Genericity: summary 1

- Type extension mechanism
- Reconciles flexibility with type safety
- Enables us to have parameterized classes
- Useful for container data structures: lists, arrays, trees, ...
- “Type” now a bit more general than “class”
(Reminder) A generic class

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

To use the class: obtain a generic derivation, e.g.

```plaintext
cities: LIST[CITY]
```

(Reminder) Using generic derivations

```plaintext
cities: LIST[CITY]
people: LIST[PERSON]
c: CITY
p: PERSON
...
cities.extend(c)
people.extend(p)
c := cities.last
c.add_tram_line(Line8)
```

(Reminder) Using genericity

```plaintext
LIST [CITY]
LIST [LIST [CITY]]
...
A type is no longer exactly the same thing as a class!

(But every type remains based on a class.)
```
Genericity: summary 1

- Type extension mechanism
- Reconciles flexibility with type safety
- Enables us to have parameterized classes
- Useful for container data structures: lists, arrays, trees, ...
- "Type" now a bit more general than "class"

Definition: Type

We use types to declare entities, as in:

```
x : SOME_TYPE
```

With the mechanisms defined so far, a type is one of:

- A non-generic class
  - e.g. METRO_STATION
- A generic derivation, i.e. the name of a class followed by a list of actual generic parameters, in brackets
  - e.g. LIST[METRO_STATION]
  - LIST[ARRAY[METRO_STATION]]

Genericity + inheritance 1: Constrained genericity

```ruby
class VECTOR[G] feature
  plus alias "+" (other: VECTOR[G]): VECTOR[G] is
    - Sum of current vector and other
    require
      lower = other.lower
      upper = other.upper
  local
    a, b, c : G
  do
    ... See next ...
  end
  ... Other features, in particular item to access an element ...
end
```
Adding two vectors

\[ \begin{array}{ccc}
  & u & + \\
 2 & v & = \\
 1 & w \\
\end{array} \]

Constrained genericity

Body of `plus alias *+*`:

```pascal
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
```

The solution

Declare class `VECTOR` as

```pascal
class VECTOR[\text{G} \rightarrow \text{NUMERIC}] feature
  ... The rest as before ...
end
```

Class `NUMERIC` (from the Kernel Library) provides features `plus alias *+*`, `minus alias -=` and so on.
Improving the solution

Make VECTOR itself a descendant of NUMERIC, effecting the corresponding features:

```plaintext
class VECTOR [G -> NUMERIC] inherit NUMERIC
feature ...
end
```

Then it is possible to define

```plaintext
v: VECTOR [INTEGER];
w: VECTOR [VECTOR [INTEGER]];
vv: VECTOR [VECTOR [VECTOR [INTEGER]]];
```

Extending the basic notion of class

Genericity + Inheritance 2: Polymorphic data structure

```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);
fl.last.display
```
Example hierarchy

Forcing a type: the problem

```
fl.store("FILE_NAME")
...
-- Two years later:
  fl := retrieved("FILE_NAME") - See next
  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.

The solution: Assignment attempt

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

But:

If `x` is declared of type `RECTANGLE`, [1] is invalid.
If `x` is declared of type `FIGURE`, [2] is invalid.
Assignment attempt

```plaintext
f: FIGURE
r: RECTANGLE
...
f.retrieve("FILE_NAME")
f := fl.last
r ?= f
if r /= Void then
  print(r.diagonal)
else
  print("Too bad.")
end
```

Assignment attempt

```plaintext
x ?= y
with
  x : A
If y is attached to an object whose type conforms to A, perform normal reference assignment. Otherwise, make x void.
```

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:

```forth
forth is deferred ensure index = old index + 1 end
```

Mixing deferred and effective features

In the same class

```forth
define (x: G) is
  -- Move to first position after current
  -- where x appears, or after if none.
  do
    from until after or else item = x loop
  forth end
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
Applications of deferred classes

Analysis and design, top-down
Taxonomy
Capturing common behaviors

Deferred classes in EiffelBase

Java and .NET solution

Single inheritance only for classes
Multiple inheritance from interfaces

An interface is like a fully deferred class, with no implementations (do clauses), no attributes (and also no contracts)
Inheritance and assertions

Correct call:
if \( a1.\alpha \) then
  \( a1.r(...) \)
else
  ...
end

Assertion redeclaration rule

Redefined version may not have require or ensure. May have nothing (assertions kept by default), or

\[
\text{require else new_pre} \\
\text{ensure then new_post}
\]

Resulting assertions are:
- original_precondition or new_pre
- original_postcondition and new_post

Invariant accumulation

Every class inherits all the invariant clauses of its parents. These clauses are conceptually "and"-ed.
What we have seen

- Genericity
- Combining inheritance with genericity
- Constrained genericity
- Forcing a type: the assignment attempt
- The relationship of contracts with inheritance
- Deferred classes and features

End of lecture 14