Part 1: Language constructs

1.3 EXCEPTION HANDLING

Java: Exception Handling

```
public class Printer {
    public print(int i) {
        try {
            throw new Exception()
        }
        catch(Exception e) {   }
        }
}
```

Eiffel: Exception Handling

```
class
    PRINTER
feature
    print_int (a_int: INTEGER)
        local
            l_retried: BOOLEAN
        do
            if not l_retried then
                (create {DEVELOPER_EXCEPTI
            else
                -- Do something
            end
        rescue
            l_retried := True
            retry
    end
end
```

- EXCEPTION
  - ASSERTION_VIOLATION
    - CHECK_VIOLATION
    - INVARIANT_VIOLATION
    - LOOP_INVARIANT_VIOLATION
    - POSTCONDITION_VIOLATION
    - PRECONDITION_VIOLATION
    - VARIANT_VIOLATION
  - DEVELOPER_EXCEPTION
  - MACHINE_EXCEPTION
    - HARDWARE_EXCEPTION
      - FLOATING_POINT_FAILURE
    - OPERATING_SYSTEM_EXCEPTION
      - COM_FAILURE
      - OPERATING_SYSTEM_FAILURE
      - OPERATING_SYSTEM_SIGNAL_FAILURE
  - OBSOLETE_EXCEPTION
    - EXCEPTION_IN_SIGNAL_HANDLER_FAILURE
    - RESCUE_FAILURE
    - RESUMPTION_FAILURE
  - SYS_EXCEPTION
    - EIFFEL_RUNTIME_PANIC
    - EIF_EXCEPTION
      - EIFFEL_RUNTIME_EXCEPTION
        - DATA_EXCEPTION
          - IO_FAILURE
          - MISMATCH_FAILURE
          - SERIALIZATION_FAILURE
        - EXTERNAL_FAILURE
        - NO_MORE_MEMORY
      - LANGUAGE_EXCEPTION
        - BAD_INSPECT_VALUE
        - EIFFELSTUDIO_SPECIFIC_LANGUAGE_EXCEPTION
          - ADDRESS_APPLIED_TO_MELTED_FEATURE
          - CREATE_ON_DEFERRED
        - ROUTINE_FAILURE
        - VOID_ASSIGNED_TO_EXPANDED
        - VOID_TARGET
    - OLD_VIOLATION

Eiffel: retry example

```
feature
    transmit (a_p: PACKET)
            -- transmit packet a_p
        local
            l_current_retries: INTEGER
            r: RANDOM_NUMBER_GENERATOR
        do
            line.send (a_p)
        rescue
            if l_current_retries < max_retries then
                r.next
                wait_millisecs (r.value_between(20, 50))
                current_retries := current_retries + 1
                retry
            end
    end
end
```

Part 1: Language constructs

1.4 ONCE ROUTINES

What are once routines?

```
foo: INTEGER
    once
        Result := factorial (10)
    end
test_foo
    do
        io.put_integer (foo)  -- 3628800, calculated
        io.put_integer (foo)  -- 3628800, directly returned
    end
```

- Executed the first time
- Result is stored
- In further calls, stored result is returned
- In other languages
    - Static variables
    - Singleton pattern

Use of once routines

- Constants, other than basic types

  i: COMPLEX

  once create *Result.make (0, 1)* **end**

- Lazy initialization

  settings: SETTINGS

  once create *Result.load_from_filesystem* **end**

- Initialization procedures

  init_graphics_system

  once *...* **end**

Part 1: Language constructs

1.5 STYLE RULES

Style rule

For indentation, use tabs, not spaces

Tabs

```
class
	PREVIEW

inherit
	TOURISM

feature
	explore
			-- Show city info
			-- and route.
		do
			Paris.display
			Louvre.spotlight
			Line8.highlight
			Route1.animate
		end
end
```

More style rules

- Class name: all upper-case

Full words, no abbreviations (with some exceptions)

- Classes have global namespace: two classes cannot have the same name (even in different clusters)

- Usually, classes are prefixed with a library prefix

    EiffelVision2: EV_

    Base is not prefixed

```
class
    PREVIEW

inherit
    TOURISM

feature
    explore
            -- Show city info
            -- and route.
        do
            Paris.display

            Louvre.spotlight
            Line8.highlight
            Route1.animate
        end
end
```

Even more style rules

- For feature names, use full words, not abbreviations
- Always choose identifiers that clearly identify the intended role
- Use words from natural language (preferably English) for the names you define
- For multi-word identifiers, use underscores

```
class
    PREVIEW

inherit
    TOURISM

feature
    explore
            -- Show city info
            -- and route.
        do
            Paris.display
            Louvre.spotlight
            Line8.highlight
            Line8.remove_all_sections
            Route1.animate
        end
end
```

