Programming Languages

Eiffel

Benjamin J. Keller
Department of Computer Science, Virginia Tech
Overview

- Classes and Objects
- Built-in types, constants and operations
- Expressions, instructions, procedures and functions, and programs
- Input and output
- Assertions
- Access Control
• Designed as object-oriented language (Bertrand Meyer)

• Goal to support OO design goals

• *Programming by Contract* — use pre- and post-conditions for functions

• Idea is to develop clusters of classes that work together

• Clusters can then be combined in different ways to build a variety of systems
Classes

- Defines type of objects
- Identifies creation procedures
- Defines features: attributes, procedures and functions

```eiffel
class CNAME
creation
  -- names of creation procedures
  -- optional

feature
  -- declaration or definition
  -- of attributes or routines
end -- class CNAME
```
**Entities and Objects**

- Objects have type determined by class
- Reference to an object called an entity
- Names (identifiers) — begin with a letter possibly followed by letters digits and the underline character
- Reference declaration — reference with name x has type T
  \[x : T\]
- Assignment binds reference \(x\) to an object that conforms to type T
  \[x := y\]
- Creates an alias
Procedures

- Abstracted compound statement
  
  \[ \text{pname} \ ( \ \text{arg1} : \ T1; \ \text{arg2} : \ T2) \ \text{is} \]
  
  local
  
  \[ x : \ T3 \quad -- \ \text{local declarations} \]
  
  do
  
  \[ c \quad -- \ \text{procedure body} \]
  
  end

- Not allowed to assign to formal parameters

- Procedure call \[ \text{pname}(\text{e1}, \text{e2}) \]

- Passes reference
Every class has as feature an object \texttt{io}

Send messages to \texttt{io} to I/O

\begin{verbatim}
io.put_character('A')
io.put_string("Hello World.\%N")
io.put_integer(12)
io.put_integer_format(cnt, len)
io.put_real(2.83)
io.put_real_format(dist, precision)
io.flush
\end{verbatim}

Look at \texttt{output_stream.e} in SmallEiffel \texttt{lib_std} directory
Compound Instructions

- Sequence of instructions
  
  \[ x := y \]
  
  \[ \text{io.put_string("a message")} \]

- Compound can be empty

- Can put optional semicolon after each instruction, but common style is to leave them out
Assertions

Compiler can generate code to check pre- and post-conditions

```
pname ( args ) is
    require
        -- preconditions (Boolean expressions)
    local
        -- local declarations
    do
        -- body
    ensure
        -- postconditions
end
```
Example Class — Creation

The following should be stored in a file `point.e`.

```e
class POINT -- class which supports a movable point
creation -- designates a method to create a POINT object.
    Create

feature

    Create (lp: LINKED_STACK [POINT]) is
        -- Create point at origin and push it onto ‘lp’
        require
            lp /= Void
        do
            lp.put (Current) -- Current is the object
        end; -- Create
```
Example Class — Features

x, y: REAL; -- attributes of class

translate (a, b: REAL) is
    -- Move by ‘a’ horizontally, ‘b’ vertically.
    do
        x := x + a
        y := y + b
    end; -- translate

scale (factor: REAL) is
    -- Scale by a ratio of ‘factor’.
    do
        x := factor * x
        y := factor * y
    end; -- scale
display is
  -- Output position of point
  do
    io.put_string ("Current position: x = ")
    io.put_real (x)
    io.put_string ("; y = ")
    io.put_real (y)
    io.new_line
  end -- display
end -- class POINT
Feature Access Control

- `feature` clause can be qualified by comma-separated list of classes

- Saying
  ```
  feature
  ```

- Is equivalent to
  ```
  feature { ANY }
  ```

- Hide features outside of class/object
  ```
  feature { NONE }
  ```

- To enable access by other objects of same class A
  ```
  feature { A }
  ```
- Unique constants — analogous to enumerated type
  
  red : INTEGER is unique
  blue : INTEGER is unique
  green : INTEGER is unique
  black : INTEGER is unique

- Compiler determines unique value within enclosing class
Creating Objects

• Following declaration

\[ x : T \]

• Can bind \( x \) to newly created object

\[ \#x \]

• Call creation procedure

\[ \#my_point.Create (point_stack) \]
Iteration

- General form
  
  from
c1    -- loop initialization
until
  b    -- exit condition
loop
  c2  -- loop body
end

- Semantics: execute c1, test b, if false then execute c2, test b ...

