although it was described in Eiffel literature and widely used in Eiffel programs many years before that term appeared in print.

Here is how it would work in this example. You assign a unique code to every variant

*Low\_id, Segment\_id, Triangle\_id, Circle\_id, ... , High\_id:*  
*-- REDO EXAMPLE -----*

and create a data structure, most conveniently an array, containing one direct instance of each variant:

```
[4]
  figure_factory: ARRAY [FIGURE]
  local
    fig: FIGURE
  once
    Result.make (Low_id, High_id)

    -- Create and enter a SEGMENT instance:
    create {SEGMENT} fig.make (...)
    Result.put (fig, Segment_id)

    -- Create and enter a TRIANGLE instance:
    create {TRIANGLE} fig.make (...)
    Result.put (fig, Triangle_id)

    ... Do the same for each variant ...

  end
```

*WARNING: there is a much more concise way to express this, using creation expressions and avoiding altogether the need to declare a local variable *fig*. See 1, page 551, which is the model you should use for this pattern.*



Instead of making `figure_factory` a once function you can declare it as an attribute, and then initialize it accordingly (with the instructions of the above routine body, substituting `figure_factory` for `Result`) in an initialization module. But initialization modules that take care of initializations for many different aspects of a system are not good for modular, extensible software construction. Using a once function is usually a better approach since it has the same effect but lets the initialization happen automatically the first time any part of the system needs to access `figure_factory`.

Then, whenever you actually need to select an alternative, you can avoid the explicit discrimination of **3**: replace the *entire* `Multi_branch` instruction by

```
[5]
    fig := clone (figure_factory @ code)
```

where `code` is the desired figure code (one of `Segment_id`, `Triangle_id` etc.). The function `clone` appearing on the right-hand side produces a new object copied from its argument; so each time you use **5** you get a new object which, depending on the value of the index `code`, will be a `SEGMENT`, or a `TRIANGLE` and so on.

*figure\_factory @ code denotes the item of index code, also written figure\_factory.item(code); see 36.4, page 924.*

→ “CLONING AN OBJECT”, 21.4, page 567.

## 20.7 RESTRICTING CREATION AVAILABILITY



The `Creators` parts in the preceding examples had at most one `Creation_clause`, and any client could create direct instances through any of the creation procedures listed there. It is also possible to define more restrictive client creation privileges. Let us take a look at this simple facility which, although not needed in elementary uses, helps build well-engineered systems that thoroughly apply the principle of information hiding.

← See 7.7, page 196, on information hiding.

You may indeed write a `Creators` part with one or more `Creation_clause` listing procedures available for creation by specific clients, as in



```
class C ... create
    make
create {A, B}
    jump_start, bootstrap
feature
    ...
end
```

The first `Creation_clause` has no restriction, so that any client can create a direct instance of `C` through an instruction `create x.make (...)` for `x` of type `C`. Because of the restriction in the second clause, however, only the descendants of `A` and `B` may use the given procedures for creation, in instructions `create x.jump_start (...)` or `create x.bootstrap (...)`.

*Remember that descendants of a class include the class itself.*

This possibility of including more than one **Creation\_clause**, each specifying that certain procedures of the class are creation procedures and giving a creation availability status, is, as you will certainly have noted, patterned after the convention for making the features of a class available to clients with a specified export status for calls. In the same way that a **Feature\_clause** may begin with one of



- [1]
  - feature**
  - ... Declaration of features callable by all clients ...
  - feature {NONE}**
- [2]
  - ... Declaration of features callable by no clients ...
- [3]
  - feature {X, Y}**
  - ... Declaration of features callable by descendants of X and Y ...

a **Creation\_clause** may begin with one of



- [1]
  - create**
  - ... List of procedures available for creation to all clients ...
  - create {NONE}**
- [2]
  - ... List of procedures available for creation to no clients ...
- [3]
  - create {X, Y}**
  - ... List of procedures available for creation to descendants of X and Y ...



Note, however, that such flexibility is not as essential for creation as it is for feature call. As part of the fundamental O-O principles of abstraction and information hiding, it is common to have several feature clauses specifying different levels of call availability: to all clients, to some clients, to no clients. This is less frequently useful for creation, and in practice many classes have just one **Creation\_clause**, or none.



The language supports the full generality of the mechanism anyway, partly for consistency with the other mechanism, and partly because the extra control over creation availability is occasionally useful.



Make sure not to confuse the two forms of specifying availability. When you list a set of creation procedures, as in **1**, **2** and **3** for a class *C*, you are only controlling the validity of a **Creation\_instruction** involving a creation call, such as

```
[1]
create x.cp (...)
```

for *x* of type *C*: valid everywhere in case **1**, invalid everywhere with **2**, and valid only in descendants of *X* and *Y* with **3**. This is completely independent of the availability status for plain (non-creation) calls such as

```
[1]
x.cp (...)
```

valid everywhere in case **1**, invalid everywhere with **2**, and valid only in descendants of *X* and *Y* with **3**. For the same *cp*, the two properties are separate. They reflect different semantics:

- The creation call **create** *x.cp* (...) creates an object and initializes it using *cp*.
- The plain call *x.cp* (...) uses *cp* to reinitialize an existing object – a right which, as the designer of a class, you may decide to grant or not to grant to clients, regardless of the right you have granted regarding the use of *cp* for creation-time initialization.

