Programming Languages

Eiffel

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Overview

- Classes and Objects
- Builtin 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

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
    pname (arg1 : T1; arg2 : T2) is
    local
        x : T3 -- local declarations
        do
        c -- procedure body
        end
```

- Not allowed to assign to formal parameters
- Procedure call pname(e1,e2)
- Passes reference

Output

- Every class has as feature an object io
- $\bullet\,$ Send messages to io to I/O

io.put_character('A')
io.put_string("Hello World.%N")

io.put_integer(12)

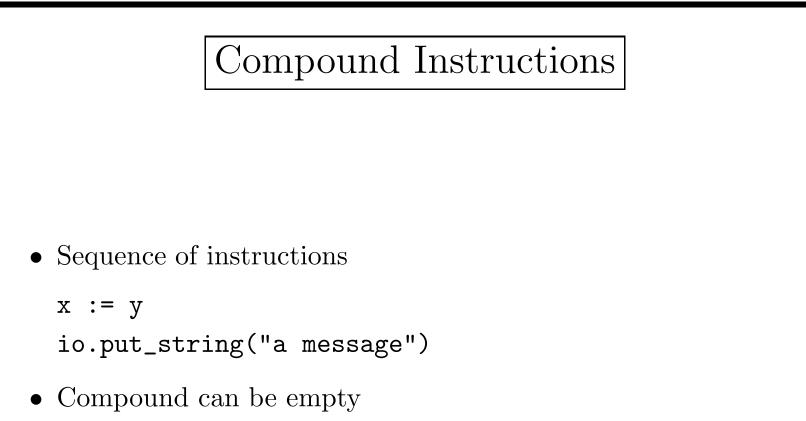
io.put_integer_format(cnt, len)

io.put_real(2.83)

io.put_real_format(dist, precision)

io.flush

• Look at output_stream.e in SmallEiffel lib_std directory



• 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.

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'

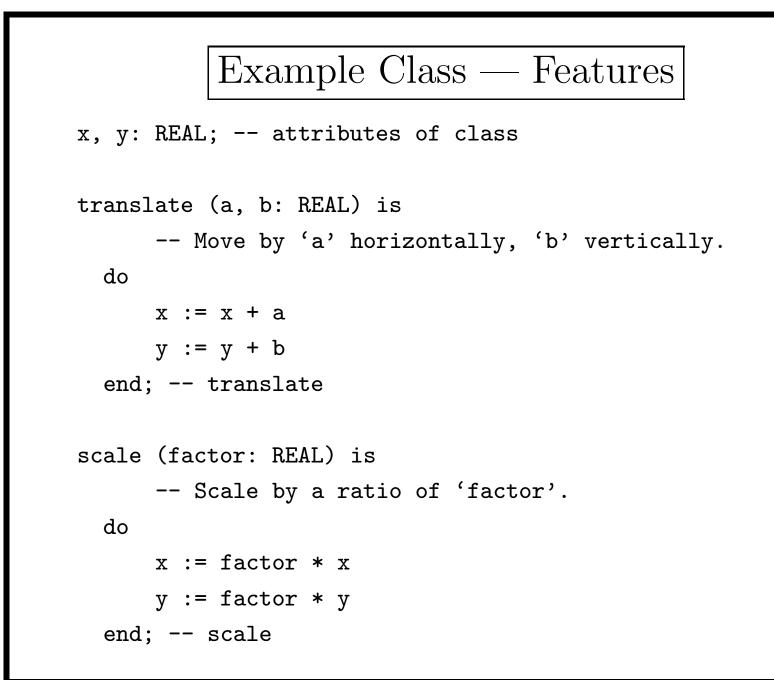
require

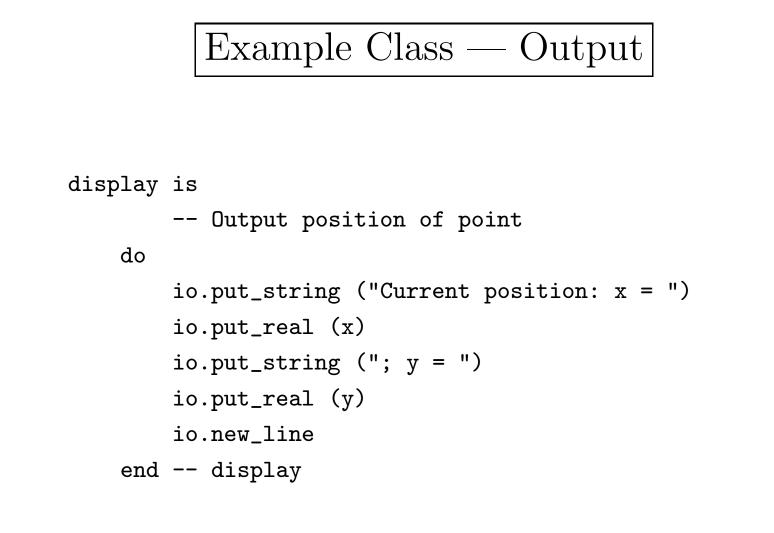
lp /= Void

do

lp.put (Current) -- Current is the object

end; -- Create
```





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 }



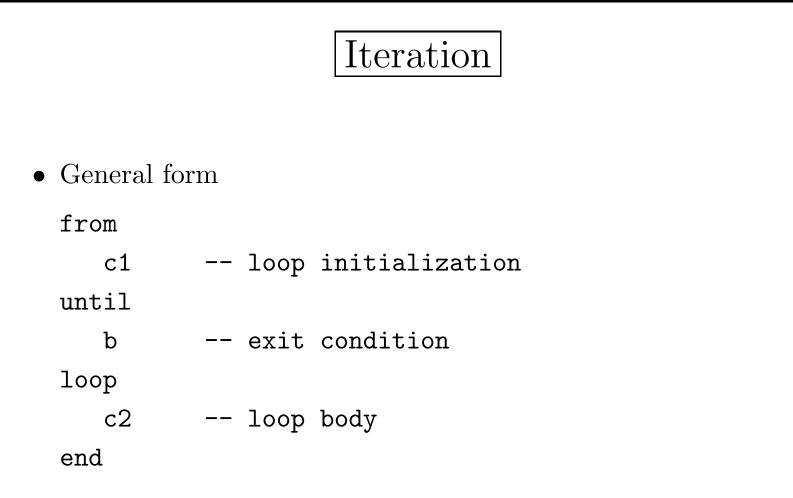
- Unique constants analogous to enumerated type
 - red : INTEGER is unique
 - blue : INTEGER is unique
 - green : INTEGER is unique
 - black : INTEGER is unique
- Compiler determines unique value within enclosing class

