Eiffel: Analysis, Design and Programming

Bertrand Meyer
(Nadia Polikarpova)
Overview

- Basics
- Redefinition
- Polymorphism and dynamic binding
- Inheritance and contracts
- Deferred classes
- Polymorphic containers
- Inheritance and genericity: constrained genericity
- Multiple inheritance
- Non-conforming inheritance
- Covariance and anchored types
Extending the basic notion of class

- Abstraction
- Inheritance
- Genericity
- Type parameterization
- Specialization
Inheritance basics

Principle:
Describe a new class as extension or specialization of an existing class
(or several with \textit{multiple} inheritance)

If $B$ inherits from $A$:

- **As modules**: all the services of $A$ are available in $B$
  (possibly with a different implementation)

- **As types**: whenever an instance of $A$ is required, an instance of $B$ will be acceptable
  ("is-a" relationship)
Terminology

If \( B \) inherits from \( A \) (by listing \( A \) in its \textit{inherit} clause):
- \( B \) is an \textbf{heir} of \( A \)
- \( A \) is a \textbf{parent} of \( B \)

For a class \( A \):
- The \textbf{descendants} of \( A \) are \( A \) itself and (recursively) the descendants of \( A \)'s heirs
- \textbf{Proper descendants} exclude \( A \) itself

Reverse notions:
- \textbf{Ancestor}
- \textbf{Proper ancestor}

More precise notion of instance:
- \textbf{Direct instances} of \( A \)
- \textbf{Instances} of \( A \): the direct instances of \( A \) and its descendants

(Other terminology: \textit{subclass, superclass, base class})
What you can do to inherited features

- Effect (implicit)
- Redefine
- Undefine
- Rename
- Change export status
- Select
Example hierarchy

FIGURE
  center*
  display*
  rotate*
  move*

OPEN FIGURE
  perimeter*

SEGMENT

POLYLINE

PERIMETER

POLYGON
  perimeter+
  perimeter++

TRIANGLE
  side1
  side2
  diagonal

RECTANGLE
  perimeter++

SQUARE
  perimeter++

CLOSED FIGURE
  perimeter+

ELLIPSE
  display+
  rotate+
  move+

CIRCLE
  display++
  rotate++
  move++

* deferred
+ effective
++ redefined
Redefinition

- Class may redefine ("override") inherited features
- Redefinition is explicit
- In line with Uniform Access principle, may redefine inherited function into attribute

Not the other way around!
Redefinition 1: polygons

class POLYGON inherit 
   CLOSED FIGURE
create
   make
feature
   vertex : ARRAY [POINT]
   vertex_count : INTEGER
   perimeter : REAL is
   -- Perimeter length
   do
       from ... until ... loop
       Result := Result + vertex [i] . distance (vertex [i + 1])
   ... 
   end
end
invariant
   vertex_count >= 3
   vertex_count = vertex.count
end
Redefinition 2: rectangles

```plaintext
class RECTANGLE inherit POLYGON

redefine
perimeter
end

create
make

feature
diagonal, side1, side2 : REAL

perimeter : REAL is
  -- Perimeter length
  do Result := 2 * (side1 + side2) end

invariant
  vertex_count = 4

end
```
Inheritance, typing and polymorphism

Assume:
\[
p: \text{POLYGON} ; \ r: \text{RECTANGLE} ; \ t: \text{TRIANGLE} \\
 x: \text{REAL}
\]

Permitted:
\[
x := p.\text{perimeter} \\
x := r.\text{perimeter} \\
x := r.\text{diagonal} \\
p := r
\]

NOT permitted:
\[
x := p.\text{diagonal} \quad \text{-- Even just after } p := r ! \\
r := p
\]
Dynamic binding

What is the effect of the following (if *some_test* is true)?

```plaintext
if some_test then
    p := r
else
    p := t
end

x := p.perimeter
```

**Redefinition:** A class may change an inherited feature, as with *POLYGON* redefining *perimeter*.

**Polymorphism:** *p* may have different forms at run-time.

**Dynamic binding:** Effect of *p.perimeter* depends on run-time form of *p*. 
Without dynamic binding?

```latex
\textit{display (f: FIGURE)}
\begin{align*}
\text{do} & \quad \text{if } \text{“f is a CIRCLE” then} \\
& \quad \quad \quad \quad \quad \quad \text{...} \\
& \quad \text{elseif } \text{“f is a POLYGON” then} \\
& \quad \quad \quad \quad \quad \quad \text{...} \\
& \quad \text{end} \\
& \text{end}
\end{align*}
```

and similarly for all other routines!

