# Einführung in die Programmierung
# Introduction to Programming

Prof. Dr. Bertrand Meyer

## Exercise Session 7

# News (Reminder)

Mock exam next week!

- Monday exercise groups: November 7
- Tuesday exercise groups: November 8
- You have to be present
- The week after we will discuss the results
- Assignment 7 due on November 15

# Today

- Inheritance
- Genericity
- Exercise: practicing contracts

# Inheritance

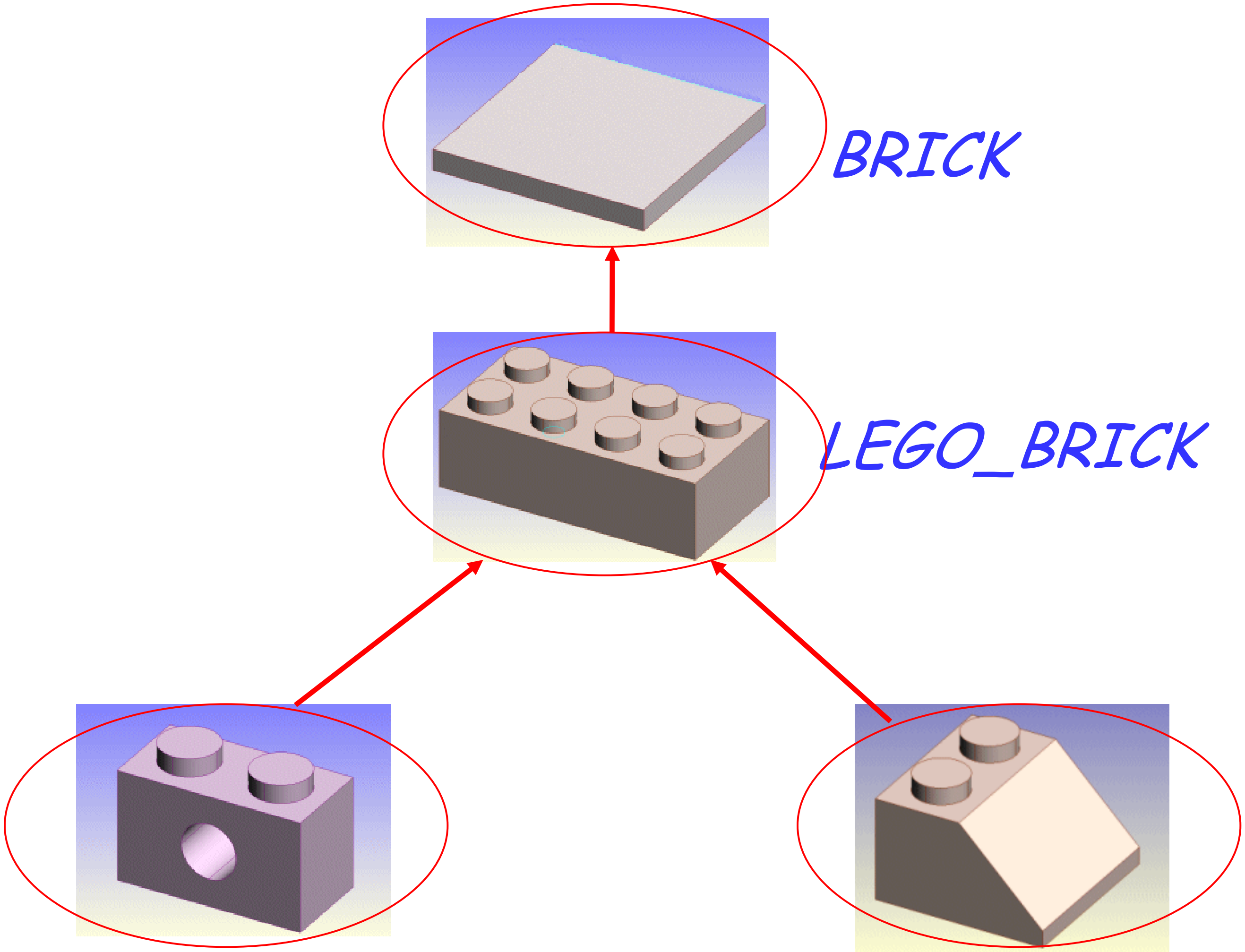
Principle:

Describe a new class as extension or specialization of an existing class
(or several with ***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)