- Exit when b becomes true
Input

- Read routines save result in a variable
  
  ```e
  io.read_integer
  result := io.last_integer
  ```

- Look at `input_stream.e` in SmallEiffel `lib_std` directory
Multi-Branch Instruction

- Select action based on value of INTEGER or CHARACTER

    inspect input_char
    when 'A' .. 'Z' then
        ch_type := Upper_case
    when 'a' .. 'z' then
        ch_type := Lower_case
    when ',', ';', ':', '.', '?', '!' then
        ch_type := Punctuation
    else
        ch_type := Special
    end

- Sets must be disjoint. Must have else if not complete
Example Class — Access Control

In a separate file: interaction.e

class INTERACTION
creation
  Create

feature {NONE}  -- private features

  my_point: POINT;
  request: INTEGER;
  Up, Down, Left, Right, Quit: INTEGER is unique;

  point_stack: LINKED_STACK [POINT];  -- from library
Example Class — Object Creation

feature -- since no qualifier, these are public

  over: BOOLEAN;

Create is   -- Create a point
  do
    !!point_stack.make; -- create and execute "make".
    !!my_point.Create (point_stack);
  end;    -- Create
get_request is

   -- Ask what the user wants to do next, returning the answer
   -- in attribute ‘request’:

local

   answer: CHARACTER;
   correct: BOOLEAN -- default value is false

do

   -- beginning of loop
Example Class — Iteration

from
until -- continue until condition is true
    correct
loop
    io.new_line;
    io.putstring ("Enter command (one character)");
    io.new_line;
    io.putstring ("U for Up, D for Down, L for Left, %
        %R for Right, Q for Quit: ");
        -- % indicates continue string to next line
    io.read_char;
    answer := io.last_char;
    io.next_line;
    correct := true;
Example Class (cont) — Iteration

inspect       -- like a case statement.
    answer
when 'u', 'U' then
    request := Up
when 'd', 'D' then
    request := Down
when 'l', 'L' then
    request := Left
when 'r', 'R' then
    request := Right
when 'q', 'Q' then
    request := Quit
else
    io.new_line;
    io.putstring ("Bad code. Please enter again.");
    io.new_line;
    correct := false
end       -- end inspect
end       -- until correct
end;       -- get_request
Eiffel Example (cont)

one_command is
    -- Get user request and execute it
    do
        get_request;
        inspect request
        when Up then
            my_point.translate (0., 1.)
        when Down then
            my_point.translate (0., -1.)
        when Left then
            my_point.translate (-1., 0.)
        when Right then
            my_point.translate (1., 0.)
        when Quit then
            over := true
    end;
    my_point.display
end -- one_command

end -- class INTERACTION
• Program is a collection of classes
• Designate a *root class* and creation procedure to compiler
• Creation procedure serves as “main” procedure
class SESSION
creation
  Create
feature
  Create is
    -- Execute sequence of interactive commands
    local
      interface: INTERACTION
    do
      from
        !!interface.Create
      until
        interface.over
    loop
      interface.one_command
    end
  end -- Create
end -- class SESSION
Compilation Control for Example

Build is controlled by an “ACE” file such as

```plaintext
system pointshift root
    SESSION : "create"
default
    assertion (require) -- only check preconditions
cluster
    application : "../"
    standard : "${SmallEiffelDirectory}lib_std"
        default
            assertion (require)
    option
        -- To override the previously defined level
        -- for some classes of this cluster:
            assertion (no): FIXED_ARRAY
            assertion (require): STRING, STD_INPUT
    end
end -- system pointshift
```
Built-in Types

- Primitive types:
  - BOOLEAN
  - CHARACTER
  - INTEGER
  - REAL
  - DOUBLE
- Composite types: ARRAY, STRING
Default Values

- Default values for primitive types
  - BOOLEAN  false
  - CHARACTER  ’%U’ (null character)
  - INTEGER  0
  - REAL  0.0

- Default value for reference of other types is void

- Unbind x with
  - x := void
### Constants

max_size : INTEGER is 4096
pi : REAL is 3.14159
warning : STRING is "Watch out!"
vector : ARRAY [INTEGER] is <<3, 4, 5, -2, 7>>
Operations

- Equality (=) and inequality (/=) return BOOLEAN
- Equality tests if references bound to same object
- For primitive types
  - BOOLEAN: not, and, or, implies, or else, and then
  - INTEGER: +, -, *, //, \\\n  - REAL: +, -, *, /, ^, <, >, <=, >=
- Integer division //, Modulus \\n
Equality