Tedious; must be changed whenever there’s a new figure type
With inheritance and associated techniques

With:

\[ f : \text{FIGURE} \]
\[ c : \text{CIRCLE} \]
\[ p : \text{POLYGON} \]

and:

\[ \text{create } c.\text{make}(\ldots) \]
\[ \text{create } p.\text{make}(\ldots) \]

Initialize:

\[ \text{if } \ldots \text{ then} \]
\[ f := c \]
\[ \text{else} \]
\[ f := p \]
\[ \text{end} \]

Then just use:

\[ f.\text{move}(\ldots) \]
\[ f.\text{rotate}(\ldots) \]
\[ f.\text{display}(\ldots) \]

-- and so on for every
-- operation on \( f \)!
Binding

Binding is the process in which

- A routine invocation $f(a)$, $x.f(a)$ is connected to code that is executed
- Accessing or writing to a variable
  
  ```
  a
  a := E
  ```

  is connected a memory location.
Static vs. dynamic binding

If binding is done at compile time, this is called “Static Binding” using “Static Linking”.

If binding is done at program load time, this is called “Static Binding” using “Dynamic Linking”.

If binding is done at runtime this is called “Dynamic Binding”.
Static Binding in C

In the .h header file:

```c
extern int foo (arg c);
extern float my_variable;
```

In the .c file:

```c
int foo (arg c);
float my_variable;
```
Execution of a call in C

On the caller side:

- Push the arguments on the stack
- Push the current \( PC + 1 \) on the stack (return address)
- Jump to the code to execute
  - This is “filled out” during linking/binding
- Clean the stack
Execution of a call in C (cont.)

On the callee side:
- Add some extra memory on the stack for the local variables
- Initialize the local variables (if necessary)
- Execute the implementation of the routine
- Read the return address from the stack
- Jump to the return address
Dynamic Binding in non-OO

Dynamic binding is not exclusive for object-orientation:

- **Lisp** (1958)
- **Forth** (1970) - a mixture called “Threaded Code”
- **C** (1970) - Why?
Dynamic Binding in C

```
#include <stdio.h>
void hello () {
    printf ("Hello\n");
}

void world () {
    printf ("World\n");
}

int main (int argc, char **argv) {
    void (*func)(void);

    if (argc > 1) func = hello;
    else func = world;

    func (); /* Dynamic Binding */
}
```
Dynamic Binding and OO

We need dynamic binding in object-orientation

Identifiers used in program text are relative to the current object:

- Attributes are relative to **Current**
- Routines may be redefined in subclasses
Using class tables

class A
feature
  a, b, c : INTEGER
  f is
    do
      c := a + b
    end
  g is ...
end

1. Read first field from Current
2. Add second field from Current
3. Store result in third field of Current
4. Go to the second entry in Current's class table
With inheritance

class \( B \)
inherit \( A \) redefine \( f \)
feature
  \( d: BOOLEAN \)
  \( f \) is ...
  \( h \) is ...
end
Contracts and inheritance

Issue: what happens, under inheritance, to

- Class invariants?
- Routine preconditions and postconditions?
Invariants

Invariant Inheritance rule:

- The invariant of a class automatically includes the invariant clauses from all its parents, “and”-ed.

Accumulated result visible in flat and interface forms.
Correct call in C:

\[
\text{if } a_1.\alpha \text{ then } \\
\text{ \quad } a_1.r(...) \\
\text{ \quad -- Here } a_1.\beta \text{ holds} \\
\text{end}
\]
Assertion redeclaration rule

When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition
A simple language rule does the trick!