# Let's play Lego!



BRICK

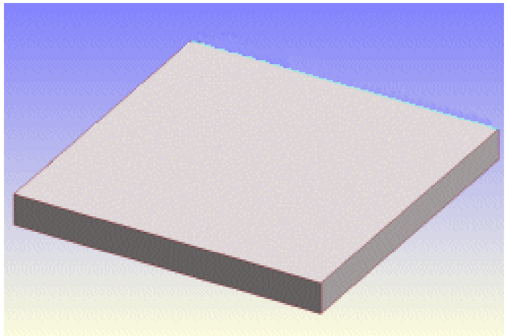
LEGO_BRICK

LEGO_BRICK_WITH_HOLE

LEGO_BRICK_SLANTED

# Class *BRICK*

```
deferred class
    BRICK

feature
    width: INTEGER
    depth: INTEGER
    height: INTEGER
    color: COLOR

    volume: INTEGER
        deferred
        end
end
```



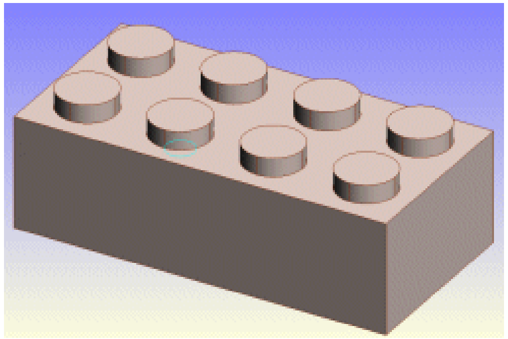
# Class *LEGO_BRICK*

```
class
    LEGO_BRICK

inherit
    BRICK

feature
    number_of_nubs: INTEGER
        do
            Result := ...
        end

    volume: INTEGER
        do
            Result := ...
        end
end
```

Inherit all features of class *BRICK.*

New feature, calculate all nubs

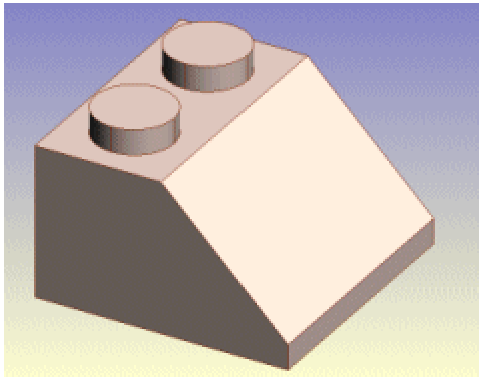
Implementation of *volume.*



# Class *LEGO_BRICK_SLANTED*

```
class
    LEGO_BRICK_SLANTED

inherit
    LEGO_BRICK
        redefine
            volume
        end

feature
    volume: INTEGER
        do
            Result := ...
        end
end
```

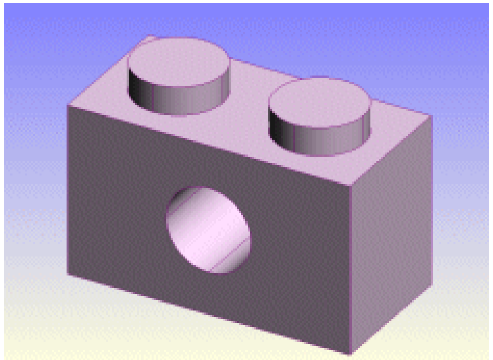
The feature *volume* is going to be redefined (=changed). The feature *volume* comes from *LEGO_BRICK*



# Class *LEGO_BRICK_WITH_HOLE*

```
class
    LEGO_BRICK_WITH_HOLE

inherit
    LEGO_BRICK
        redefine
            volume
        end

feature
    volume: INTEGER
        do
            Result := ...
        end
end
```

The feature *volume* is going to be redefined (=changed). The feature *volume* comes from *LEGO_BRICK*