Eiffel Naming: Locals / Arguments

- Locals and arguments share namespace with features
  - Name clashes arise when a feature is introduced, which has the same name as a local (even in parent)

- To prevent name clashes:
  - Locals are prefixed with **l_**
  - Some exceptions like “i“ exist
  - Arguments are prefixed with **a_**

Part 1: Language constructs

1.6 GENERICS

Declaring generics

```
class
    MY_QUEUE [G]

feature

    item: G
        -- First item in queue.
      do … end

    extend (a_element: G)
        -- Add new element.
      do … end

end
```

G is called the generic parameter. By convention, the generic parameter name is G. If there are more parameters, use G, H, etc. or a meaningful abbreviation such as K for keys in a hash table

Creating instances of generics classes

```
class
    EXAMPLE1

feature
    int_queue
            -- An integer queue.
        local
          qi: MY_QUEUE [INTEGER]
        do
          create qi
            qi.extend (35)
            qi.extend (6)
        end
end
```

```
class
    EXAMPLE2

feature
    string_queue
            -- A string queue.
        local
          qs: MY_QUEUE [STRING]
        do
          create qs
            qs.extend (“Asterix”)
            qs.extend (“Obelix”)
            qs.extend (“Suffix”)
        end
end
```

Constraint generics

```
class
    MY_LIST [G -> COMPARABLE]

feature

    item: G
        -- First item in queue.
      do … end

    extend (a_element: G)
        -- Add new element.
      do
            … if a_element < item then
            …
    end

end
```

The generic parameter G must be a class inheriting from COMPARABLE

Creating instances of constraint generics classes

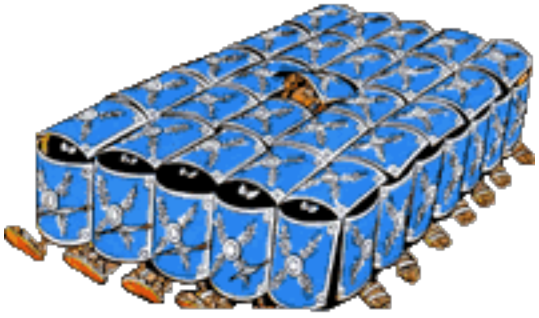
```
-- Valid declarations
  li: MY_LIST [INTEGER]
  ls: MY_LIST [STRING]
  lr: MY_LIST [REAL]
  ld: MY_LIST [DOUBLE]
  …
```

```
-- Invalid declarations
  la: MY_LIST [ACCOUNT]
  lb: MY_LIST [BANK]
  lm: MY_LIST [MAIN]
  …
```