Redefined version may have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

Resulting assertions are:
- `original_precondition or new_pre`
- `original_postcondition and new_post`
The role of deferred classes

Express abstract concepts independently of implementation

Express common elements of various implementations

Terminology: Effective = non-deferred (i.e. fully implemented)
A deferred feature

In e.g. **LIST**:

```plaintext
forth
require not after
deferred
ensure index = old index + 1
end
```
Mixing deferred and effective features

In the same class

\[ \text{search}(x: G) \]

-- Move to first position after current
-- where \( x \) appears, or after if none.

do
from until after or else \( item = x \)
end

“Programs with holes”
“Don’t call us, we’ll call you!”

A powerful form of reuse:

- The reusable element defines a general scheme
- Specific cases fill in the holes in that scheme

Combine reuse with adaptation
Deferred classes in EiffelBase

- CONTAINER
  - BOX
  - COLLECTION
    - BAG
    - SET
  - TRAVERSABLE
    - HIERARCHICAL
    - LINEAR
  - TRAVERSABLE
    - INFINITE
    - BOUNDED
    - UNBOUNDED
    - COUNTABLE
    - RESIZABLE
    - INDEXABLE
    - CURSOR_STRUCTURE
    - DISPELNER
    - SEQUENCE
    - TABLE
    - ACTIVE
    - INTEGER_INTERVAL
    - BILINEAR
    - ARRAY
    - STRING
    - HASH_TABLE
    - STACK
    - QUEUE

* deferred
Deferred classes for analysis

defered class
   VAT
inherit
   TANK
feature
   in_valve, out_valve: VALVE
fill is
       -- Fill the vat.
require
   in_valve.open
   out_valve.closed
defered ensure
   in_valve.closed
   out_valve.closed
   is_full
end
empty, is_full, is_empty, gauge, maximum, ... [Other features] ...

invariant
   is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)
end
Polymorphic data structures

```plaintext
class LIST[G]
feature
  ...
  last: G is ...
  extend (x: G) is ...
end

fl: LIST[FIGURE]
r: RECTANGLE
s: SQUARE
t: TRIANGLE
p: POLYGON
...
fl.extend (p); fl.extend (t); fl.extend (s); fl.extend (r)

from fl.start until fl.after loop fl.item.display; fl.forth end
```
Figure hierarchy (reminder)

- **center**
- **perimeter**
- **perimeter +
- **display**
- **rotate**
- **move**

* deferred
+ effective
++ redefined

**FIGURE**

* **OPEN_FIGURE**
  - **SEGMENT**
  - **POLYLINE**
  - ...
* **CLOSED_FIGURE**
  - **POLYGON**
  - **ELLIPSE**
  - **CIRCLE**

**SEGMENT**

**POLYLINE**

**POLYGON**

**TRIANGLE**

**RECTANGLE**

**SQUARE**

**ELLIPSE**

**CIRCLE**
Enforcing a type: the problem

```
fl.store("FILE_NAME")
...
    -- Two years later:
fl := retrieved("FILE_NAME")
x := fl.last       -- [1]
print(x.diagonal)   -- [2]
```

What's wrong with this?

- If `x` is declared of type `RECTANGLE`, [1] is invalid.
- If `x` is declared of type `FIGURE`, [2] is invalid.
Enforcing a type: the Object Test

if attached {RECTANGLE} fl.last as r then
  print (r.diagonal)
  ... Do anything else with r, guaranteed to be non-void
  ... and of dynamic type (descendant of) RECTANGLE
else
  print ("Too bad."")
end
Earlier mechanism: assignment attempt

\( f : \text{FIGURE} \)
\( r : \text{RECTANGLE} \)
...

