1. Agenda for today

- Constrained genericity
- Creating with a specified type
- Once routines
- Multiple inheritance

2. Adding two vectors

\[
\begin{align*}
\mathbf{u} + \mathbf{v} &= \mathbf{w} \\
\text{let } i &= \begin{bmatrix} a & b & c \end{bmatrix}
\end{align*}
\]

3. Constrained genericity

```java
class VECTOR [G]
feature
infix "+" (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 ...
end
```
Constrained generality

- The body of infix "+":

```plaintext
create Result.make (lower, upper)
from
i := lower
until
i > upper
loop
a := item (i)
b := other.item (i)
c := a + b  -- Requires a "+" operation on G!
Result.put (c, i)
i := i + 1
end
```

Adding two vectors

```
u + v = w
```

Constrained generality: The solution

- Declare class VECTOR as

```plaintext
class VECTOR [G -> NUMERIC]
feature
    ... The rest as before ...
end
```

- Class NUMERIC (from the Kernel Library) provides features infix "+", infix "*" 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
    ... The rest as before, including infix "+"...
end
```

- Then it is possible to define e.g.

```plaintext
v: VECTOR [VECTOR [VECTOR [INTEGER]]]
```

Agenda for today

- Constrained generality
- Creating with a specified type
- Once routines
- Multiple inheritance

Creating with a specified type

- To avoid this:

```plaintext
a1: A
b1: B
... create b1.make (...) a1 := b1
```

- Simply use

```plaintext
a1: A
... create {B} a1.make (...)  
```

(See factory pattern)
Agenda for today

- Constrained genericity
- Creating with a specified type
- Once routines
- Multiple inheritance

Once routines

- If instead of
  \[ r \text{ is} \begin{array}{ll} \text{do} & \cdots \text{Instructions} \cdots \\ \text{end} \end{array} \]
- you write
  \[ r \text{ is} \begin{array}{ll} \text{once} & \cdots \text{Instructions} \cdots \\ \text{end} \end{array} \]
- then \textit{Instructions} will be executed only for the first call by any client during execution. Subsequent calls return immediately.
- In the case of a function, subsequent calls return the result computed by the first call.

Scheme for shared objects

```plaintext
class SHARED_OBJECTS
  feature
  error_window: WINDOW is
    once create Result.make(...)
  end

  exit_button: BUTTON is
    once create Result.make(...)
  end
end

class MY_APPLICATION_CLASS inherit SHARED_OBJECTS
feature
  r is do
    error_window.put [my_error_message]
  end
end
```

Multiple inheritance

- Allow a class to have two or more parents.
- Examples that come to mind: \textit{ASSISTANT} inherits from \textit{TEACHER} and \textit{STUDENT}.

Example: Teaching assistant

- This is in fact a case of \textit{repeated} inheritance:
Other examples of multiple inheritance

- Combining separate abstractions:
  - Restaurant, train car
  - Calculator, watch
  - Plane, asset

Multiple inheritance: Combining abstractions

Multiple inheritance: Composite figures

Defining the notion of composite figure

Composite figures through multiple inheritance

A composite figure as a list
**Composite figures**

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

**Complex figures**

- A simpler form of procedures `display`, `move` etc. can be obtained through the use of iterators.
- We'll learn to use agents for that purpose.

**Multiple inheritance: Name clashes**

```plaintext
class C
  inherit A
  rename foo as fog
end
B
rename foo as zoo
end
feature
  ...
```

**Resolving name clashes**

```plaintext
class C
  inherit A
  rename foo as fog
end
B
rename foo as zoo
end
feature
  ...
```

**Results of renaming**

```plaintext
a1: A
b1: B
c1: C
...
c1.fog
c1.zoo
a1.foo
b1.foo

Invalid:
a1.fog, a1.zoo, b1.zoo, b1.fog, c1.foo
```
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: `child (TREE); subwindow (WINDOW)`.

End of lecture 10