You may indeed be justified in deciding on different privileges in each case. Consider a class manipulating bank accounts:



```
class
  ACCOUNT
create
  make
feature {NONE} -- Initialization
  make (initial: AMOUNT)
    -- Set balance to initial.
  is do ... end
feature -- Element change
  withdraw (a: AMOUNT)
    -- Record removal of a units of currency.
  do ... end
  deposit (a: AMOUNT)
    -- Record addition of a units of currency.
  do ... end
  ... Other features, invariant ...
end-- class ACCOUNT
```

The use of **feature** *{NONE}* for the declaration of the class's creation procedure is a common Eiffel idiom, but surprising at first here: why hide this fundamental operation on the class? The reason is that we are hiding it for call, not for creation. The **Creation\_instruction**

```
create your_account.make (some_amount)
```

is indeed valid since *make* appears in an unrestricted **Creators** clause (lines 3 and 4, highlighted in the class above). What is **not** valid is a plain call

```
your_account.make (some_amount)
```

**WARNING:** not valid with class text as given.

which would reinitialize the account to *some\_amount*. The author of class *ACCOUNT* has decided that the only way to affect the balance of an account is to deposit or withdraw money (adding a value, positive or not, to the balance, rather than setting it to a specified value). Such policies are often legitimate and explain why **feature** *{NONE}* is a common style for declaring a creation procedure, even one that is unrestrictedly available for creation.

## 20.8 THE CASE OF EXPANDED TYPES

---- THIS SECTION IS NOW WRONG, REWRITE (lazy initialization) --  
-----

The preceding examples assumed that the type of the target entity was a reference (non-expanded) type. What if it is expanded?

In this case there is no need to create an object, since the value of the target is already an object, not a reference to an object that a **Creation\_instruction** must allocate dynamically.

Rather than disallowing **Creation\_instruction** for expanded targets, it is convenient to define a simple semantics for the instruction in this case, limited to the steps of the above process that still make sense: the instruction will execute the default initializations on the object attached to the target, then call the appropriate version of *default\_create*. This convention also has the advantage that if you change your mind about the expanded status of a class you can change it without to worry about its **Creation\_clause** becoming invalid.

As a consequence of this rule, if we have a class whose instances contain sub-objects, as in

```
class COMPOSITE feature
  a: SOME_REFERENCE_TYPE
  b: SOME_EXPANDED_TYPE
  ...
end
```



then the default initialization rule for the *b* field of a *COMPOSITE* instance will be to apply a *Creation\_instruction*, recursively, to the corresponding sub-object. This creation instruction will use as creation procedure the version of *default\_create* in the corresponding base class.

--- NO LONGER QUITE TRUE, REWRITE -----This semantic rule justifies a basic constraint on expanded types (given in the chapter on classes as the *Class Header rule*): the base class of an expanded type **must** have its version of *default\_create* as one of its creation procedures (either explicitly in its *Creators* part, or implicitly by not having a *Creators* part). This does not prevent the class from having other creation procedures if desired; but for automatic initialization of sub-objects such as *b* the procedure to be applied is *default\_create*, as any other choice would require further information from the client (choice of creation procedure and actual arguments). ← Page 126,

## 20.9 CREATING INSTANCES OF FORMAL GENERICS

More delicate than the expanded types is the case in which we would like to create an instance of one of the *Formal\_generic* parameter types of a class, as in *create x..* where *x* is of type *G* in a class *C [G]*.

The problem is that *G*, in the class text, denotes not a known type but a placeholder for many possible types or, in the case of unconstrained genericity, *any* valid type. So we have no way to know what creation procedures will be available on the corresponding instances.

This seems at first to preclude any hope of allowing creation instructions in this case. Fortunately, constrained genericity allows an elegant solution.

As you know, constrained genericity is the mechanism that allows us to declare a class as

```
class C [G → CONST] ...
```

where *CONST* is a type, known as the constraining type for the formal generic parameter *G*. Then you may only write a generic derivation *C [T]*, using a type *T* as actual generic parameter, if *T* conforms to *G*. The benefit is that, within class *C*, you know that any entity *x* of type *G* represents objects of type *T* or conforming, so you may apply to *x* any of the features of *T* — rather than being limited, as in the unconstrained case *C [G]*, to the features of class *ANY*, applicable to all types.

A small syntactic extension enables us to take advantage of constrained genericity to allow creation of objects of generic type. Declare the class as

```
class D [G → CONST create cp1, cp2, ... end] ...
```

← “*CONSTRAINED GENERICITY*”, 12.6, page 346,



to state that  $G$  represents any type that both:

- (As always with constrained genericity) conforms to  $CONST$ .
- Admits as creation procedures its versions of  $cp1, cp2, \dots$ , which must be procedures of  $CONST$ .

These obligations are enforced: a generic derivation  $D [T]$  will only be valid if (as always)  $T$  conforms to  $CONST$  and, in addition, the given procedures  $cp1, cp2, \dots$  are creation procedures of  $T$ . More precisely, their **versions** in  $T$  — which may differ from the originals versions in  $CONST$  as a result of renaming, redefinition and effecting — must be listed among the creation procedures of  $T$ .

With  $D$  declared as shown, it becomes possible, for  $x$  declared of type  $G$  in the text of class  $D$  itself, to use a creation instruction

```
create x.cpi (args)
```

where  $cp_i$  is one of the procedures of  $D$  listed in the **create ... end** part for  $CONST$  as shown above, and  $args$  is a valid argument list for that procedure. The instruction will always make sense dynamically since, thanks to the preceding rule, the type  $T$  of  $x$  — in any valid generic derivation  $D [T]$  — will always be a descendant of  $CONST$ , so that:

- $cp_i$  will be one of its procedures, taking the appropriate arguments.
- $T$  will have listed  $cp_i$  as one of its creation procedures (hence, among other properties, we may expect that  $cp_i$  ensures the invariant of  $T$ ).

