The Eiffel language

Slides partly based on:
http://se.inf.ethz.ch/courses/2015b_fall/eprog/english_index.html
Eiffel, in brief

- Procedural, object-oriented programming language created by Bertrand Meyer in 1986
- Pascal-like syntax ("old-school" language ?)
- Static typing
- Concepts related to the language: uniform-access principle, Hoare logic, design by contract, multiple inheritance, and many more
Quick illustration of the syntax

factorial (n: INTEGER): INTEGER

require

   legal_input: n >= 0

do

   if n = 0 then
      Result := 1
   else
      Result := n * factorial (n - 1)
   end
end

Reference: https://www.rosettacode.org/wiki/Factorial#Eiffel

factorial_2 (n: NATURAL): NATURAL

local

   v: like n  -- or v: NATURAL

do

   from

      Result := 1

   v := n

   until

      v <= 1

   loop

      Result := Result * v

      v := v - 1

   end
end
Multiple inheritance

- Classes can (should ?) inherit from multiple different classes
- Not always directly supported in many languages
- Java, Scala can “simulate” it (interfaces, traits...)
- In Eiffel, multiple inheritance is embarrassingly easy
Multiple inheritance: Name clashes

Diagram:

- Node A with an edge labeled f to Node B
- Node B with an edge labeled f to Node C
- Node C

Question mark (?) indicating potential name clash.

Legend:
- f: Inheritance link
- ?: Potential name clash indication
Multiple inheritance: Name clashes

rename f as A_f
Multiple inheritance: Name clashes

Given:

a1: A  b1: B  c1: C

What is valid?

- c1.f ✔
- a1.A_f ✘
- c1.A_f ✔
- b1.f ✘
- b1.A_f ✘

```
class -- The code
  C
  inherit
    A
    rename
      f as A_f
  end
B

rename f as A_f
```
Multiple inheritance: Name clashes

Given:

a1: A  b1: B  c1: C

What is valid?

c1.f ✔
a1.A_f ✘
c1.A_f ✘
b1.f ✘
b1.A_f ✘

class -- The code

C
inherit
A
rename
f as A_f
end

B
class C
renaming f as A_f

A_f, f
Multiple inheritance: Name clashes

Given:

\( a_1: A \quad b_1: B \quad c_1: C \)

What is valid?

- \( c_1.f \) ✔
- \( a_1.A_f \) ✘
- \( c_1.A_f \) ✘
- \( b_1.f \) ✘
- \( b_1.A_f \) ✘

The code:

```plaintext
class -- The code
  C
inherit
  A
  rename
    f as A_f
end
B

feature...
  ...
end
```
Multiple inheritance: Name clashes

Given:

a1: A  b1: B  c1: C

What is valid?

c1.f  ✔
a1.A_f  ✘
c1.A_f  ✔
b1.f  ✘
b1.A_f  ✘

```
class -- The code
  C
  inherit
    A
    rename
      f as A_f
  end
B
feature
  ...
end
```
Given:

a1: A  b1: B  c1: C

What is valid?

c1.f ✔

a1.A_f ✘

c1.A_f ✔

b1.f ✔

b1.A_f
Multiple inheritance : Name clashes

Given:

a1: A       b1: B       c1: C

What is valid?

c1.f ✔

a1.A_f ✘

c1.A_f ✔

b1.f ✔

b1.A_f ✘

```class          -- The code
   C
  inherit
    A
    rename
      f as A_f
  end

B

feature...

end```
More name clashes

How could we resolve the name clashes for functions “switch_on” and “send”? 
A possible solution

class ALL_IN_ONE_DEVICE
inherit
  PRINTER
    rename
      switch_on as start
    undefine
      start
    end
SCANNER
    rename
      switch_on as start,
      send as send_data
    end
FAX
    rename
      send as send_message
    undefine
      start
    end
feature ... end

-- Given the following :
s : SCANNER
f : FAX
a : ALL_IN_ONE_DEVICE

-- Which ones are valid ?
a.switch_on
a.print_page
f.send_message
s.switch_on
f.send
a.send
A possible solution

```ruby
class ALL_IN_ONE_DEVICE
  inherit PRINTER
    rename
      switch_on as start
    undefined
      start
    end
  SCANNER
    rename
      switch_on as start,
      send as send_data
    end
  FAX
    rename
      send as send_message
    undefined
      start
    end
feature ... end

-- Given the following :

s : SCANNER

f : FAX

a : ALL_IN_ONE_DEVICE

-- Which ones are valid ?