Classes ACCOUNT, BANK & MAIN don't inherit from COMPARABLE

Part 1: Language constructs

1.8 INFORMATION HIDING



Two kinds of routine

Procedure: doesn't return a result

- Yields a **command**
- Calls are **instructions**

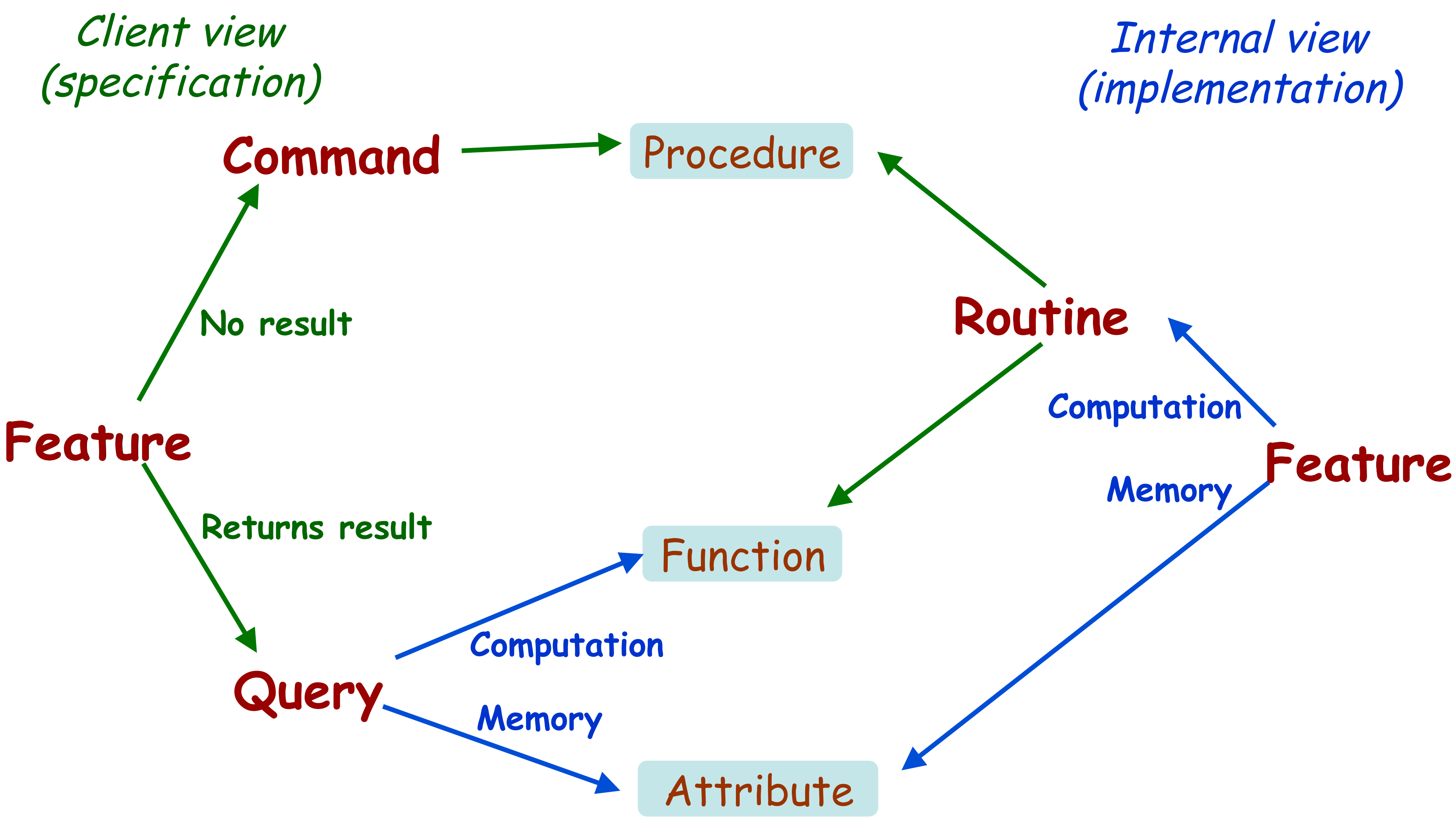
Function: returns a result

f (*arg* : *TYPE*; ...): $\boxed{\textit{RESULT\_TYPE}}$

... (The rest as before) ...