As a special case, you can permit the procedure-less form **create**  $x$  by including *default\_create* (rather, its name in  $CONST$ ) among the  $cp_i$ .

What's particularly useful in this mechanism is that at the level of  $D$  we only require the listed  $cp_i$  to be **procedures** of the constraining type  $CONST$  — so that we can ascertain, from  $D$ 's text only, the validity of  $args$  as arguments in the creation call **create**  $x.cp_i (args)$ : we do not require the  $cp_i$  to be **creation procedures** of  $CONST$ . This last requirement will only come up where it matters: in types  $T$ , descendants of  $CONST$  used in actual generic derivations  $D [T]$ . In such a  $T$ , the local version of  $cp_i$  must indeed be one of  $T$ 's creation procedures.

This means in particular that the above scheme will work even if *CONST* is deferred, as in



```

class
  D [G → CONST create cp end]
feature
  some_routine
  local
    x: G
  do
    create x.cp (3)
  end
end
deferred class CONST feature
  cp (n: INTEGER)
  ... Could be effective or deferred ...
  end
  ... Other features, possibly including deferred ones ...
end

```

We don't care that the boxed creation instruction works on a target  $x$  whose type  $G$  is based on a deferred class *CONST*, and that the creation procedure *cp* might itself be deferred in *CONST*: any type  $T$  used for  $G$  in practice must make its version of *cp* a creation procedure. This implies among other things that  $T$  is an effective class and *cp* an effective procedure, so everything will work properly.



Note that this creation mechanism for formal generics assumes *constrained* genericity. In a class  $C [G]$ , where  $G$  is an *unconstrained* generic parameter, no creation instruction **create**  $x \dots$  is valid for  $x$  of type  $G$ . This includes the procedure-less form **create**  $x$ : making it valid would mean assuming that *default\_create* will be a creation procedure in all possible types — certainly not true. You can, however, write the class as

```

class C [G → ANY create default_create end]

```

thereby unfolding unconstrained genericity into its constrained equivalent. Then the generic derivation  $C [T]$  will be valid for a type  $T$  if and only if  $T$ 's base class doesn't list any creation procedures, or lists *default\_create* among its creation procedures. With this form of  $C$ 's declaration, **create**  $x$  is valid in the text of class  $C$ .



More generally, remember that the procedure-less form `create x` is only valid, for  $x$  of a formal generic type, if you have explicitly listed `default_create` (under its local name) in a `create` subclause after the constraint. There is no equivalent here to the implicit rule of the `Creators` part, where requesting no creation procedures means requesting `default_create` only. For generic parameters, you don't get creation privileges unless you specify them expressly.

## 20.10 PRECONDITIONS OF CREATION PROCEDURES

The creation process, when it involves a creation procedure, applies it to an object caught in its virginal state, just after default initializations. Such a state does not, in general, satisfy the class invariant; it is indeed the very purpose of the creation procedure to ensure the invariant from the first time.

A consequence of dealing with an object in such a fragile temporary state is that the creation procedure must refrain, if it has a precondition, from including in it certain properties that are meaningful only in later stages of the object's life. In particular

- The precondition should not use any feature of the object, since the client could not legitimately access the value of that feature to ensure the precondition. Assume for example a creation procedure `cp` with a precondition clause  $a > 0$  where  $a$  is an attribute; the client should be able, before a creation instruction `create x.cp (...)`, to test for  $x.a > 0$ , but this makes no sense since the required object doesn't exist yet. So we must prohibit the use of any `Unqualified_call`, to a feature of any kind, in the precondition.
- For the same reason, we must prohibit any use of `Current`, denoting a current object that doesn't exist yet.

The precondition can still refer to any properties of the creation procedure's arguments, including through feature calls on these arguments.

In addition, we have a requirement similar to the general rule for feature availability in feature calls. That rule specified that any feature  $p$  used in the precondition of a feature  $f$  must be available to all the clients to which  $f$  itself is available, so that any client that may call  $x.f(...)$  may also check for  $x.p$ . In the case of a creation instruction `create x.cp (...)`, we have seen that a creation procedure `cp` must be "available for creation" to the client; to any such client,  $p$  has to be available (for call). This is a new requirement since it is possible for `cp` to be "available for creation" to a client, but not available for call.

← *Precondition Export rule: VAPE, page 233.*

← *"RESTRICTING CREATION AVAILABILITY", 20.7, page 531*

These observations lead to a rule on the precondition clauses of any routine used as a creation procedure:



**Creation Precondition rule** *VGCP*

A **Precondition** of a routine *r* is **creation-valid** if and only if its unfolded form *uf* satisfies the following conditions:

- 1 • The predefined entity **Current** does not appear in *uf*.
- 2 • No **Unqualified\_call** appears in *uf*.
- 3 • Every feature whose final name appears in the *uf* is available to every class to which *r* is available for creation.

This definition is not itself a validity constraint, but is used by condition 5 of the Creation Clause rule below; giving it a code as for a validity constraint enables compilers to provide a precise error message in case of a violation. → *VGCC, page 540.*

Requiring preconditions to be creation-valid will ensure that a creation procedure doesn't try to access, in the object being created, fields whose properties are not guaranteed before initialization.

The definition relies on the “unfolded form” of an assertion, which reduces it to a boolean expression with clauses separated by **and then**. Because the unfolded form uses the Equivalent Dot Form, condition 3 also governs the use of operators: with *plus alias* “+”, the expression *a + b* will be acceptable only if the feature *plus* is available for creation as stated.