- Operator = returns true if references bound to same object
- Function `equal(obj1, obj2)` returns true if
  - both arguments are of same type
  - the attributes of both arguments are identical (using =)
- Two objects could be `o1 /= o2` but `equal(o1, o2)`
# Precedence Rules

<table>
<thead>
<tr>
<th>Level</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>9</td>
<td>old, not, unary +, unary -, free unary operators</td>
</tr>
<tr>
<td>8</td>
<td>free binary operators</td>
</tr>
<tr>
<td>7</td>
<td>^</td>
</tr>
</tbody>
</table>
| 6     | *, /, //, \\
| 5     | +, -     |
| 4     | =, /=, <, >, <=, >= |
| 3     | and, and then |
| 2     | or, or else |
| 1     | implies  |
Feature Access

- If class B has attribute a access attribute of x:B as x.a
- Cannot assign to attribute outside of class — must have mutator
- If feature is procedure, provide arguments
- Creation procedure (b:BOOK)

!!b.make(i,a,t)
In the context of a class A, an expression is
- Attribute of A: a
- Function of A (no arguments): f
- Function of A (with arguments): f(e1,e2,...,e3)
- Feature x of class C for accessed through expression e: e.x
- Expression with infix operation: a + b

Expressions are evaluated from left to right

Operations and then and or else are short-circuited
Instructions

- Object creation
- Assignment
- Loop
- Conditional
- Multi-Branch
Conditional Instruction

- General form
  ```
  if b1 then
    c1
  elseif b2 then
    c2
  else
    ce
  end
  ```

- Valid to say to have empty else clause, but also can say
  ```
  if x < y then
    min := x
  end
  ```
Debug Instruction

- Execute instructions when debugging is turned on
  
  debug(key1, ..., keyn)
  
  c
  
  end

- No effect is debugging is off

- Selective debugging uses keys — arbitrary strings

- If one key is on then c is executed.
### Functions

- Similar syntax to procedure
- Must have return type, and at least one assignment to `result`

```eiffel
gcd (m, n : INTEGER) : INTEGER is
  do
    if n = 0 then
      result := m
    else
      result := gcd(n, m \ n)
    end
  end

- `local` is optional
External Routines

- Using non-Eiffel code

```eiffel
pname(arg1 : T1; ...) is
  external "C" -- C language program
  alias "other_name" -- Alias within Eiffel
end
```

- Example:

```eiffel
integer2c (i: INTEGER) is -- Send an INTEGER to C
  external "C"
end;
```

- C source

```c
void integer2c(int i){
  printf("%d\n",i);
}
```
One Time Routines

- Replace do with once in procedure definition and will only be invoked once no matter how many times it is called.

```
init is
  once
    -- code that should only be done one time
end
```

- Once functions always return same value after first call