- Yields a **query**
- Calls are **expressions**

Features: the full story

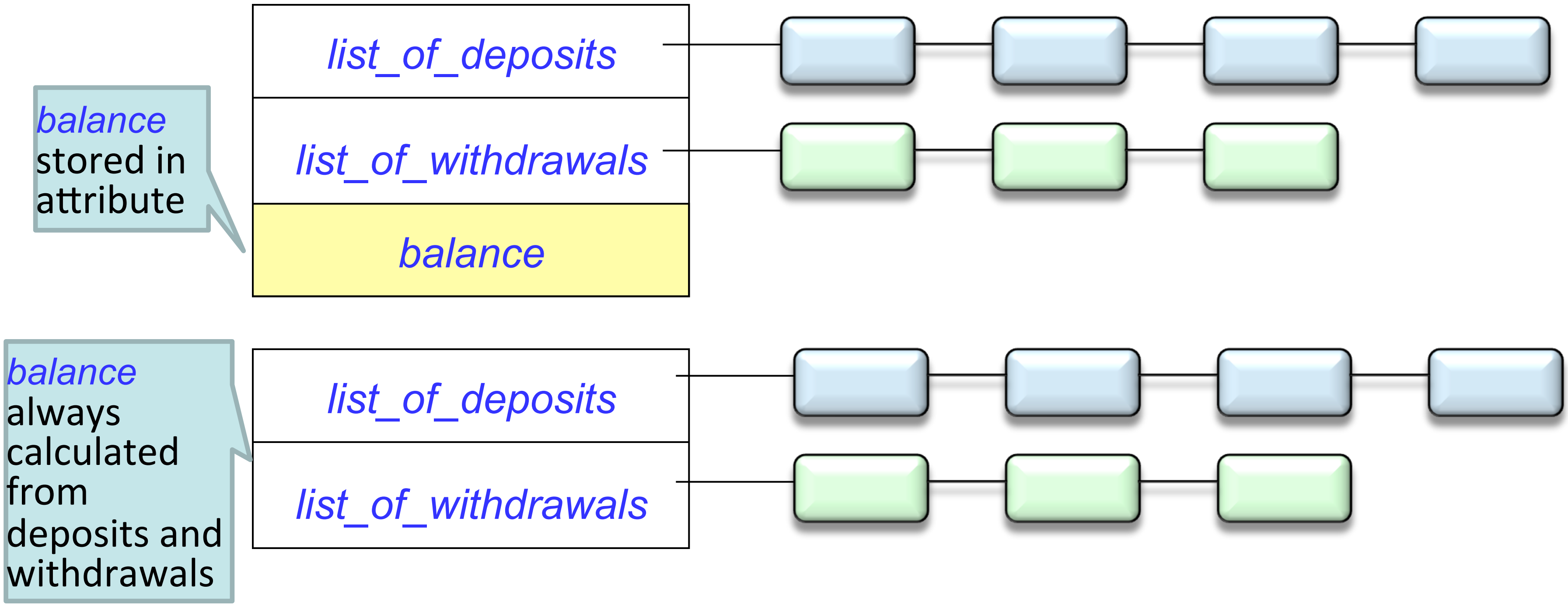


The Uniform Access principle

It doesn‘t matter to the client whether you look up or compute

Uniform Access: an example

balance = list_of_deposits.total – list_of_withdrawals.total



A call such as

your_account.balance

could use an attribute or a function

Exporting (making public) an attribute

In Eiffel, exporting an attribute means exporting it **read-only**

From the outside, it is not shown as an attribute, just as a **query**: it could be a function

In C++, Java & C#, if you make public an attribute* *x*, it is available for both read and write:

- *v* := *a1.x*
- *a1.x* := *v*

As a result, it is almost always a bad idea to export an attribute.

* (field, member variable)

Getter functions

In C++, Java & C#, the standard technique, if *private_x* is secret, is to export an associated **getter function**:

x : *T*
    do
        Result := *private_x*
    end

Eiffel needs no getter functions: just export the attribute

This is safe: the attribute is exported

- Only for reading
- Without the information that it is an attribute: it could be a function (Uniform Access principle)

Information hiding

```
class
   A

feature
          f ...
          g ...

feature {NONE}

          h, i ...

feature {B, C}

          j, k, l ...

feature {A, B, C}

          m, n ...
end
```

Status of calls in a client with *a1*: *A*:

- *a1.f*, *a1.g*: valid in any client
- *a1.h*: invalid everywhere

  (including in *A*'s own text!)
- *a1.j*: valid only in *B*, *C* and their descendants

  (not valid in *A*!)
- *a1.m*: valid in *B*, *C* and their descendants, as well as in *A* and its descendants

Information hiding

Information hiding only applies to use by clients, i.e. using dot notation or infix notation, as with *a1*.*f* (***Qualified*** calls).

Unqualified calls (within class) not subject to information hiding:

```
class A feature {NONE}
    h do ... end
feature

    f
        do
            ...; h ; ...
        end
end
```

PART 2: CONTRACTS

Contracts

A contract is a semantic condition characterizing usage properties of a class or a feature

Three principal kinds:

- Precondition
- Postcondition
- Class invariant

Design by Contract

Together with the implementation ("*how*") of each software element, describe "*what*" it is supposed to do: its contract

Three basic questions about every software element:

- What does it assume?
- What does it guarantee?
- What does it maintain?

Precondition

Postcondition

Invariant

Contracts in programming languages

Eiffel: integrated in the language

Java: Java Modeling Language (JML), iContract etc.

.Net languages: Code Contracts (a library)

Spec# (Microsoft Research extension of C#): integrated in the language

UML: Object Constraint Language

etc.

Precondition

Property that a feature imposes on every client:

```
factorial (i: INTEGER): INTEGER
    require
        valid_arg: i >= 0
    do
      …
    end
```

A feature with no **require** clause is always applicable, as if it had

```
    require
        always_OK: True
```

A client calling a feature must make sure that the precondition holds before the call

A client that calls a feature without satisfying its precondition is faulty (buggy) software.

Precondition

Another example:

```
extend (a_element: G)
    require
        valid_elem: a_element /= void
        not_full: not is_full
    do … end
```

A feature with a **require** clause

require

label_1: cond_1

label_2: cond_2 …

label_n: cond_n

is equivalent to

require

label: cond_1 **and** cond_2 **and** … *cond_n*

Assertions

Assertion tag

Condition

not_too_small: *i* >= 0

Assertion

Let's code…

Go to:

https://codeboard.io/projects/86

Task: in class CUSTOMER, write a precondition for the creation routine make_with_name_and_age

Task: create an invalid CUSTOMER object and try to run your program. What happens?