## 20.11 CREATION SYNTAX AND VALIDITY



Here now are the precise rules applying to **Creators** parts and **Creation** instructions. This section only formalizes previously introduced concepts, so on first reading you may skip this section and the next two (which formalize the semantics). *If skipping, go to “CREATION EXPRESSIONS AND ANONYMOUS OBJECTS”, 20.14, page 550.*

First, the syntax of a **Creators** part, an optional component of the **Class** text, appearing towards the beginning of a class, after **Inheritance** and before **Features**: *The structure of a Class text, with all its parts, is on page 119.*



**Creators parts**

$$\text{Creators} \triangleq \text{Creation\_clause}^+$$

$$\text{Creation\_clause} \triangleq \text{create} [\text{Clients}]$$

$$\qquad\qquad\qquad [\text{Header\_comment}]$$

$$\qquad\qquad\qquad \text{Creation\_procedure\_list}$$

$$\text{Creation\_procedure\_list} \triangleq \{\text{Creation\_procedure} \text{ , } \dots \}^+$$

$$\text{Creation\_procedure} \triangleq \text{Feature\_name}$$

*The optional Header\_comment emphasizes the similarity with the syntax of a Feature\_clause, given page 137.*

To talk about the validity and semantics of creation clauses and creation instructions, it is useful to take care once and for all of the special case of *default\_create* as creation procedure through the following definition:



### Unfolded Creators part of a class

The **unfolded creators part** of a class *C* is a **Creators** defined as:

- 1 • If *C* has a **Creators** part *c*: *c*.
- 2 • If *C* is deferred: an empty **Creators** part.
- 3 • Otherwise, a **Creators** part built as follows, *dc\_name* being the final name in *C* of its version of *default\_create* from **ANY**:  
     **create**  
         *dc\_name*

For generality the definition is applicable to any class, even though for a deferred class (case 2) it would be invalid to include a **Creators** part. This causes no problem since the rules never refer to a deferred class actually extended with its unfolded creators part.

Case 3 reflects the convention that an absent **Creators** part stands for **create** *dc\_name* — normally **create** *default\_create*, but *dc\_name* may be another name if the class or one of its proper ancestors has renamed *default\_create*.

← Discussed informally in previous sections.

With this we can define the constraint on **Creators** part of a class:



### Creation Clause rule

VGCC

A **Creation\_clause** in the unfolded creators part of a class *C* is valid if and only if it satisfies the following conditions, the last four for every **Feature\_name** *cp\_name* in the clause's **Feature\_list**:

- 1 • *C* is effective.
- 2 • *cp\_name* appears only once in the **Feature\_list**.
- 3 • *cp\_name* is the final name of some procedure *cp* of *C*.
- 4 • *cp* is not a once routine.
- 5 • The precondition of *cp*, if any, is creation-valid.

As a result of conditions [1](#) and [4](#), a creation procedure may only be of the **do** form (the most common case) or **External**.

The prohibition of **once** creation procedures in condition [4](#) is a consequence of the Creation principle: with a once procedure, the first object created would satisfy the invariant (assuming the creation procedure is correct), but subsequent creation instructions would not execute the call, and hence would limit themselves to the default initializations, which might not ensure the invariant.

As a corollary of condition [4](#), a class that has no explicit **Creators** part may not redefine *default\_create* into a once routine, or inherit *default\_create* as a once routine from one of its deferred parents. (Effective parents would themselves violate the condition and hence be invalid.)

Condition [5](#) is the rule on preconditions of creation procedures, whose rationale was discussed in the preceding [section](#).

← “[PRECONDITIONS OF CREATION PROCEDURES](#)”, [20.10](#), [page 538](#).



To complement this study of the syntax and semantics of **Creators** parts, it is useful to remind ourselves of their counterpart for generic parameters: the **Constraint\_creators** subclause of the syntax for generic constraints, a simplified form of the **Creators** part. Here is the relevant syntax:



```

Formal_generics  $\triangleq$  "["Formal_generic_list"]"
Formal_generic_list  $\triangleq$  [Formal_generic", "...]
Formal_generic  $\triangleq$  [frozen] Formal_generic_name
                    [Constraint]
Formal_generic_name  $\triangleq$  Identifier
Constraint  $\triangleq$  "->"Class_type [Constraint_creators]
Constraint_creators  $\triangleq$  create Feature_list end
  
```

← This was first seen in the chapter on types; syntax on [page 343](#), validity in “[CONSTRAINED GENERICITY](#)”, [12.6](#), [page 346](#).

The applicable validity rule there was that the elements of the **Feature\_list** must be the names of distinct procedures of the constraining type — corresponding to clauses [1](#) and [2](#) of the Creation Clause rule above. There was no need for an equivalent to the other clauses since they are taken care of by the Creation Clause rule itself when we provide an actual generic parameter conforming to the constraining type.



A language design note: it would have been possible to use **Creators** for **Constraint\_creators**, permitting a more flexible form of creation availability specification for a generic parameter — with more than one **Creation\_clause**, each listing specific clients and procedures. This would in fact make the language definition simpler by avoiding the construct **Constraint\_creators**. The extra capabilities, however, seems useless, and could yield unduly complicated **Formal\_generics** parts, so the language sticks to a primitive form of **Constraint\_creators** for generic parameters.

The Creation Clause rule allows us to define the set of creation procedures of a class:



### Creation procedures of a class

The **creation procedures** of a class are all the features appearing in any **Creation\_clause** of its unfolded creators part.

If there is an explicit **Creators** part, the creation procedures are the procedures listed there. Otherwise there is only one creation procedure: the class's version of *default\_create*.

The following property is a consequence of the definitions of “unfolded creators part” and “creation procedures of a class”.

### Creation procedure property

An effective class has at least one creation procedure.