\( f.l.\text{retrieve}("\text{FILE\_NAME}\") \)
\( f := f.l.\text{last} \)

\( r \neq f \)

if \( r \neq \text{Void} \) then
    \( \text{print}(r.\text{diagonal}) \)
else
    \( \text{print}("\text{Too bad.}\") \)
end
Assignment attempt

$x\ ?=\ y$

with

$x: A$

Semantics:

- If \( y \) is attached to an object whose type conforms to \( A \), perform normal reference assignment.
- Otherwise, make \( x \) void.
What we have seen

- Basics
- Redefinition
- Polymorphism and dynamic binding
- Inheritance and contracts
- Deferred classes
- Polymorphic containers
Topics for today

- Inheritance and genericity: constrained genericity
- Multiple inheritance
- Non-conforming inheritance
- Covariance and anchored types
Inheritance + Genericity

Unconstrained genericity

\[ \text{LIST}[G] \]

e.g. \( \text{LIST}[\text{INTEGER}], \text{LIST}[\text{PERSON}] \)

Constrained genericity

\[ \text{HASH\_TABLE}[G, H \rightarrow \text{HASHABLE}] \]

\[ \text{VECTOR}[G \rightarrow \text{NUMERIC}] \]
Constrained genericity

class VECTOR [G] feature
  plus alias "+" (other: VECTOR [G]): VECTOR [G]
  -- Sum of current vector and other.
  require
    lower = other.lower
    upper = other.upper
  local
    a, b, c: G
  do
    ... See next ...
  end
  ... Other features ...
end
Adding two vectors

\[ u + v = w \]

2 1
a + b = c
Constrained genericity

Body of \textit{plus alias "+"}:

\begin{verbatim}
create Result.make(lower, upper)
from
  i := lower
until
  i > upper
loop
  a := item(i)
  b := other.item(i)
  c := a + b -- Requires "+" operation on G!
  Result.put(c, i)
  i := i + 1
end
\end{verbatim}
The solution

Declare class \texttt{VECTOR} as

\begin{verbatim}
class VECTOR [G \rightarrow NUMERIC] feature
  ... The rest as before ...
end
\end{verbatim}

Class \texttt{NUMERIC} (from the Kernel Library) provides features \texttt{plus alias "+", minus alias "-"} and so on.
Improving the solution

Make \texttt{VECTOR} itself a descendant of \texttt{NUMERIC}, effecting the corresponding features:

\begin{verbatim}
class VECTOR [G \rightarrow NUMERIC] inherit NUMERIC
    feature
        ... Rest as before, including \texttt{infix "+"...}
    end
\end{verbatim}

Then it is possible to define

\begin{verbatim}
v : VECTOR [INTEGER]
v v : VECTOR [VECTOR [INTEGER]]
v v v : VECTOR [VECTOR [VECTOR [INTEGER]]]
\end{verbatim}
In the end...

Genericity is always constrained because

\[ \text{LIST}[G] \]

is just an abbreviation of

\[ \text{LIST}[G \rightarrow \text{ANY}] \]
Combining abstractions

Given the classes

- TRAIN_CAR, RESTAURANT

how would you implement a DINER?
Examples of multiple inheritance

Combining separate abstractions:

- Restaurant, train car
- Calculator, watch
- Plane, asset
- Home, vehicle
- Tram, bus
Composite figures
Multiple inheritance: Composite figures

Simple figures

A composite figure
Defining the notion of composite figure

center
display
hide
rotate
move
...

FIGURE

LIST [FIGURE]

count
put
remove
...

COMPOSITE FIGURE
In the overall structure
A composite figure as a list

Cursor

before

item

forth

after
Composite figures

```plaintext
class COMPOSITE_LINE inherit FIGURE
  LIST[FIGURE]
feature
  display
    do
      from start until after loop
        item.display
      forth
    end
end
... Similarly for move, rotate etc. ...
end

Requires dynamic binding
```
Going one level of abstraction higher

A simpler form of procedures *display, move* etc. can be obtained through the use of iterators

Use *agents* for that purpose
Multiple inheritance: Combining abstractions

- **COMPARABLE**: <, <=, >, >=, ...
- **INTEGER**
- **REAL**
- **STRING**
- **NUMERIC** +, -, *, /...
- **COMPLEX**

(total order relation)

(commutative ring)
The Java-C# solution

No multiple inheritance for classes

“Interfaces”: specification only (but no contracts)
  ➢ Similar to completely deferred classes (with no effective feature)