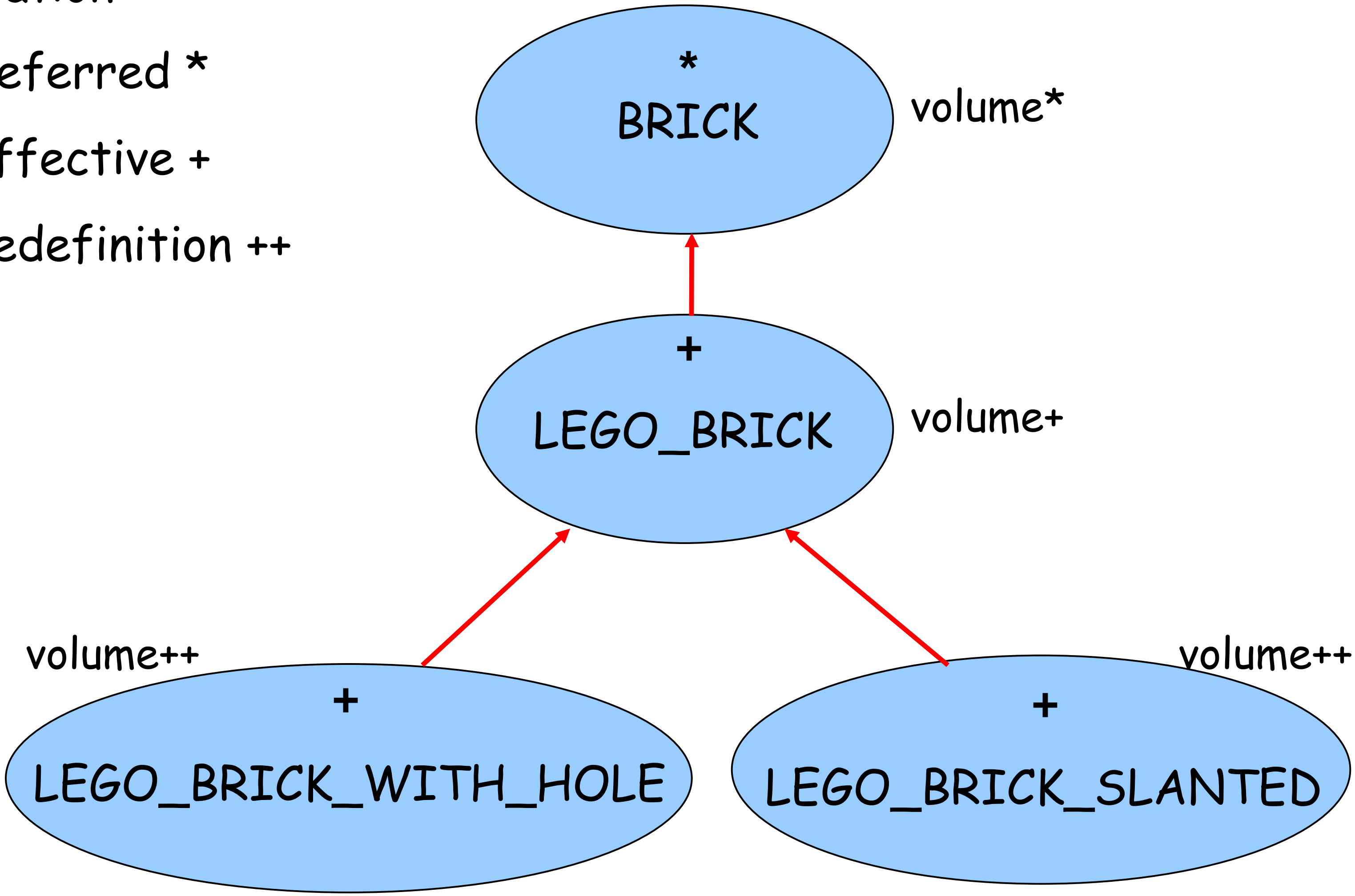
# Inheritance Notation

Notation:

Deferred *

Effective +

Redefinition ++



# Deferred

- Deferred
  - Deferred classes can have deferred features.
  - A class with at least one deferred feature must be declared as deferred.
  - A deferred feature does not have an implementation yet.
  - Deferred classes cannot be instantiated and hence cannot contain a create clause.

# Effective

- Effective
  - Effective classes do not have deferred features (the "standard case").
  - Effective routines have an implementation of their feature body.

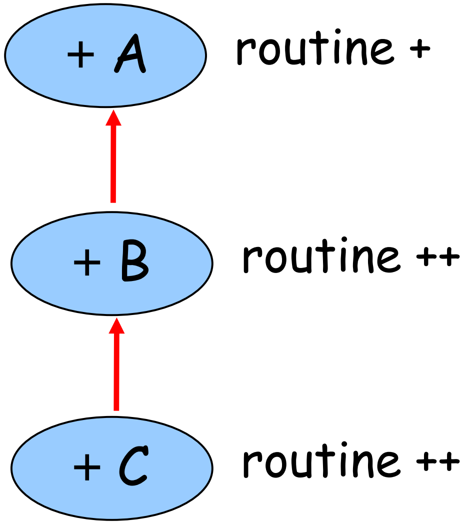
# Precursor

- If a feature was redefined, but you still wish to call the old one, use the **Precursor** keyword.

```
volume: INTEGER
  do
    Result := Precursor - ...
  end
```

# A more general example of using Precursor

```
    -- Class A
routine (a_arg1 : TYPE_A): TYPE_R
    do … end

    -- Class C
routine (a_arg1 : TYPE_A): TYPE_R
    local
        l_loc : TYPE_R
    do
            -- pre-process
        l_loc := Precursor {B}  (a_arg1 )
            -- Not allowed: l_loc := Precursor {A} (a_arg1 )
            -- post-process
    end
```