Those explicitly listed if any, otherwise *default\_create*.

Only in the first case (explicit **Creators** part) can the set of creation procedures be empty: this is achieved, as we have seen, by including a **Creators** part, but an empty one, listing no name at all.

← See the example class *NOT\_INSTANTIABLE* on page 525.

We need a small refinement of this definition to extend it to the case of types, to support the mechanism for creation on generic parameters:

← See “*CREATING INSTANCES OF FORMAL GENERICS*”, 20.9, page 535.



### Creation procedures of a type

The **creation procedures** of a type *T* are:

- 1 • If *T* is a **Formal\_generic\_name**, the constraining creators for *T*.
- 2 • Otherwise, the creation procedures of *T*'s base class.



The definition of case 2 is not good enough for case 1, because in the scheme **class *D* [*G* → *CONST* create *cp1*, *cp2*, ... end]** it would give us, as creation procedures of *G*, the creation procedures of *CONST*, and what we want is something else: the set of procedures *cp1*, *cp2*, ... specifically listed after *CONST* — the “*constraining creators for G*”. These are indeed procedures of *CONST*, but they are not necessarily *creation* procedures of *CONST*, especially since *CONST* can be deferred. What matters is that they must be creation procedures in any instantiatable descendant of *CONST* used as actual generic parameter for *G*.

Other useful definitions:

DEFINITION

### Available for creation; general creation procedure

A creation procedure of a class *C*, listed in a *Creation\_clause* *cc* of *C*'s unfolded creators part, is **available for creation** to the descendants of the classes given in the *Clients* restriction of *cc*, if present, and otherwise to all classes.

If there is no *Clients* restriction, the procedure is said to be a **general creation procedure**.

*As with a Feature\_clause, the absence of a Clients restriction is equivalent to a restriction of the form {ANY}.*



Remember, once again, that the descendants of a class include the class itself. A *Creation\_clause* with no *Clients* part, as in **create** *cp1*, *cp2*, ..., is a shortcut for one with a *Clients* part listing only *ANY*, as in **create** {*ANY*} *cp1*, *cp2*, ...

Now for the *Creation\_instruction* itself, starting with its syntax:

SYNTAX

### Creation instructions

*Creation\_instruction*  $\triangleq$  **create** [*Explicit\_creation\_type*]  
*Creation\_call*

*Explicit\_creation\_type*  $\triangleq$  "{" *Type* "}"

*Creation\_call*  $\triangleq$  *Variable* [*Explicit\_creation\_call*]

*Explicit\_creation\_call*  $\triangleq$  "." *Unqualified\_call*

Every creation instruction has a *creation type*, explicit or implicit:

DEFINITION

### Creation target, creation type

The **creation target** (or just "target" if there is no ambiguity) of a *Creation\_instruction* is the *Variable* of its *Creation\_call*.

The **creation type** of a creation instruction, denoting the type of the object to be created, is:

- The *Explicit\_creation\_type* appearing (between braces) in the instruction, if present.
- Otherwise, the type of the instruction's target.

so that in



```
account1: ACCOUNT; point1, point2: POINT; figure1: FIGURE
...
create account1
create point1.make_polar (1, Pi/4)
create {SAVINGS_ACCOUNT} account1
create {SEGMENT} figure1.make (point1, point2)
```

the creation types for the four instructions are *ACCOUNT*, *POINT*, *SAVINGS\_ACCOUNT* and *SEGMENT*. The targets are *account1*, *point1*, *account1* and *figure1*.

The creation type of a *Creation\_instruction* is the type of the objects that it may create. It will always satisfy the following property:

### Creation Type theorem

The creation type of a creation instruction is always effective.

This theorem is corollary 1 of the Creation Instruction rule, seen next. That rule will need one more auxiliary definition: → *The corollary is on page 547.*



### Unfolded form of a creation instruction

Consider a *Creation\_instruction* *ci* of creation type *CT*. The **unfolded form** of *ci* is a creation instruction defined as:

- 1 • If *ci* has an *Explicit\_creation\_call*, then *ci* itself.
- 2 • Otherwise, a *Creation\_instruction* obtained from *ci* by making the *Creation\_call* explicit, using as feature name the final name in *CT* of *CT*'s version of *ANY*'s *default\_create*.

This definition parallels the earlier one of “unfolded creators part of a class” and expresses the property, stated informally before, that we understand the procedure-less form of creation **create** *x* as a shortcut for **create** *x.default\_create* (with the new name for *default\_create* if different).



A final notion that the Creation Instruction rule will need is a property defined only in a subsequent chapter, but already presented informally in the discussion of calls, and in fact rather obvious: the concept of a call being **argument-valid**. This property is part of the more complete definition of call validity; it states that in a call  $x.f(a, b, c)$  where *x* is of type *T* and *f* is a feature of *T* with formal arguments  $u1: T1; u2: T2; u3: T3$ , → *“Argument rule”*, page 626.:

the number of actual arguments  $a, b, c$  must be the same as the number of these formal arguments, here three, and each actual's type must conform to the corresponding formal's type — here the type of  $a$  to  $T1$ , of  $b$  to  $T2$ , and of  $c$  to  $T3$ . We of course expect this fundamental property to hold for all calls, and must enforce it for a creation instruction `create x.f(a, b, c)` involving a `Creation_call`. This is clause 3 of the following rule.

We indeed by now have enough preparation to express the validity rule for creation instructions:



### Creation Instruction rule

VGCI

A `Creation_instruction` of creation type  $CT$ , appearing in a class  $C$ , is valid if and only if it satisfies the following conditions:

- 1 •  $CT$  conforms to the target's type.
- 2 • The feature of the `Creation_call` of the instruction's unfolded form is available for creation to  $C$ .
- 3 • That `Creation_call` is argument-valid.
- 4 •  $CT$  is generic-creation-ready.