Creating Objects

- Following declaration
 - х : Т
- Can bind **x** to newly created object

!!x

- Call creation procedure
 - !!my_point.Create (point_stack)



- Semantics: execute c1, test b, if false then execute c2, testb ...
- Exit when **b** becomes **true**



• Read routines save result in a variable

```
io.read_integer
result := io.last_integer
```

• Look at input_stream.e in SmallEiffel lib_std directory

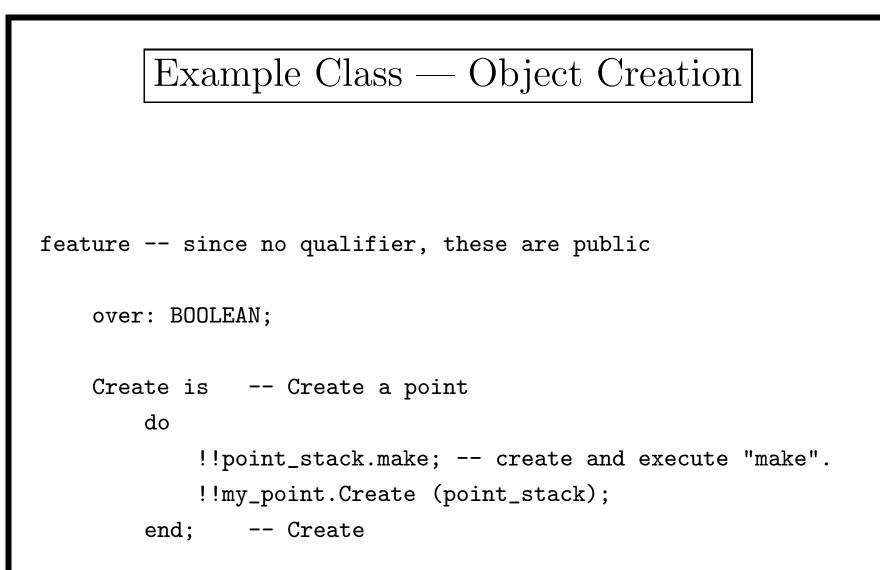
Multi-Branch Instruction

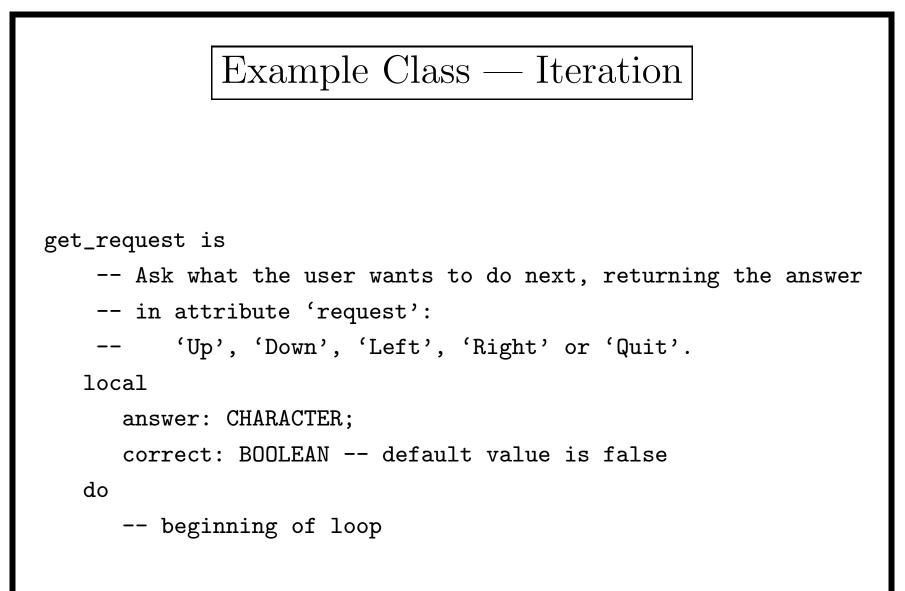
 $\bullet\,$ Select action based on value of <code>INTEGER</code> or <code>CHARACTER</code>

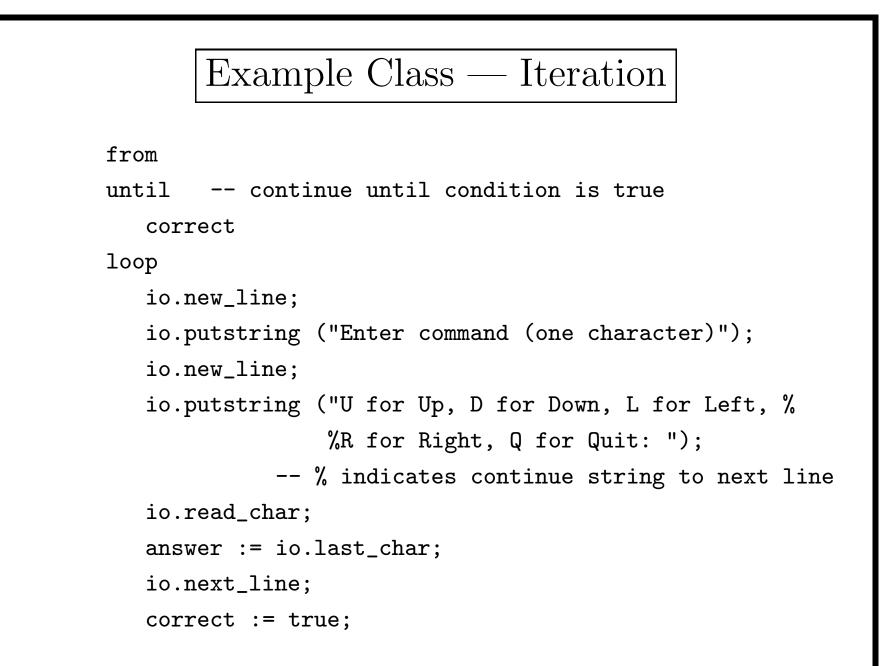
```
inspect input_char
when 'A' .. 'Z' then
   ch_type := Upper_case
when 'a' .. 'z' then
   ch_type := Lower_case
when ',', ';', ':', '.', '?', '!' then
   ch_type := Punctuation
else
   ch_type := Special
end
```