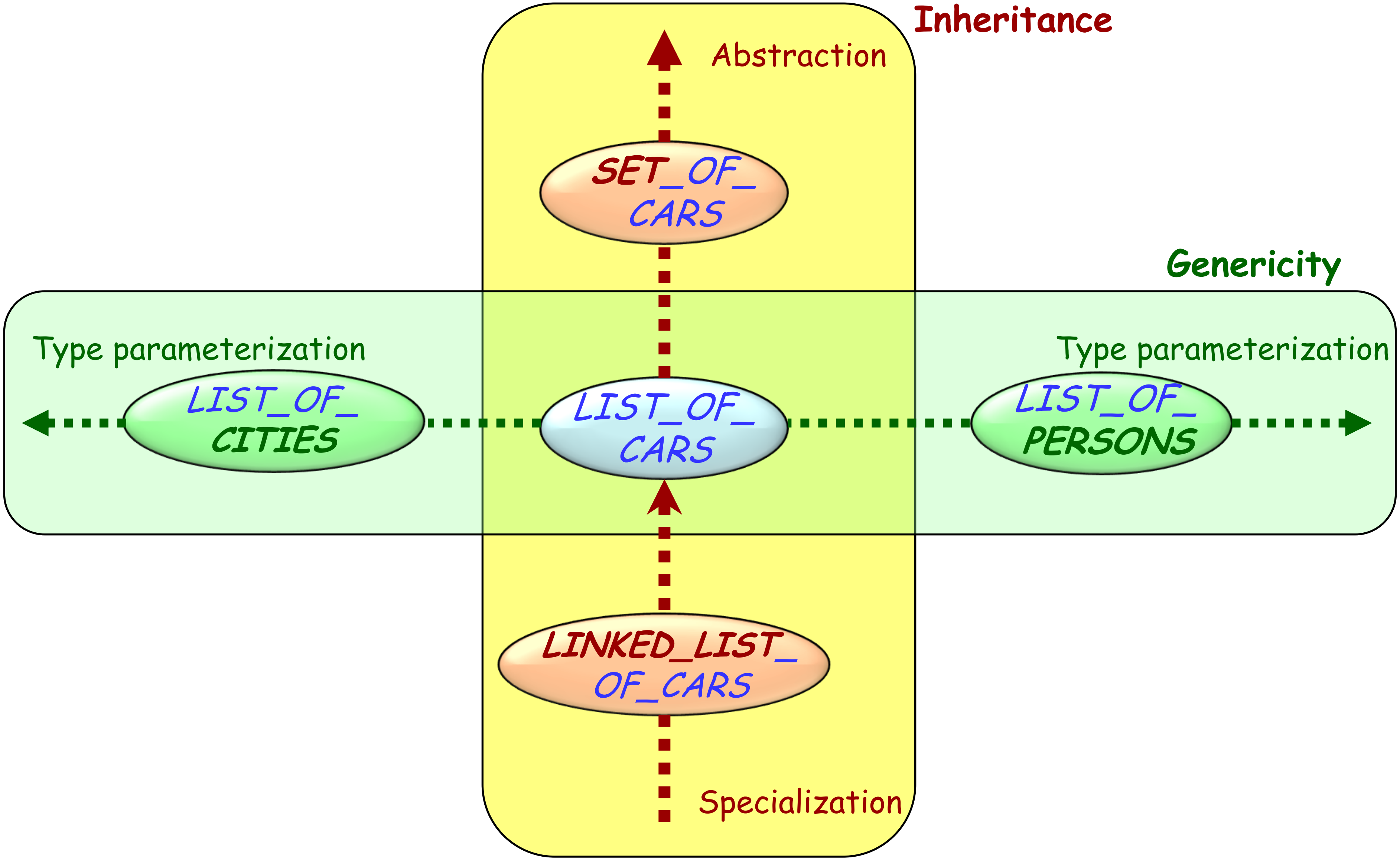
# Today

- Inheritance
- Genericity
- Exercise: practicing contracts

# Genericity

- Genericity lets you parameterize a class. The parameters are types. A single class text may be reused for many different types.