→ Another version of this rule appears below, page 547, with clauses labeled by numbers rather than letters.

I can see that puzzled look on your face: surely, with all the possibilities seen in this chapter, the complete validity constraint for creation instructions must be longer?

In spite of its compactness, the Creation Instruction rule suffices in fact to capture all properties of creation instructions thanks to the auxiliary definitions of “creation type”, “unfolded form” of both a `Creation_instruction` and a `Creators` part, “available for creation” and others. The rule captures in particular the following cases:

- The procedure-less form `create x` is valid only if  $CT$ 's version of `default_create` is available for creation to  $C$ ; this is because in this case the unfolded form of the instruction is `create x.dc_name`, where `dc_name` is  $CT$ 's name for `default_create`. On  $CT$ 's side the condition implies that there is either no `Creators` part (so that  $CT$ 's own unfolded form lists `dc_name` as creation procedure), or that it has one making it available for creation to  $C$  (through a `Creation_clause` with either no `Clients` specification or one that lists an ancestor of  $C$ ).
- If  $CT$  is a `Formal_generic_name`, its creation procedures are those listed in the `create` subclause after the constraint. So `create x` is valid if and only if the local version of `default_create` is one of them, and `create x.cp (...)` only if `cp` is one of them.
- If  $CT$  is generically derived, and its base class needs to perform creation operations on targets of some of the formal generic types, the last condition (generic-creation readiness) ensures that the corresponding actual parameters are equipped with appropriate creation procedures.

The very brevity of this rule may make it less suitable for one of the applications of validity constraints: enabling compilers to produce precise diagnostics in case of errors. For this reason a complementary rule, conceptually redundant since it follows from the Creation Instruction rule, but providing a more explicit view, appears next. It is stated in “*only if*” style rather than the usual “*if and only if*” of other validity rules, since it limits itself to a set of necessary validity conditions.

All together, these conditions do come close to the full set of sufficient conditions listed in the first variant, but we don’t really care, since that first variant gives us the “*if and only if*” property that we need.



### Creation Instruction properties VGCP

A Creation\_instruction *ci* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a Formal\_generic\_name and calling *BCT* the base class of *CT* and *dc* the version of *ANY*'s *default\_create* in *BCT*:

- 1 • *BCT* is an effective class.
- 2 • If *ci* includes a Type part, the type it lists (which is *CT*) conforms to the type of the instruction's target.
- 3 • If *ci* has no Creation\_call, then *BCT* either has no Creators part or has one that lists *dc* as one of the procedures available to C for creation.
- 4 • If *BCT* has a Creators part which doesn't list *dc*, then *ci* has a Creation\_call.
- 5 • If *ci* has a Creation\_call whose feature *f* is not *dc*, then *BCT* has a Creators part which lists *f* as one of the procedures available to *C* for creation.
- 6 • If *ci* has a Creation\_call, that call is argument-valid.

If *CT* is a Formal\_generic\_name, the instruction is valid only if it satisfies the following conditions:

- 7 • *CT* denotes a constrained generic parameter.
- 8 • The Constraint for *CT* specifies one or more procedures as constraining creators.
- 9 • If *ci* has no Creation\_call, one of the constraining creators is the Constraint's version of *default\_create* from *ANY*.
- 10 • If *ci* has a Creation\_call, one of the constraining creators is the feature of the Creation\_call.

**WARNING:** although this rule looks complicated, it is in fact just a series of consequences of a short and simple rule: the original "VGCI", page 545.

Compiler writers may refer, in error messages, to either these "Creation Instruction Properties" or the earlier "Creation Instruction rule" of which they are consequences. For the language definition, **the official rule is the Creation Instruction rule**, which provides a necessary and sufficient set of validity conditions.



The number of clauses in this second variant justifies *a contrario* using the first variant as the official definition. Fundamentally, the rule is straightforward once you have defined the "creation type", explicit or implicit and "unfolded" both the creation instruction and the creation type's base class to take care of the *default\_create* convention, so that every class has a list of creation procedures and every creation instruction lists a creation procedure. Then the rule is simply that the creation type must be OK for the creation's

target, that the creation procedure must be available for creation, and that the call must have valid arguments. That's all. The “corollaries” form is long because it expands the various simplifications (creation type, creation procedures of a class, creation procedure of an instruction) for the various possible cases, and treats all these cases individually — accounting for various errors that an absent-minded developer might make.

## 20.12 CREATION SEMANTICS

**SEMANTICS** With the preceding validity rules, we can define the precise semantics of a `Creation_instruction`.

### Creation Instruction Semantics

The effect of a creation instruction of target  $x$  and creation type  $TC$  is the effect of the following sequence of steps, in order:

- 1 • If there is not enough memory available for a new direct instance of  $TC$ , trigger an exception of type  $NO\_MORE\_MEMORY$  in the routine that attempted to execute the instruction. The remaining steps do not apply in this case.
- 2 • Create a new direct instance of  $TC$ , with reference semantics if  $CT$  is a reference type and copy semantics if  $CT$  is an expanded type.
- 3 • Call, on the resulting object, the feature of the Unqualified\_call of the instruction's unfolded form.
- 4 • Attach  $x$  to the object.

← See [19.3, page 498](#) about a reference being attached to an object.