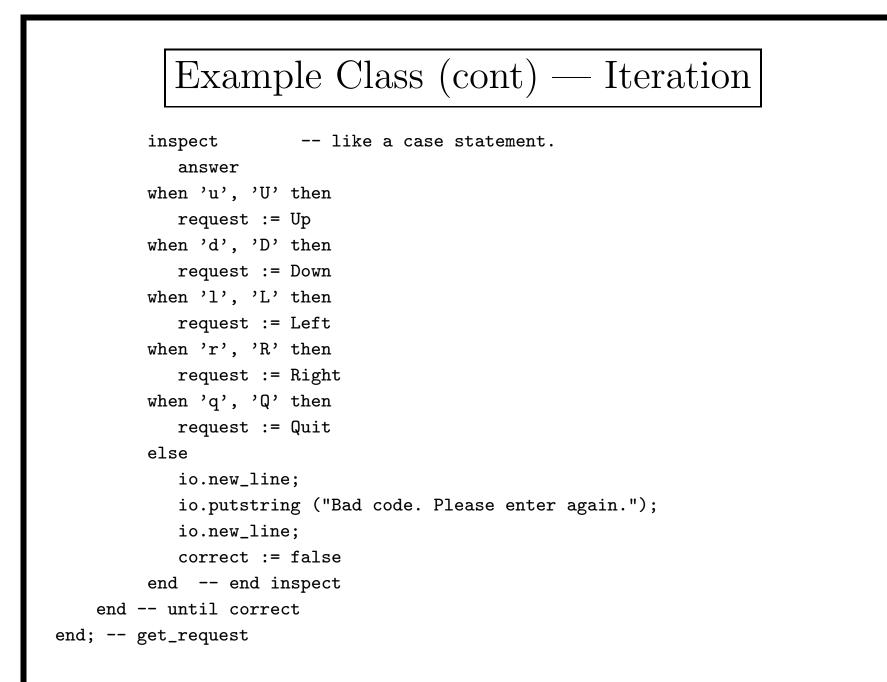
• Sets must be disjoint. Must have **else** if not complete

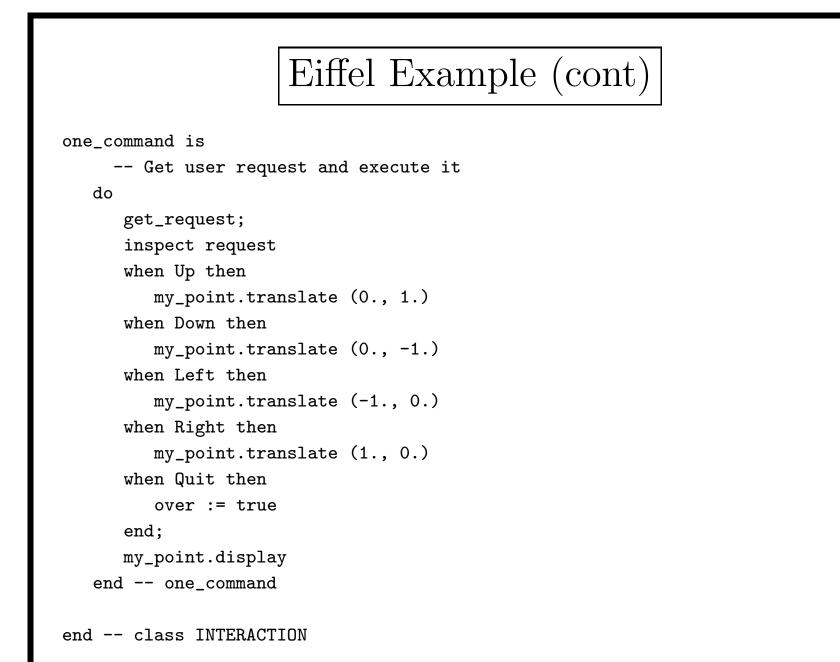
```
Example Class — Access Control
In a separate file: interaction.e
class INTERACTION
creation
   Create
feature {NONE} -- private features
   my_point: POINT;
   request: INTEGER;
   Up, Down, Left, Right, Quit: INTEGER is unique;
   point_stack: LINKED_STACK [POINT]; -- from library
```





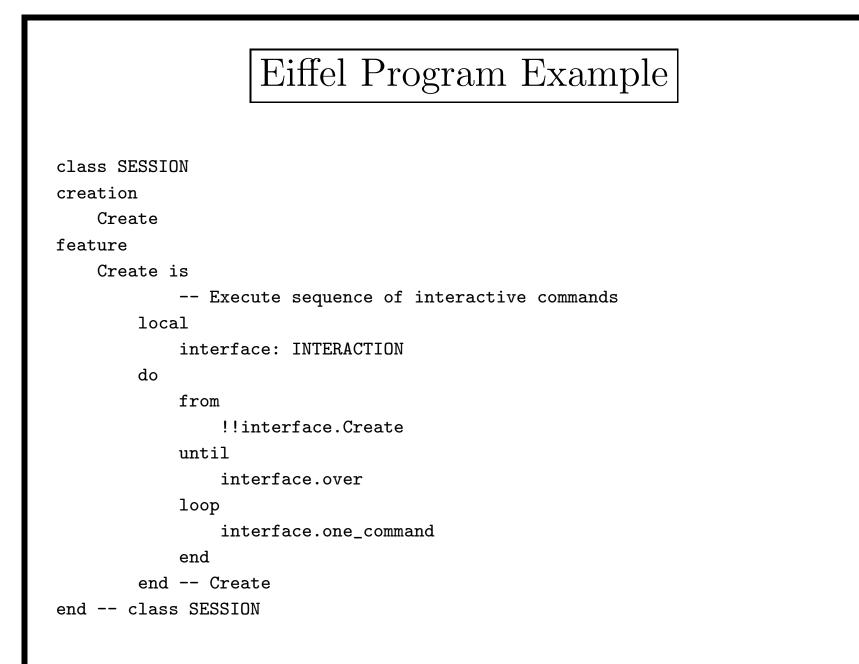






Eiffel Programs

- Program is a collection of classes
- Designate a *root class* and creation procedure to compiler
- Creation procedure serves as "main" procedure