a.switch_on  no/yes (more info needed)
a.print_page

f.send_message

s.switch_on

f.send

a.send
```
A possible solution

class ALL_IN_ONE_DEVICE
inherit
PRINTER
    rename
        switch_on as start
    undefine
        start
end
SCANNER
    rename
        switch_on as start,
        send as send_data
end
FAX
    rename
        send as send_message
    undefine
        start
end

-- Given the following :

s : SCANNER

f : FAX

a : ALL_IN_ONE_DEVICE

-- Which ones are valid ?

a.switch_on no/yes (more info needed)

a.print_page yes/no (more info needed)

f.send_message

s.switch_on

f.send

a.send

feature ... end
A possible solution

class ALL_IN_ONE_DEVICE
inherit
  PRINTER
    rename
      switch_on as start
    undefine
      start
    end
SCANNER
  rename
    switch_on as start,
    send as send_data
  end
FAX
  rename
    send as send_message
  undefine
    start
  end
feature ... end

-- Given the following :
s : SCANNER
f : FAX
a : ALL_IN_ONE_DEVICE

-- Which ones are valid ?
a.switch_on  no/yes  (more info needed)
a.print_page  yes/no  (more info needed)
f.send_message  no
s.switch_on
f.send
a.send
A possible solution

class ALL_IN_ONE_DEVICE
  inherit
    PRINTER
      rename
        switch_on as start
      undefine
        start
    end
  SCANNER
    rename
      switch_on as start,
      send as send_data
    end
  FAX
    rename
      send as send_message
    undefine
      start
    end
feature ... end

-- Given the following:

s : SCANNER
f : FAX
a : ALL_IN_ONE_DEVICE

-- Which ones are valid?

a.switch_on   no/yes   (more info needed)
a.print_page  yes/no   (more info needed)
f.send_message   no
s.switch_on yes
f.send
a.send
A possible solution

```ruby
A possible solution

class ALL_IN_ONE_DEVICE
  inherit
    PRINTER
      rename
        switch_on as start
      undefine
        start
    end
  SCANNER
    rename
      switch_on as start,
      send as send_data
    end
  FAX
    rename
      send as send_message
    undefine
      start
    end
  feature ... end

-- Given the following :

s : SCANNER
f : FAX
a : ALL_IN_ONE_DEVICE

-- Which ones are valid ?

a. switch_on  no/yes  (more info needed)

a. print_page  yes/no  (more info needed)

f. send_message  no

s. switch_on  yes

f. send  yes

a. send
```
A possible solution

class ALL_IN_ONE_DEVICE
inherit
   PRINTER
   rename
       switch_on as start
   undefine
       start
   end
SCANNER
   rename
       switch_on as start,
       send as send_data
   end
FAX
   rename
       send as send_message
   undefine
       start
   end
feature ... end

-- Given the following :
s : SCANNER
f : FAX
a : ALL_IN_ONE_DEVICE

-- Which ones are valid ?
a.switch_on  no/yes  (more info needed)
a.print_page  yes/no  (more info needed)
f.send_message  no  
s.switch_on  yes
f.send  yes
a.send  no
We are not done yet ...

- There is more to multiple inheritance than just renaming names
- Other issues: repeated inheritance (diamond problem), feature merging, ...
- And what about polymorphism?
- For more details about how Eiffel deals with these, see the (free) online course “Introduction to Programming” by Bertrand Meyer (ETH Zürich)
Design by contract

- Idea: use assertions (i.e. contracts) to ensure that your program is "correct"
- Based on Hoare logic
- Different types of contracts: preconditions, postconditions, loop variants, loop invariants, class invariants
Design by contract: example of pre/post-conditions

deposit (amount: INTEGER)

    require -- Preconditions
    amount >= 0

do

    balance := balance + amount

ensure -- Postconditions

    correct_balance : balance = old balance + amount
    arithmetic_still_works : 2+2=4

end

See http://comcom.csail.mit.edu/comcom/#AutoProof
Design by contract: example of loop variants, invariants

sum (n : INTEGER) : INTEGER
-- computes 0+1+2+...+n
require
  --???
do
  from
    Result := 0
    i := 0
  invariant
    --???
  variant
    --???
  until
    i >= n
loop
  i := i + 1
  Result := Result + i
end
ensure
  --???
end

Loop variants are integer expressions that:
- Must be non-negative at initialization
- Must **decrease** at each iteration while remaining non-negative

Loop invariants are boolean expressions that:
- Must hold before entering the loop body
- Must hold at each iteration of the loop
- Must hold immediately after exiting the loop
Design by contract: example of loop variants, invariants

\[ \text{sum} \ (n : \text{INTEGER}) : \text{INTEGER} \]
\[ \text{-- computes } 0+1+2+...+n \]
\[ \text{require} \]
\[ n \geq 0 \]
\[ \text{do} \]
\[ \text{from} \]
\[ \text{Result} \ := \ 0 \]
\[ i \ := \ 0 \]
\[ \text{invariant} \]
\[ --??? \]
\[ \text{variant} \]
\[ --??? \]
\[ \text{until} \]
\[ i \geq n \]
\[ \text{loop} \]
\[ i \ := \ i + 1 \]
\[ \text{Result} \ := \ \text{Result} + i \]
\[ \text{end} \]
\[ \text{ensure} \]
\[ --??? \]
\[ \text{end} \]

Loop variants are integer expressions that:

- Must be non-negative at initialization
- Must **decrease** at each iteration while remaining non-negative

Loop invariants are boolean expressions that:

- Must hold before entering the loop body
- Must hold at each iteration of the loop
- Must hold immediately after exiting the loop
Design by contract: example of loop variants, invariants