# Genericity



Inheritance
Abstraction
SET_OF_ CARS
Genericity
Type parameterization
Type parameterization
LIST_OF_ CITIES
LIST_OF_ CARS
LIST_OF_ PERSONS
LINKED_LIST_ OF_CARS
Specialization

# A generic list

Formal generic parameter

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

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

Actual generic parameter

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

# A generic list with constraints

```
class
    STORAGE [G]-> RESOURCE

inherit
    LIST [G]

feature
    consume_all
        do
            from start until after
            loop
                item.consume
                forth
            end
        end
end
```

constrained generic parameter

The feature *item* is of type G. We cannot assume *consume*.

is
n
an
assume this.

# Type-safe containers

- Using genericity you can provide an implementation of type safe containers.

```
x: ANIMAL
animal_list: LINKED_LIST [ANIMAL]
a_rock: MINERAL

animal_list.put (a_rock) -- Does this rock?
```

# 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 ***types***, the actual generic parameters, in brackets (also recursive)

  e.g. *LIST* [*ARRAY* [*METRO_STATION* ]]

  *LIST* [*LIST* [*CITY* ]]

  *TABLE* [*STRING*, *INTEGER*]

# So, how many types can I possibly get?

Two answers, depending on what we are talking about:

- Static types

Static types are the types that we use while writing Eiffel code to declare types for entities (arguments, locals, return values)

- Dynamic types

Dynamic types on the other hand are created at run-time. Whenever a new object is created, it gets assigned to be of some type.

# Static types

```
class EMPLOYEE
feature
        name: STRING
        birthday: DATE
end

class DEPARTMENT
feature
        staff: LIST[EMPLOYEE]
end
```

bound by the program text:

*EMPLOYEE*

*STRING*

*DATE*

*DEPARTMENT*

*LIST[G]*

*becomes LIST[EMPLOYEE]*

# Object creation, static and dynamic types

```
class TEST_DYNAMIC _CREATION
feature
    ref_a: A; ref_b: B
        -- Suppose B, with creation feature make_b,
        -- inherits from A, with creation feature make_a

    do_something
        do
            create ref_a.make_a
                -- Static and dynamic type is A
            create {B} ref_a.make_b
                -- Static type is A, dynamic type is B
        end
end
```

# Dynamic types: another example

```
class SET[G] feature
  powerset: SET[SET[G]] is
    do
          create Result
          -- More computation…
    end

  i_th_power (i: INTEGER): SET[ANY]
    require i >= 0
    local n: INTEGER
    do
          Result := Current
          from n := 1 until n > i loop
                Result := Result.powerset
                n := n + 1
          end
    end
end
```

Dynamic types from *i_th_power* :

*SET[ANY]*

*SET[SET[ANY]]*

*SET[SET[SET[ANY]]]*

…

From http://www.eiffelroom.com/article/fun_with_generics

# Today

- Inheritance
- Genericity
- Exercise: practicing contracts

# Specification of a card game

A deck is initially made of 36 cards

Every card in the deck represents a value in the range 2..10

Every card also represents 1 out of 4 possible colors

The colors represented in the game cards are:
red ('R'), white ('W'), green ('G') and blue ('B')

As long as there are cards in the deck, the players can look at the top card and remove it from the deck

# Class CARD create make

Hands-On

```
make (a_color: CHARACTER, a_value: INTEGER)
                -- Create a card given a color and a value.
        require
                ...

        ensure
                ...

color: CHARACTER
                -- The card color.
value: INTEGER
                -- The card value.
```

# Class CARD: which colors are valid?

Hands-On

```
is_valid_color (a_color: CHARACTER): BOOLEAN
                -- Is `a_color' a valid color?
        require
                ...

        ensure
                ...
```

# Class CARD: which ranges are valid?

Hands-On

```
is_valid_range (n: INTEGER): BOOLEAN
                -- Is `n' in the acceptable range?
        require
                ...

        ensure
                ...

invariant
                ...
```

# Class CARD create make: reloaded

Hands-On

```
make (a_color: CHARACTER, a_value: INTEGER)
                -- Create a card given a color and a value.
        require
                ...

        ensure
                ...

color: CHARACTER
                -- The card color.
value: INTEGER
                -- The card value.
```

# Class DECK create make

Hands-On

```
make
            -- Create a deck with random cards.
    require
            ...
    ensure
            ...

feature {NONE} -- Implementation

card_list: LINKED_LIST [CARD]
            -- Deck as a linked list of cards.
```

# Class DECK queries

Hands-On

```
top_card: CARD
                -- The deck's top card.


is_empty: BOOLEAN
                -- Is Current deck empty?
        do
                ...
        end
count: INTEGER
                -- Number of remaining cards in the deck.
        do
                ...
        end
```

# Removing the top card from DECK

Hands-On

```
remove_top_card
            -- Remove the top card from the deck.
    require
            ...

    ensure
            ...
```

# The class invariant

Hands-On

**invariant**

...