The rules requires the *effect* described by this sequence of steps; it does not require that the implementation literally carry out the steps. In particular, if the target is expanded and has already been set to an object value, the implementation (in the absence of cycles in the client relation between expanded classes) may **not have to allocate new memory**; instead, it may be able simply to reuse the memory previously allocated to that object. (Because only expanded types conform to an expanded type, no references may exist to the previous object, and hence it is not necessary to preserve its value.) In that case, there will always at step 1 be “enough memory available for a new direct instance” — the memory being reused — and so the exception cannot happen.

One might expect, between steps 2 and 3, a step of *default initialization* of the fields of the new object, since this is the intuitive semantics of the language: integers initialized to zero, detachable references to void etc. There is no need, however, for such a step since the Variable Semantics rule → Page [513](#). implies that an attribute or other variable, unless previously set by an explicit attachment, is automatically set on first access. The rule implies for example that an integer field will be set to zero. More generally, the semantics of the language guarantees that in every run-time circumstance any object field and local variable, even if never explicitly assigned to yet, always has a well-defined value when the computation needs it.

About step 3, remember that the notion of “unfolded form” allows us to consider that every creation instruction has an Unqualified call; in the procedure-less form **create** *x*, this is a call to *default\_create*. ← Page [544](#).

Also note the order of steps: attachment to the target *x* is the last operation. Until then, *x* retains its earlier value, void if *x* is a previously unattached reference.

## 20.13 REMOTE CREATION



The syntax of creation instructions does not support “remote creation” instructions as in:

```
create x1.y1.cp (...)
```

**WARNING:** syntactically incorrect.

To obtain an equivalent effect, assuming that *x1* is of type *X* and that *y1* is an attribute of type *Y* in *X*, you must introduce a specific procedure in *X*



```
make_y1 (arguments: ...)
  -- Attach y1 to new instance of Y.
do
  create y1.cp (arguments)
end
```

so that instead of the above attempt at remote creation clients will use the instruction

```
x1.make_y1 (...)
```



This is in line with the principle of information hiding: deciding whether or not clients of *X* may directly “create” the *yl* field is the privilege of the designer of *X* who, if the answer is positive, will write a specific procedure to grant this privilege — restricting its availability if desired.

## 20.14 CREATION EXPRESSIONS AND ANONYMOUS OBJECTS

We have seen all there is to see about creation instructions, but there remains to study a variant of the mechanism: creation *expressions*.

Creation expressions will provide us with *anonymous objects*. The objects that we produce with a creation instruction **create** *x*... have a name — *x* — in the software text. This is usually what we want, because after we have created the object we will start manipulating it in the same routine, or others of the same class. But in some cases the name is useless because all we do with the newly created object is to pass it to another software element. Having to declare a local variable *x* just for the purpose of a creation instruction is a nuisance. A small nuisance to be sure, but whatever the language can do to avoid writing useless elements will be good for the quality of your software and your schedule.

We saw an example of such a situation when examining the clonable array technique. We had the following scheme



```
figure_factory: ARRAY [FIGURE]
  local
    fig: FIGURE
  once
    Result.make (Low_id, High_id)
        -- Create and enter a SEGMENT instance:
    create {SEGMENT} fig.make (...)
    Result.put (fig, Segment_id)
        -- Create and enter a TRIANGLE instance:
    create {TRIANGLE} fig.make (...)
    Result.put (fig, Triangle_id)
    ... Do the same for each variant ...
  end
```

“Language terseness and family vacations”, in *SPOOF 84 (Sociology and Psychology of Object-Oriented Fanatics)*, Martha’s Vineyard, 1999, pp. 6574-6598.

← This was example 4, page 530. See simpler formulation next.

All we use *fig* for is to create successive objects — instances of descendants of *FIGURE*. But as soon as we have produced such an object with a creation instruction, we store it into the corresponding entry of the *Result* array (by passing it to the corresponding assignment procedure), and we will never, in this routine, need the object again! This is why we can reuse the same local variable, *fig*, for every *FIGURE* variant.

In this case the entity *fig* is not needed; neither is a separate creation instruction. All we really want is an expression denoting the new object, which we can directly pass to a routine or, as here, assign to an array element.

Creation expressions serve this need. They look like one of

[1]	<b>create</b> { <i>SOME_TYPE</i> }
[2]	<b>create</b> { <i>SOME_TYPE</i> }. <i>creation_procedure</i> (...)

The first variant, as you have guessed, is applicable if *SOME\_TYPE*'s base class has no *Creators* part, or one that includes *default\_create*; the second, if *creation\_procedure* is one of its creation procedures.

Note how both variants look like a *Creation\_instruction*:

- The first recalls the instruction **create** {*SOME\_TYPE*} *target*, with no explicit *Creation\_call*.
- The second recalls **create** {*SOME\_TYPE*} *target*.*creation\_procedure* (...).

You see the idea: starting from a creation instruction, you will get a creation expression simply by removing the *target* — a natural convention, since what you want is an anonymous object.

The constructs given (in any of the two forms [1] and [2]) are **expressions**, denoting values that can be assigned to a *Variable* entity, as in



$x :=$ <b>create</b> { <i>SEGMENT</i> }. <i>make</i> ( <i>point1</i> , <i>point2</i> )
--

or, more commonly, passed as arguments to a routine, as in



<i>segment_operation</i> ( <b>create</b> { <i>SEGMENT</i> }. <i>make</i> ( <i>point1</i> , <i>point2</i> ))
---

*Expression form.*

which has exactly the same effect as



<b>create</b> { <i>SEGMENT</i> } <i>seg</i> . <i>make</i> ( <i>point1</i> , <i>point2</i> ) <i>segment_operation</i> ( <i>seg</i> )
--

*Instruction form.*

with *seg* declared of type *FIGURE* (or directly of the ancestor type *SEGMENT*, in which case we can write the first line as just **create** *seg.make (point1, point2)*). With the creation expression we write a single call instead of three components — the declaration of *seg*, the creation instruction, and the call.