A class may inherit from:
  ➢ At most one class
  ➢ Any number of interfaces
Multiple inheritance: Combining abstractions

- COMPARABLE
- STRING
- INTEGER
- REAL
- NUMERIC
- COMPLEX

<, <=, >, >=, ...
(total order relation)

+, −, *, /'
(commutative ring)
How do we write COMPARABLE?

defered class COMPARABLE [G] feature

  less alias "<" (x: COMPARABLE [G]): BOOLEAN deferred end

  less_equal alias "<=" (x: COMPARABLE [G]): BOOLEAN do
    Result := Current < x or Current ~ x
  end

  greater alias ">" (x: COMPARABLE [G]): BOOLEAN do
    Result := x < Current end

  greater_equal alias ">=" (x: COMPARABLE [G]): BOOLEAN do
    Result := x <= Current end

end
Lessons from this example

Typical example of *program with holes*

We need the full spectrum from fully abstract (fully deferred) to fully implemented classes

*Multiple inheritance is there to help us combine abstractions*
Multiple inheritance: Name clashes
Resolving name clashes

rename \( f \) as \( A_f \)
Consequences of renaming

\[ a1: A \]
\[ b1: B \]
\[ c1: C \]
\[ ... \]
\[ c1.f \]
\[ c1.A_f \]
\[ a1.f \]
\[ b1.f \]

Invalid:
- \( a1.A_f \)
- \( b1.A_f \)
Are all name clashes bad?

A name clash must be removed unless it is:

- Under repeated inheritance (i.e. not a real clash)

- Between features of which at most one is effective (i.e. others are deferred)
Feature merging

\[ f^* \quad A \quad f^* \quad B \quad f^* \quad C \quad f^+ \]

- * Deferred
- + Effective
Feature merging: with different names

class D
inherit A
    rename g as f
end
B
C
    rename h as f
end

feature ...
end

* Deferred
+ Effective
\[Renaming\]
Feature merging: effective features

```
f+  A  f+  B  C  f+
f--  D  f--

* Deferred
+ Effective
-- Undefine
```
deferred class $T$
  inherit $S$
  undefine $\nu$ end

feature
  ...
end
Merging through undefinedness

class D
  inherit A
    undefine f end
  B
    undefine f end
  C
  feature ...
  end

* Deferred
+ Effective
-- Undefine
Merging effective features with different names

```
class D
  inherit A
  undefine f end
  B
  rename g as f
  undefine f end
  C
  rename h as f
  undefine f end
feature ...
end
```
Acceptable name clashes

If inherited features have all the same names, there is no harmful name clash if:

- They all have compatible signatures
- At most one of them is effective

Semantics of such a case:

- Merge all features into one
- If there is an effective feature, it imposes its implementation
Feature merging: effective features

- $g^+$
- $f^+$
- $h^+$
- $g^-$
- $f^-$
- $h^-$

$A$: $a1$, $a1.g$
$B$: $b1$, $b1.f$
$C$: $c1$, $c1.h$
$D$: $d1$, $d1.f$

Diagram:

- $A$ connected to $D$ by $g^+$ and $f^-$
- $B$ connected to $D$ by $f^+$ and $h^-$
- $C$ connected to $D$ by $h^+$ and $f^-$

$D$ is the root node, and $A$, $B$, and $C$ are its children.
A special case of multiple inheritance

This is a case of repeated inheritance
Indirect and direct repeated inheritance

Diagram showing inheritance relationships involving classes A, B, C, and D.
Multiple is also repeated inheritance

A typical case:

- Copy: `copy`
- Is equal: `is_equal`
- D

Diagram:
- ANY
- LIST
- C
- D
- C_copy
- C_is_equal

**copy**

**is_equal**

**copy**

**is_equal**

**copy**

**is_equal**
Sharing and replication

Features such as \( f \), not renamed along any of the inheritance paths, will be shared.
Features such as \( g \), inherited under different names, will be replicated.
The need for select

A potential ambiguity arises because of polymorphism and dynamic binding:

\[
\begin{align*}
  a1 & : \text{ANY} \\
  d1 & : \text{D} \\
  \ldots \\
  a1 & := d1 \\
  a.\text{copy} & (\ldots) \\
\end{align*}
\]
Removing the ambiguity

class D
inherit LIST [T]
end

C
rename copy as C_copy,
is_equal as C_is_equal,
...
end
When is a name clash acceptable?