Compilation Control for Example

```
Build is controlled by an "ACE" file such as
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
```

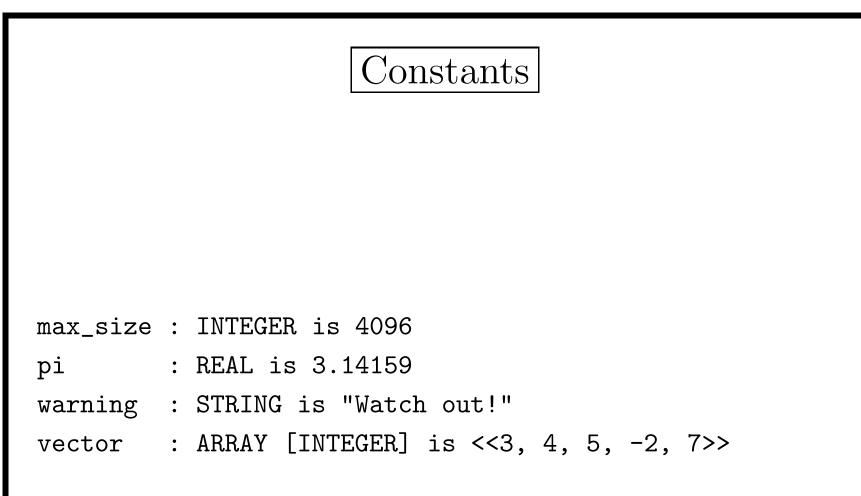


- Primitive types:
 - BOOLEAN
 - CHARACTER
 - INTEGER
 - REAL
 - DOUBLE
- Composite types: ARRAY, STRING



- Default values for primitive types
 - BOOLEAN false
 - CHARACTER '%U' (null character)
 - INTEGER O
 - REAL 0.0
- Default value for reference of other types is void
- Unbind **x** with

x := void



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: +, -, *, //, $\, \hat{}, <, >, <=, >=$
 - REAL: +, -, *, /, ^, <, >, <=, >=
- Integer division //, Modulus $\backslash \backslash$

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 | |
|------------------|--|
| Level | Operator |
| 10 | • |
| 9 | old, not, unary +, unary -, free unary operators |
| 8 | free binary operators |
| 7 | ^ |
| 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)

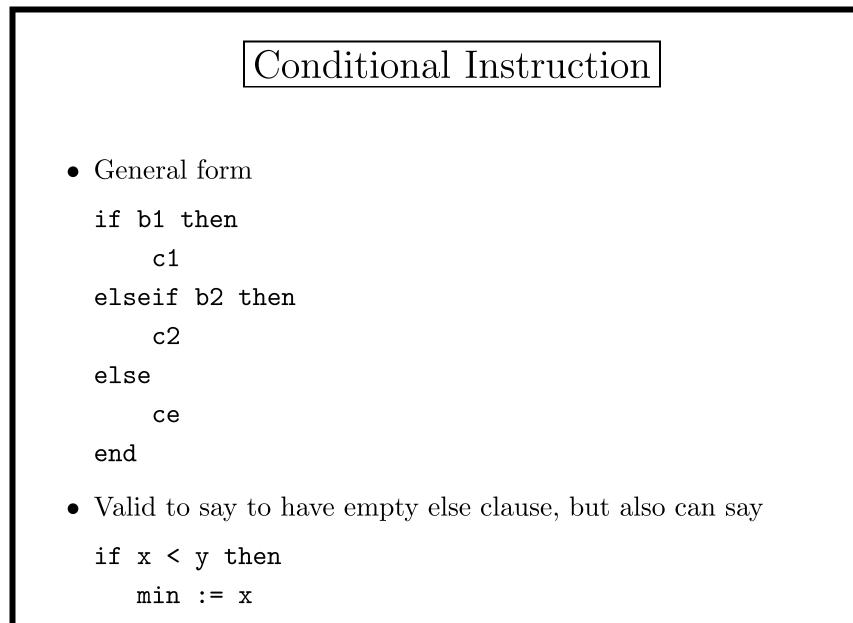
!!b.make(i,a,t)

Expressions

- 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



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



end



• 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 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

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

• Example:

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

```
• C source
```

```
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

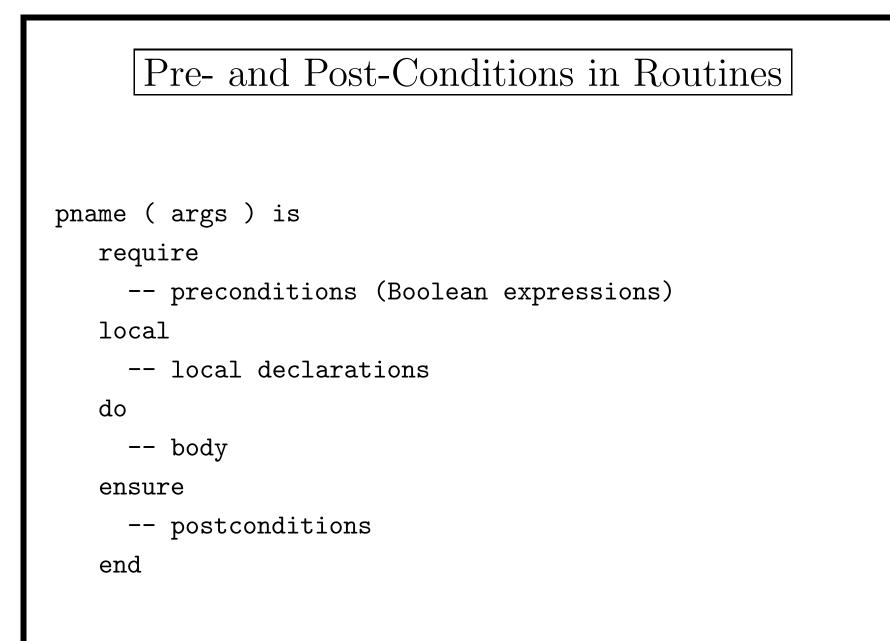
-- code that should only be done one time end

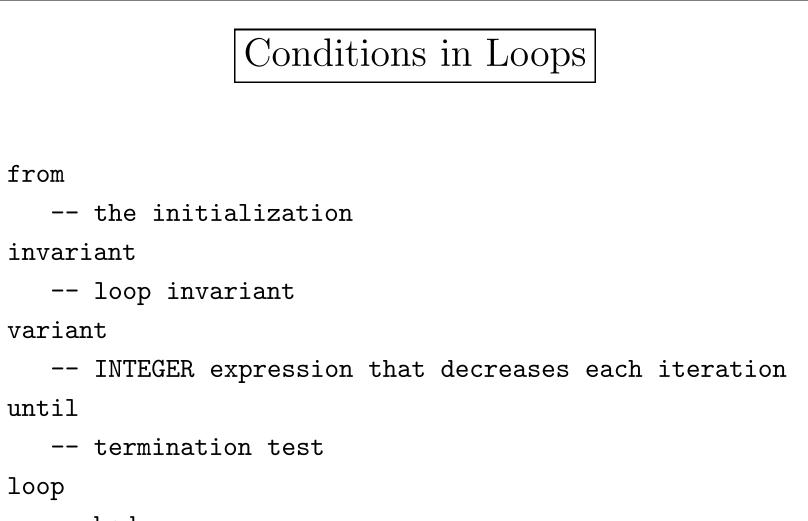
• Once functions always return same value after first call

```
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



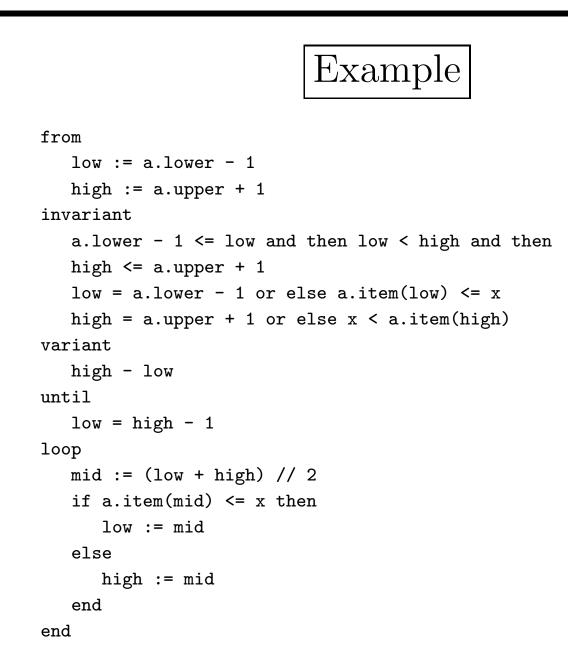


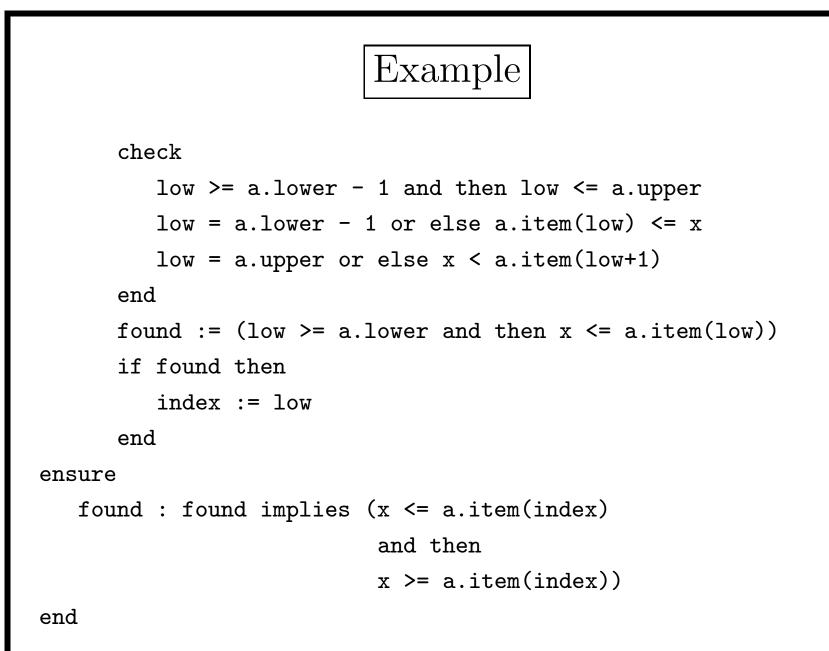
-- body

end



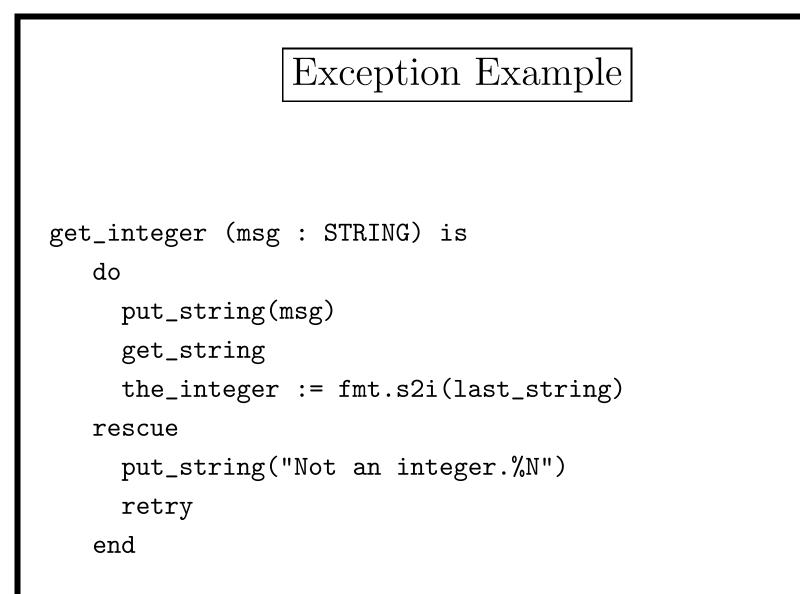
```
binary_search (a : ARRAY [ELEMENT], x : ELEMENT) is
require
    non_trivial : a /= void and then a.count > 0
    is_sorted : -- a is sorted in increasing order
    local
    low, mid, high : INTEGER
    do
```