```
sum (n : INTEGER) : INTEGER
  -- computes 0+1+2+...+n
require
  n >= 0
do
  from
    Result := 0
    i := 0
invariant
  i <= n
variant
  --???
until
  i >= n
loop
  i := i + 1
  Result := Result + i
end
ensure
  --???
end
```

Loop variants are integer expressions that:

- Must be non-negative at initialization
- Must **decrease** at each iteration while remaining non-negative

Loop invariants are boolean expressions that:

- Must hold before entering the loop body
- Must hold at each iteration of the loop
- Must hold immediately after exiting the loop
Design by contract: example of loop variants, invariants

```
sum (n : INTEGER) : INTEGER
   -- computes 0+1+2+...+n
require
   n >= 0
do
   from
   Result := 0
   i := 0
invariant
   i <= n
   Result = (i+1)*i/2
variant
   --???
until
   i >= n
loop
   i := i + 1
   Result := Result + i
end
ensure
   --???
end
```

Loop variants are integer expressions that:

- Must be non-negative at initialization
- Must **decrease** at each iteration while remaining non-negative

Loop invariants are boolean expressions that:

- Must hold before entering the loop body
- Must hold at each iteration of the loop
- Must hold immediately after exiting the loop
Design by contract: example of loop variants, invariants

```
sum (n: INTEGER): INTEGER
    -- computes 0+1+2+...+n
require
    n >= 0
do
    from
    Result := 0
    i := 0
invariant
    i <= n
    Result = (i+1)*i/2
variant
    n - i
until
    i >= n
loop
    i := i + 1
    Result := Result + i
end
ensure
    -- ???
end
```

Loop variants are integer expressions that:

- Must be non-negative at initialization
- Must **decrease** at each iteration while remaining non-negative

Loop invariants are boolean expressions that:

- Must hold before entering the loop body
- Must hold at each iteration of the loop
- Must hold immediately after exiting the loop
Design by contract: example of loop variants, invariants

\[
\text{sum (} n : \text{INTEGER}) : \text{INTEGER} \\
\text{-- computes 0+1+2+...+n}
\text{require} \\
n \geq 0
\text{do}
\text{from} \\
\text{Result} := 0 \\
i := 0
\text{invariant} \\
i \leq n \\
\text{Result} = (i+1) \times i/2 \\
\text{variant} \\
n - i
\text{until} \\
i \geq n
\text{loop} \\
i := i + 1 \\
\text{Result} := \text{Result} + i
\text{end}
\text{ensure} \\
\text{Result} = (n+1) \times n/2
\text{end}
\]

Loop variants are integer expressions that:

- Must be non-negative at initialization
- Must decrease at each iteration while remaining non-negative

Loop invariants are boolean expressions that:

- Must hold before entering the loop body
- Must hold at each iteration of the loop
- Must hold immediately after exiting the loop
But, why should we care of contracts ?

- Documentation: it makes the code easier to read; it highlights what the programmer wanted to do and what the code should compute.
- Program correctness
- If the loop invariants (and other contracts) are strong enough, we can even prove that a program is correct and terminates, with the help of a proof system (Hoare logic)
- Example of projects: Dafny (Microsoft), Autoproof (Eiffel) -> Online tools that try to prove at compile time that your code terminates and is correct. If the proof attempt fails, then we cannot infer whether your program is correct/terminates or not.
The end