(Between $n$ features of a class, all with the same name, immediate or inherited.)

- They must all have compatible signatures.
- If more than one is effective, they must all come from a common ancestor feature under repeated inheritance.
Another application of renaming

Provide locally better adapted terminology.

Example: \textit{child (TREE); subwindow (WINDOW)}
Multiple Inheritance and Class Tables

Why does this not work with multiple subtyping?

Can we still do $O(1)$ lookup?
The dynamic binding function

Name $\times$ StaticType $\times$ DynamicType $\rightarrow$ Code
The dynamic binding function

\[(\text{Name} \times \text{StaticType}) \to \text{DynamicType} \to \text{Code}\]
The dynamic binding function

\[(\text{Name } \times \text{StaticType}) \rightarrow \text{DynamicType} \rightarrow \text{Code}\]

Static

Dynamic
### Feature Tables

<table>
<thead>
<tr>
<th>Static Type</th>
<th>Dynamic Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>f of A</strong></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>Code Pointer</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>Code Pointer</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Code Pointer</td>
</tr>
<tr>
<td><strong>g of A</strong></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>Code Pointer</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>Code Pointer</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Code Pointer</td>
</tr>
<tr>
<td><strong>h of B</strong></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>g of C</strong></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Non-conforming inheritance

class ARRAY [G] feature ...
    lower, upper: INTEGER
    resize (l, u: INTEGER)
        ensure lower = l; upper = u
end

class ARRAYED_LIST [G]
    inherit
        LIST [G]
        ARRAY [G]
...

    invariant
        starts_from_1: lower = 1
end

a: ARRAY [INTEGER]; l: ARRAYED_LIST [INTEGER]
...

a := l
a.resize (-10, 10)

Class invariant violation
Inheritance basics: extended

If $B$ inherits from $A$:

- **As modules**: all the services of $A$ are available in $B$ (possibly with a different implementation)

- **As types**: whenever an instance of $A$ is required, an instance of $B$ will be acceptable ("is-a" relationship)

If $B$ inherits from $A$ in a non-conforming way:

- **As modules**: all the services of $A$ are available in $B$ (possibly with a different implementation)

- No relationship between types!
Non-conforming inheritance

class ARRAYED_LIST[G]
    inherit LIST[G]
    inherit {NONE}
    inherit ARRAY[G]

... 

invariant
    starts_from_1: lower = 1
end

a: ARRAY [INTEGER];
l: ARRAYED_LIST [INTEGER]

... 

a := l
a.resize (-10, 10)
No need for select

Potential ambiguity is resolved in favor of conforming parent:

\[ f : \text{FINITE} [...] \]
\[ al : \text{ARRAYED_LIST} [...] \]

\[ f := al \]

\[ \text{print}(f.\text{count}) \]
Covariance

class \textit{LIST}[G] \textbf{feature}
  cursor: \textit{CURSOR}
  \textit{go\_to} \,(c: \textit{CURSOR}) \textbf{is} \textbf{do} ... \textbf{end}
  ...
end

class \textit{LINKED\_LIST}[G] \textbf{inherit}
  \textit{LIST}[G]
  \textbf{redefine} cursor, go\_to, ...
feature
  cursor: \textit{LINKED\_CURSOR}
  \textit{go\_to} \,(c: \textit{LINKED\_CURSOR}) \textbf{is} \textbf{do} ... \textbf{end}
  ...
end
Anchored types

class LIST[G] feature
  cursor: CURSOR
  go_to (c: like cursor) is do ... end
...
end

class LINKED_LIST[G] inherit LIST[G]
  redefine cursor, ... end
 feature
  cursor: LINKED_CURSOR
  -- No need to redefine `go_to'
  ...
end
Semantics of anchored types