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





• 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
```

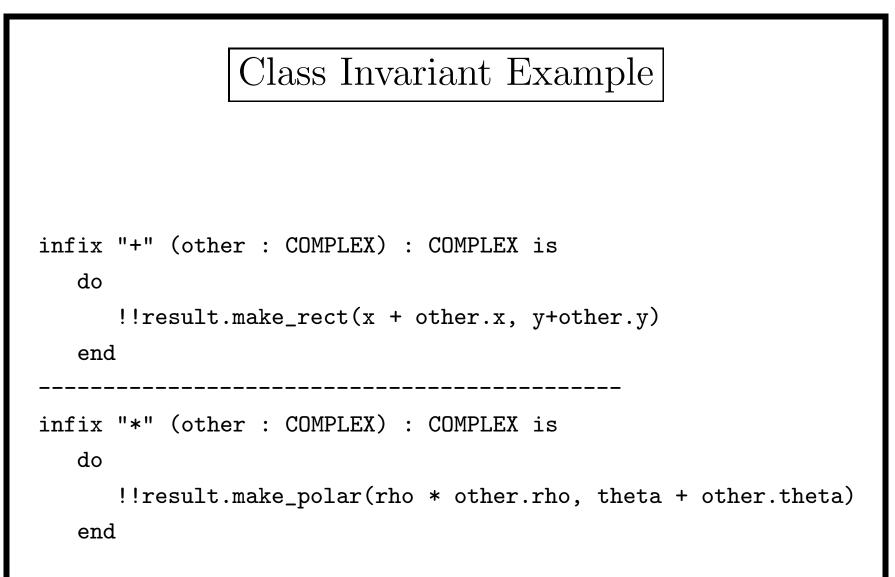
feature

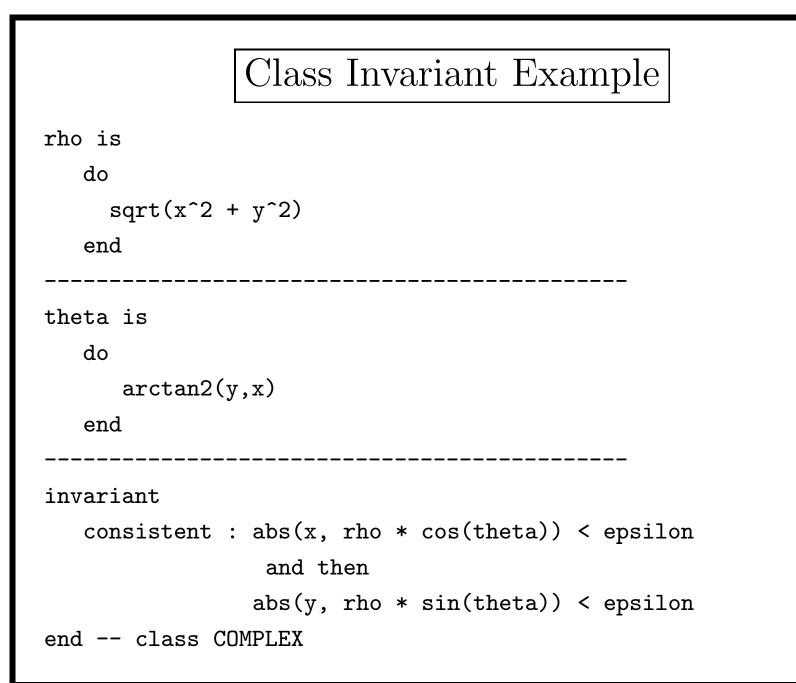
- x : REAL -- real part
- y : REAL -- imaginary part

epsilon : REAL is 1.0E-6

Class Invariant Example

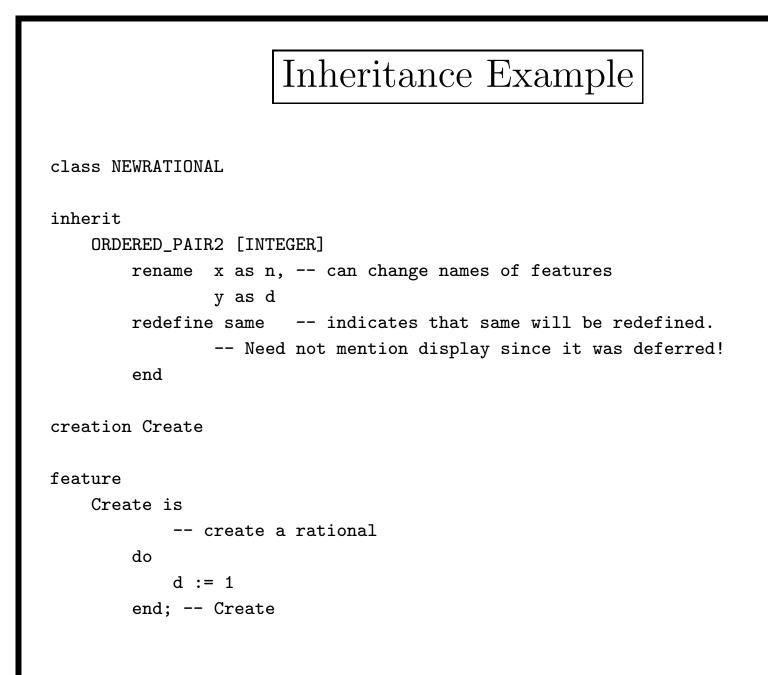
```
make_rect (r, i : REAL) is
  do
     x := r
     y := i
  end
         ______
make_polar (r, t: REAL) is
  do
     x := r * cos(t)
     y := r * sin(t)
  end
```

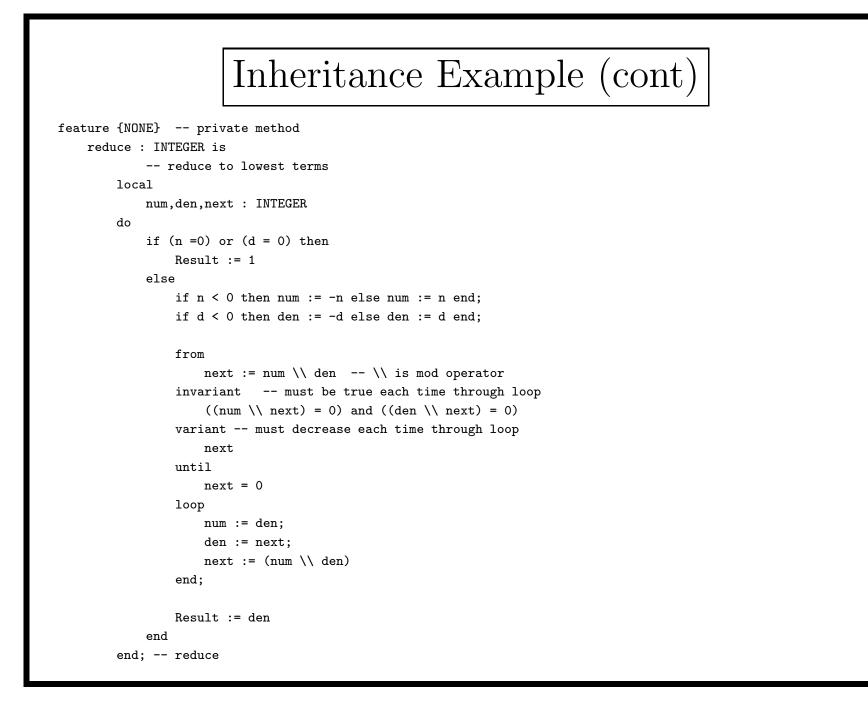


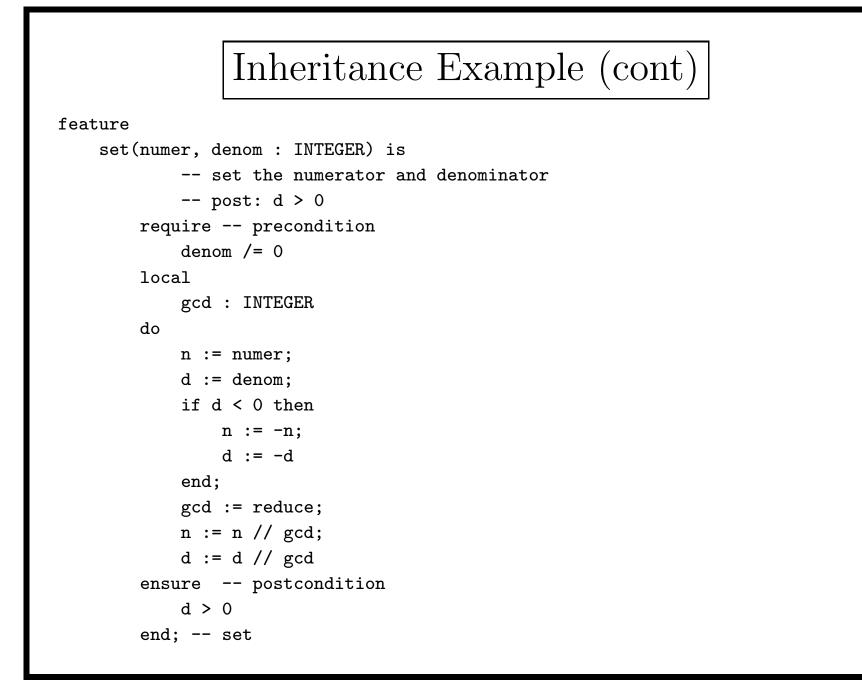