Task: fix your CUSTOMER object to satisfy the precondition of the creation routine

Postconditions

Precondition: obligation for clients

Postcondition: benefit for clients

```
extend (a_element: G)
    ensure
        inserted: i_th (count) = a_element


index (a_element: G): INTEGER
    ensure
        exists: result > 0 implies i_th (result)  = a_element
        no_exists: result = -1 implies not is_inserted (a_element)
```

Let's code…

Go to:

https://codeboard.io/projects/86

Task: in class CUSTOMER, write a postcondition for the creation routine make_with_name_and_age

Task: modify the implementation of make_with_name_and_age such that it breaks your postcondition. Run the program. What happens?

Old notation

Usable in postconditions only

Denotes value of an expression as it was on routine entry

Example (in a class ACCOUNT):

```
balance : INTEGER
            -- Current balance.

deposit (v : INTEGER)
            -- Add v to account.
      require
            positive: v > 0
      do
            …
      ensure
            added: balance = old balance + v
      end
```

Postcondition principle

A feature must make sure that, if its precondition held at the beginning of its execution, its postcondition will hold at the end.

A feature that fails to ensure its postcondition is buggy software.

Invariant

An invariant states properties about an object that are true

- **after** the object has been initialized
- **before and after** every routine call
  (but not necessarily in between a call)

The invariant is listed after the last feature block.

A class with no **invariant** is the same a

```
invariant
    always_OK: True
```

Example (from class ARRAY):

```
invariant
    area_exists: area /= Void
    consistent_size: capacity = upper - lower + 1
    non_negative_count: count >= 0
    index_set_has_same_count: valid_index_set
```

A class with contracts

```
class
    BANK_ACCOUNT
create
    make
feature
   make (n : STRING)
            -- Set up with name n

        require
            n /= Void

        do
            name := n
            balance := 0

        ensure
            name = n
        end

   name : STRING

   balance : INTEGER

   deposit ( v : INTEGER)
            -- Add amount v
      do
         balance := balance + v
      ensure
         balance = old balance + v
      end

invariant
   name /= Void
   balance >= 0

end
```

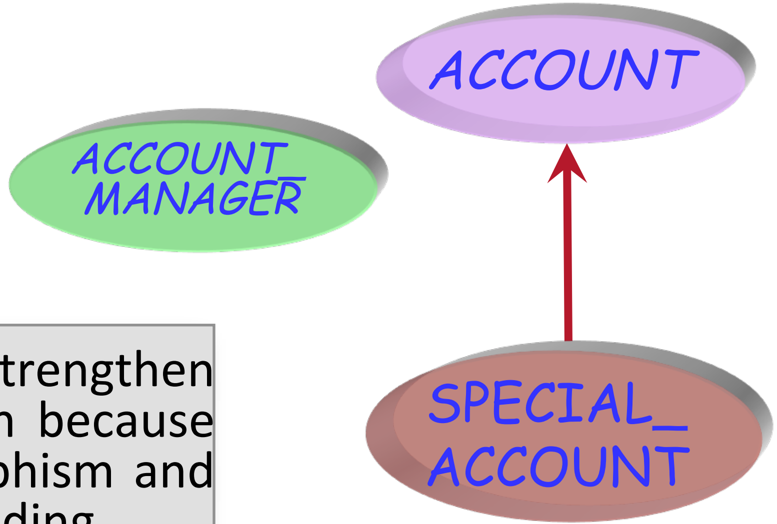
Let's code…

Go to:

https://codeboard.io/projects/86

Task: in class ACCOUNT, replace all
-- Important: …
comments with contracts

Contracts and inheritance (Example)




```
class
  ACCOUNT_MANAGER

feature -- Operations

  init_new_account(a_acc: ACCOUNT)

    do
      -- do all initialization
      a_acc.set_balance(0)
    end
```

Must not strengthen precondition because of polymorphism and dynamic binding.

```
class
  ACCOUNT

feature -- Operations

  set_balance(a_balance: DOUBLE)
    require
     non_neg: a_balance >= 0
    do
      balance := a_balance
    end
```

```
class
  SPECIAL_ACCOUNT
inherit
ACCOUNT redefine set_balance end

feature -- Operations

  set_balance(a_balance: DOUBLE)
    require
     min_bal: a_balance > 100
    do
      balance := a_balance
    end
```

Contracts and inheritance

When redeclaring a routine, we may only:

Keep or weaken the precondition

Keep or strengthen the postcondition

Invariant Inheritance rule:

The invariant of a class automatically includes the invariant clauses from all its parents,

“and”-ed.

Assertion redeclaration rule in Eiffel

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*