In class $C$:

\[
\begin{align*}
&x: \text{SOME\_TYPE} \\
y: \text{like } x
\end{align*}
\]

In class $C$, $y$ is treated exactly as with $y: \text{SOME\_TYPE}$

In a descendant $D$ of $C$, if $x$ has been redeclared to some other type, $y$ will be treated as it if also had that type.
Type redefinition rule

Eiffel:
- **covariant** redefinition of result (may change type to a descendant of the original type)
- **covariant** redefinition of arguments

Traditional notion of subtyping:
- **covariant** redefinition of result
- **contravariant** redefinition of arguments (may change type to an ancestor of the original type)

Contravariant redefinition: safe but useless
The problem with covariance

list: LIST [...]  
linked_list: LINKED_LIST [...]  

c: CURSOR

...  
list := linked_list 
list.go_to (c)

Catcall!
CAT calls

- **CAT** stands for **Changing Availability or Type**
- A routine is a CAT if some redefinition changes its export status or the type of any of its arguments
- A call is a **catcall** if some redefinition of the routine would make it invalid because of a change of export status or argument type
Catcall cases

- Covariant redefinition of arguments
  - Non-generic case
  - Generic case
- Descendant hiding (in earlier versions of Eiffel)
Covariant redefinition: non-generic case

class ANIMAL
  feature
    eat (a_food: FOOD)
      deferred
      end
  end
end

class WOLF
  inherit ANIMAL
  redefine eat end
  feature
    eat (a_meat: MEAT)
      do ...
      end
  end
end
animal: ANIMAL
wolf: WOLF
food: FOOD
grass: GRASS

create wolf
create grass
animal := wolf
food := grass
animal.eat (grass)
Covariant redefinition: generic case

animal_list: LINKED_LIST[ANIMAL]
sheep_list: LINKED_LIST[SHEEP]
sheep: SHEEP
wolf: WOLF

sheep_list.extend(sheep)
animal_list := sheep_list
animal_list.extend(wolf)
Covariant redefinition: generic case

```plaintext
class LINKED_LIST[ANY]
  feature
    extend (v: ANY) do ... end
  end

class LINKED_LIST[SHEEP]
  feature
    extend (v: SHEEP) do ... end
  end

class LINKED_LIST[WOLF]
  feature
    extend (v: WOLF) do ... end
  end
```
class RECTANGLE
  inherit POLYGON
  export \{NONE\}
    add_vertex
  end

...  

invariant
  vertex_count = 4  
end

\textit{r: RECTANGLE; p: POLYGON}

... 

\texttt{p := r}\n\texttt{p.add_vertex(...)}
Descendant hiding: solution 1

class RECTANGLE
inherit {NONE} POLYGON
  export {NONE}
    add_vertex
  end
...

invariant
  vertex_count = 4
end

r: RECTANGLE; p: POLYGON
...

p := r
p.add_vertex(...)
class POLYGON feature ...  
  variable_vertex_count: BOOLEAN  
  do Result := True end  
  add_vertex (...)  
  require variable_vertex_count  
  do ... end  
end

class RECTANGLE inherit POLYGON  
  redefine variable_vertex_count end
feature ...  
  variable_vertex_count: BOOLEAN = False  
end

p: POLYGON  
...  
if p.variable_vertex_count then  
  p.add_vertex (...)  
end
Inheritance: summary (1)

- **Type mechanism:** lets you organize our data abstractions into taxonomies
- **Module mechanism:** lets you build new classes as extensions of existing ones
- **Polymorphism:** flexibility *with* type safety
- **Dynamic binding:** automatic adaptation of operation to target, for more modular software architectures
- **Contracts inheritance:** ensures properties to clients in presence of polymorphism
- **Deferred classes:** a mechanism to express properties of abstract concepts
Inheritance: summary (2)

- Polymorphic containers + object test: flexible and type-safe
- Constrained genericity: allows calling specific features on expression of parameter types
- Multiple inheritance: powerful way of combining abstractions
- Non-conforming inheritance: a way to express code reuse without subtyping
- Catcalls: result from covariant redefinition of argument types (even more tricky with genericity!)