A difference with creation instructions is that for creation expressions you may not omit the *Explicit\_creation\_type*, *SOME\_TYPE* or *SEGMENT* in the examples above. This is precisely because the created objects are anonymous. In the instruction **create target ...**, if no type is specified, we use as creation type the type of *target*; but for a creation expression there is no named *target*, so you **must** specify *{SOME\_TYPE}* in all cases.

Here is the clonable array extract rewritten with creation expressions:



```
figure_factory: ARRAY [FIGURE]
  once
    Result.make (Low_id, High_id)

    -- Create and enter an instance of each desired kind:
    Result.put (create {SEGMENT}.make (...), Segment_id)
    Result.put (create {TRIANGLE}.make (...), Triangle_id)
    ... Similarly for each variant ...
  end
```

← The original was example 4, page 530, repeated above on page 550. To use the array, use clone operations; see

The comparison with the original form clearly shows the advantage of creation expressions in such a case. It's not so much a matter of writing *less*, since Eiffel is happy to be verbose when needed, as when specifying type properties of every entity, or expressing clear control structures. Rather, it's about avoiding elements that bring no useful information and can in fact, through their verbosity, obscure the text.



Note, however, that creation expressions are useful only in the special case of creating an object for the sole purpose of passing it to another software element, without using it further in the given routine. In every other situation — that is to say, in the vast majority of object creation needs — you should use a creation *instruction*.

Do not then be misled by the observation that you can rewrite any creation instruction

[1]  
`create x...`

*Instruction form.*

as

[1]  
`x := create {X_TYPE} ...`

*Expression form.*  
*WARNING: this is not the recommended style.*

If you are going to do anything else with  $x$ , you should stay with the first form. In any case it saves you the need to specify  $X\_TYPE$ , which you have already specified as the type of  $x$  in its declaration.

In summary: reserve creation expressions for anonymous objects. This important methodological note is in line with the general Eiffel principle that the language should provide *one* good way to address any specific need. Both creation expressions and creation instructions are useful, each appropriate in a different situation.

The syntax, validity and semantics of creation expressions will now follow without further comment, since they are directly deduced from the corresponding properties of creation instructions.



### Creation expressions

$\text{Creation\_expression} \triangleq \text{create Explicit\_creation\_type}$   
 $[\text{Explicit\_creation\_call}]$

← *Explicit\\_creation\\_type*, was defined on page 543 as {Type}.

The concepts introduced for creation instructions transpose directly here:

### Properties of a creation expression

The **creation type** and **unfolded form** of a creation expression are defined as for a creation instruction.

The validity rule is also similar:

← *“Creation Instruction rule”, page 545.*



### Creation Expression rule VGCE

A *Creation\_expression* of creation type  $CT$ , appearing in a class  $C$ , is valid if and only if it satisfies the following conditions:

- 1 • The feature of the *Creation\_call* of the expression’s unfolded form is available for creation to  $C$ .
- 2 • That *Creation\_call* is argument-valid.
- 3 •  $CT$  is generic-creation-ready.

Here too it is useful to have an “only if” version:



### Creation Expression Properties VGCX

A Creation\_expression *ce* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a Formal\_generic\_name and calling *BCT* the base class of *CT* and *dc* the version of *ANY*'s default\_create in *BCT*:

- 1 • *BCT* is an effective class.
- 2 • If *ce* has no Explicit\_creation\_call, then *BCT* either has no Creators part or has one that lists *dc* as one of the procedures available to C for creation.
- 3 • If *BCT* has a Creators part which doesn't list *dc*, then *ce* has an Explicit\_creation\_call.
- 4 • If *ce* has an Explicit\_creation\_call whose feature *f* is not *dc*, then *BCT* has a Creators part which lists *f* as one of the procedures available to C for creation.
- 5 • If *ce* has an Explicit\_creation\_call, that call is argument-valid.

If *CT* is a Formal\_generic\_name, the expression is valid only if it satisfies the following conditions:

- 6 • *CT* denotes a constrained generic parameter.
- 7 • The Constraint for *CT* specifies one or more procedures as constraining creators.
- 8 • If *ce* has no Creation\_call, one of the constraining creators is the Constraint's version of default\_create from *ANY*.
- 9 • If *ce* has a Creation\_call, one of the constraining creators is the feature of the Creation\_call.

**WARNING:** a more concise form of this rule appears just before.

← See “Creation Instruction properties”, page 547.

As with the corresponding “Creation Instruction Properties”, this is not an independent rule but a set of properties following from previous constraints, expressed with more detailed requirements that may be useful for error reporting by compilers. ← “Creation Instruction properties”, page 547.



Finally, the semantics:

### Creation Expression Semantics

The value of a creation expression of creation type *TC* is — except if step 1 below triggers an exception, in which case the expression has no value — a value attached to a new object as can be obtained through the following sequence of steps:

- 1 • If there is not enough memory available for a new direct instance of *TC*, trigger an exception of type *NO\_MORE\_MEMORY* in the routine that attempted to execute the expression. In this case the expression has no value and the remaining steps do not apply.
- 2 • Create a new direct instance of *TC*, with reference semantics if *CT* is a reference type and copy semantics if *CT* is an expanded type.
- 3 • Call, on the resulting object, the feature of the Unqualified\_call of the expression's unfolded form.

The notes appearing after the Creation Instruction Semantics rule also ← [“Creation Instruction Semantics”, , page 548.](#)