```
f (x : T) : U is
  once
    -- code computed first time
end
```
Assertions

- Goal is to allow “programming by contract”
- Assertions are Boolean expressions that can be checked at runtime
- Labels for assertions will appear in error messages if fails
- Kinds of assertions
  - Pre- and Postconditions of routines
  - Loop invariants
  - Object invariants
- Can also put sequence of conditions in `check` command
Pre- and Post-Conditions in Routines

pname ( args ) is
  require
    -- preconditions (Boolean expressions)
  local
    -- local declarations
  do
    -- body
  ensure
    -- postconditions
end
Conditions in Loops

from
   -- the initialization
invariant
   -- loop invariant
variant
   -- INTEGER expression that decreases each iteration
until
   -- termination test
loop
   -- body
end
Example

binary_search (a : ARRAY [ELEMENT], x : ELEMENT) is
  require
    non_trivial : a /= void and then a.count > 0
    is_sorted : -- a is sorted in increasing order
  local
    low, mid, high : INTEGER
  do
Example

from
    low := a.lower - 1
    high := a.upper + 1
invariant
    a.lower - 1 <= low and then low < high and then
    high <= a.upper + 1
    low = a.lower - 1 or else a.item(low) <= x
    high = a.upper + 1 or else x < a.item(high)
variant
    high - low
until
    low = high - 1
loop
    mid := (low + high) // 2
    if a.item(mid) <= x then
        low := mid
    else
        high := mid
end
end
Example

check

  low >= a.lower - 1 and then low <= a.upper
  low = a.lower - 1 or else a.item(low) <= x
  low = a.upper or else x < a.item(low+1)

end

found := (low >= a.lower and then x <= a.item(low))
if found then
  index := low
end

ensure

  found : found implies (x <= a.item(index)
    and then
    x >= a.item(index))

end
Exceptions (and Assertions)

- Exceptions are raised by the failure of an assertion
- Routine has a `rescue` clause to handle exception
- Handle exception by restoring invariant state of object and calling `retry`
- `retry` resumes at the beginning of the routine
- Exception propagated if no `rescue` clause or end of clause is reached
get_integer (msg : STRING) is
    do
        put_string(msg)
        get_string
        the_integer := fmt.s2i(last_string)
    rescue
        put_string("Not an integer. %N")
    retry
    end
Class Invariants

• Condition that must be true between executions of routines

class CNAME
creation
  -- list of creation procedures
feature
  -- list of features
invariant
  -- list of class invariants
end

• Routine can assume preconditions and class invariant

• Routine must ensure postconditions and class invariant
class COMPLEX
inherit
    MATH -- import math routines (blech!)

creation
    make_rect, make_polar

feature
    x : REAL -- real part
    y : REAL -- imaginary part

    epsilon : REAL is 1.0E-6
make_rect (r, i : REAL) is
   do
      x := r
      y := i
   end
---------------------------------------------
make_polar (r, t: REAL) is
   do
      x := r * cos(t)
      y := r * sin(t)
   end
infix "+" (other : COMPLEX) : COMPLEX is
   do
      !!result.make_rect(x + other.x, y+other.y)
   end
end

infix "*" (other : COMPLEX) : COMPLEX is
   do
      !!result.make_polar(rho * other.rho, theta + other.theta)
   end
Class Invariant Example

rho is
do
do
    sqrt(x^2 + y^2)
end
---------------------------------------------
theta is
do
do
    arctan2(y,x)
end
---------------------------------------------
invariant
    consistent : abs(x, rho * cos(theta)) < epsilon
    and then
    abs(y, rho * sin(theta)) < epsilon
end -- class COMPLEX
Subclasses and Inheritance

- Declare subclass using `inherit`
- New class then inherits all features of the old class and can
  - add new features
  - rename inherited features
  - redefine inherited features
- Multiple inheritance — list all parent classes
Inheritance Example

class NEWRATIONAL

inherit

  ORDERED_PAIR2 [INTEGER]
  
    rename  x as n, -- can change names of features
            y as d
    redefine same  -- indicates that same will be redefined.
          -- Need not mention display since it was deferred!

end

creation Create

feature

  Create is
    -- create a rational
    do
      d := 1
    end; -- Create
Inheritance Example (cont)

feature {NONE} -- private method
    reduce : INTEGER is
        -- reduce to lowest terms
        local
            num, den, next : INTEGER
        do
            if (n = 0) or (d = 0) then
                Result := 1
            else
                if n < 0 then num := -n else num := n end;
                if d < 0 then den := -d else den := d end;
            from
                next := num \\ den -- \\ is mod operator
            invariant -- must be true each time through loop
                ((num \\ next) = 0) and ((den \\ next) = 0)
            variant -- must decrease each time through loop
                next
            until
                next = 0
            loop
                num := den;
                den := next;
                next := (num \\ den)
            end;
        end
        Result := den
    end
end; -- reduce
Inheritance Example (cont)

feature
    set(numer, denom : INTEGER) is
        -- set the numerator and denominator
        -- post: d > 0
        require -- precondition
            denom /= 0
        local
            gcd : INTEGER
        do
            n := numer;
            d := denom;
            if d < 0 then
                n := -n;
                d := -d
            end;
            gcd := reduce;
            n := n // gcd;
            d := d // gcd
        ensure -- postcondition
            d > 0
        end; -- set
Inheritance Example (cont)

read is
  -- get rational in form n/d from input
local
  num, den, attempts : INTEGER
do
  io.readint; num := io.lastint;
  io.readchar; io.readint; den := io.lastint;
  set(num,den)
ensure
  d > 0
rescue  -- exception handler
  if attempts < 3 then
    io.next_line;  -- go to next input line
    io.new_line;   -- go to next output line
    io.putstring("A fraction is an integer ");
    io.putstring("divided by a non-zero integer.");
    io.putstring(" Enter a fraction: ");
    attempts := attempts + 1;
    retry
  end
end;  -- read
display is                        -- display the fraction
  do
    if n = d*(n // d) then
      io.putint(n // d)
    else
      io.putint(n);
      io.putchar('/');
      io.putint(d)
    end
  end; -- display
same(other : like Current) : BOOLEAN is                        -- are the fractions equal?
  do
    Result := (n*other.d = d*other.n)
  end; -- same
lessthan(other : like Current) : BOOLEAN is                        -- is Current < other
  do
    Result := (n*other.d < d*other.n)
  end;
invariant
  d /= 0
end -- NEWRATIONAL
Parameters and Inheritance

- The parameter type like Current in less than refers to the class of the object that is receiving the message.
- Can also use like x for x any instance variable of class
- Declaring class to be like Current helps ensure that routine will work properly in subclasses — guarantees that class of the argument is the same as class of object sending message to
Subclassing

class RATIONALMATH
inherit
   NEWRATIONAL

creation
   Create

feature
   plus(other : like Current) : like Current is
      local
         sumnum, sumden : INTEGER;
      do
         sumnum := n*other.d + other.n*d;
         sumden := d*other.d;
         !!Result.Create;
         Result.set(sumnum,sumden)
      end; -- plus

   -- add other operations here
end -- RATIONALMATH
Program using RATIONALMATH

class TESTRATIONAL
creation
  Create
feature
  Create is
    -- manipulate some rational numbers
local
  p1,p2,p3 : RATIONALMATH
do
  !!p1.Create;
  !!p2.Create;
  !!p3.Create;
  io.putstring("Enter a fraction as n/d: ");
  p1.read;
  io.putstring("Enter a fraction as n/d: ");
  p2.read;
  p1.display;
  io.new_line;
  p2.display;
  io.new_line;
  if p1.same(p2) then
    io.putstring("They’re equal")
  else
    io.putstring("They’re not equal")
  end
  io.new_line;
end -- Create
end -- TESTRATIONAL
The Inherit Clause

- Eiffel supports multiple inheritance
- Requires resolving name clashes
- Options to `inherit` clause (must occur in this order):
  1. `rename` – change the name of inherited features, helpful if have name clashes in multiple inheritance
  2. `export` – change the export status of inherited features
  3. `undefine` – used to resolve name clashes in multiple inheritance
  4. `redefine` – indicate that inherited feature will be redefined
  5. `select` – indicate which method is to use if there are two methods with same name
Renaming

- Suppose feature \( m \) is defined in class \( A \) and have