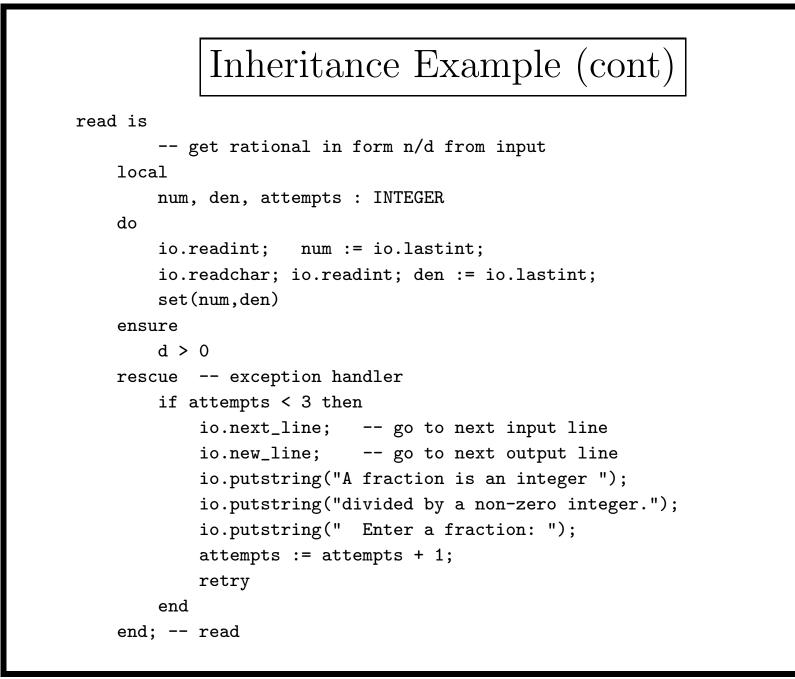
Subclasses and Inheritance

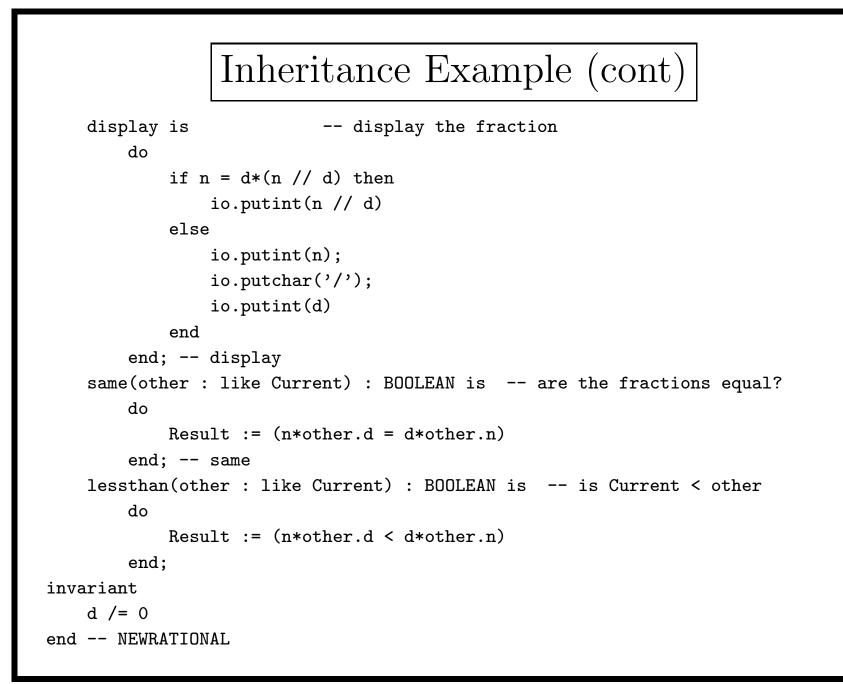
- Declare subclass using inherit
- New class then inherits all features of the old class and can
 - add new features
 - rename inherited features
 - redefine inherited features
- Multiple inheritance list all parent classes





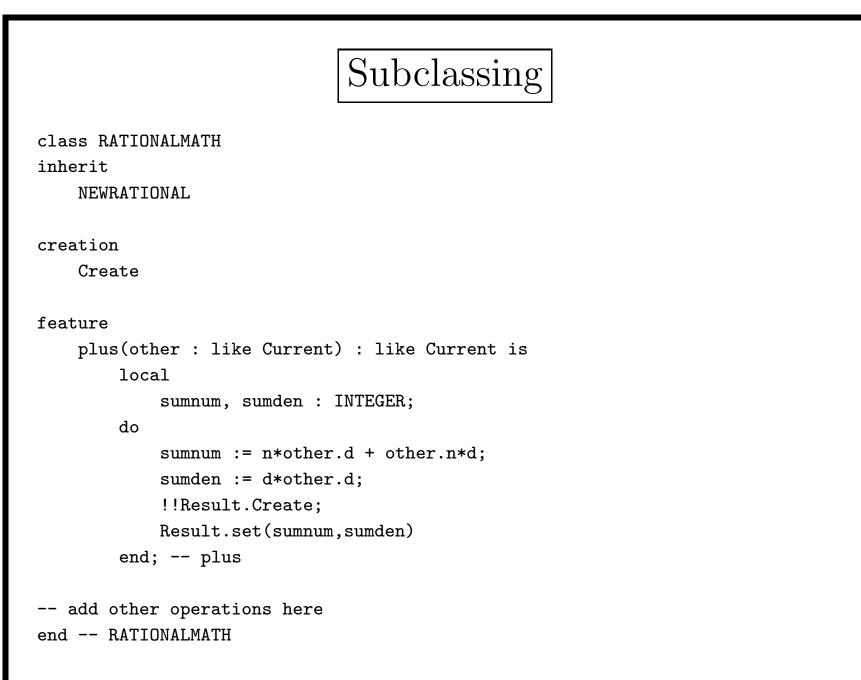


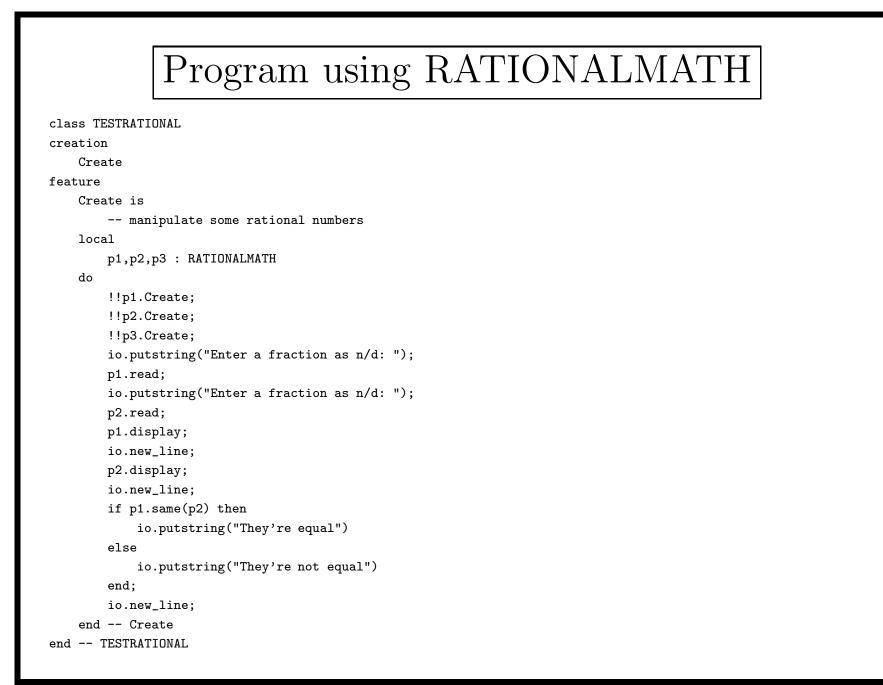




Parameters and Inheritance

- The parameter type like Current in lessthan 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





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

```
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:

```
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 old_m) will be executed.

Redefining

• To get desired behavior inherit twice:

```
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!).

```
(Unconstrained) Genericity
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;</pre>
```

```
infix "<=" (other: like Current): BOOLEAN is
    deferred;</pre>
```

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

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

end interface -- class COMPARABLE