```eiffel
class B
    inherit A
    rename
        m as k
    end;
feature ... 
```

- Now suppose that \( x:A \), but at run time \( x \) holds a value of type \( B \)
  static type of \( x \) is \( A \), but dynamic type is \( B \)

- By static type-checking, \( x.m \) should be defined. What is actually executed?

- Answer: method \( k \) of \( B \)
Redefining

- Redefinition of \( m \) in \( B \), using inherited definition:

  ```eiffel
  class B
    inherit A
    rename
      m as old_m
    redefine m
    end;
  feature
    m(...) is
      do ... old_m ... end;
  ```

- What happens if have \( x:A \) holding a value of type \( B \), and execute \( x.m \)?

- Answer: the renamed version of \( m \) (really \( \text{old}_m \)) will be executed.
Redefining

- To get desired behavior inherit twice:

```plaintext
class B
    inherit A
    rename
        m as old_m
    inherit A
    redefine
        m
    select
        m
    end;
feature
    m(...) is
        do ... old_m ... end;
```

- **Select** says which of two definitions of `m` to use
- The **select** clause says to resolve the ambiguity by taking the `m` from the second version (which is redefined in the class!).
class LINKABLE [G]

feature

  item: G;
  
  right: like Current;  -- Right neighbor

  put_right (other: like Current) is
  -- Put ‘other’ to the right of current cell.
  do
    right := other
  ensure
    chained: right = other
  end;

end -- class LINKABLE
Constrained Genericity

- Restriction on class parameter by inheritance
- Abstract class
  
  ```
  deferred class interface COMPARABLE
  feature specification
    infix "<" (other: like Current): BOOLEAN is deferred;

    infix "<=" (other: like Current): BOOLEAN is deferred;

    infix ">" (other: like Current): BOOLEAN is deferred;

    infix ">=" (other: like Current): BOOLEAN is deferred;
  
  end interface -- class COMPARABLE
  ```
Constrained Genericity (cont)

- Now define

  class INTORD
  feature
    value:INTEGER;
    infix "<"(other:like Current) is
      do
        Result := value < other.value
      end;
    ...
  end -- class INTORD

- Can use in

  class Sorting[T -> COMPARABLE]
  feature
    sort(thearray:ARRAY[T]):ARRAY[T] is
      local ....
      do
        ....
        .... if thearray.item(i) < thearray.item(j) ....
      end;