



Object-Oriented Software Construction

Bertrand Meyer

Lecture 15: Exception handling



The original strategy

```
r (...) is  
  require  
  ...  
  do  
    op1  
    op2  
    ...  
    opi ← Fails, triggering an exception in r  
    ...  
    opn  
  ensure  
  ...  
end
```

(r is recipient of exception).



Exception handling

- The need for exceptions arises when the contract is broken.
- Two concepts:
 - **Failure**: a routine, or other operation, is unable to fulfill its contract.
 - **Exception**: an undesirable event occurs during the execution of a routine — as a result of the **failure** of some operation called by the routine.



Causes of exceptions

- Assertion violation
- Void call ($x.f$ with no object attached to x)
- Operating system signal (arithmetic overflow, no more memory, interrupt ...)
- Program-triggered

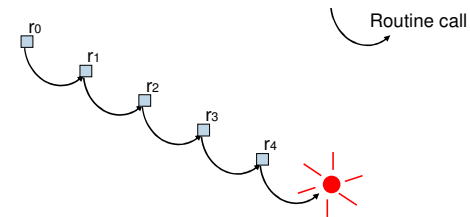
Handling exceptions properly

5

- **Safe exception handling principle:**
 - There are only two acceptable ways to react for the recipient of an exception:
 - Concede failure, and trigger an exception in the caller (**Organized Panic**).
 - Try again, using a different strategy (or repeating the same strategy) (**Retrying**).

The call chain

7



How not to do it

6

(From an Ada textbook)

```
sqrt (x: REAL) return REAL is
begin
  if x < 0.0 then
    raise Negative;
  else
    normal_square_root_computation;
  end
exception
  when Negative =>
    put ("Negative argument");
    return;
  when others => ...
end; -- sqrt
```

Exception mechanism

8

- Two constructs:
 - A routine may contain a **rescue** clause.
 - A rescue clause may contain a **retry** instruction.
- A **rescue** clause that does not execute a **retry** leads to failure of the routine (this is the organized panic case).

Transmitting over an unreliable line (1) 9

```

Max_attempts: INTEGER is 100
attempt_transmission (message: STRING) is
  -- Transmit message in at most
  -- Max_attempts attempts.
  local
    failures: INTEGER
  do
    unsafe_transmit (message)
  rescue
    failures := failures + 1
    if failures < Max_attempts then
      retry
    end
  end
end
  
```

If no exception clause (1) 11

- Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

```

f (...) is
do
...
end
  
```

is an abbreviation for

```

f (...) is
do
...
rescue
-- Nothing here
end
  
```

- (This is a provisional rule; see next.)

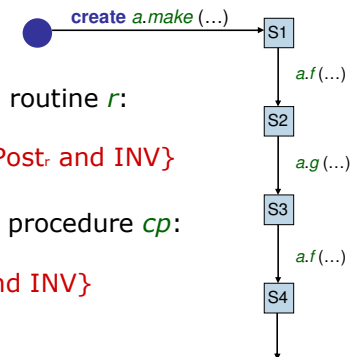
Transmitting over an unreliable line (2) 10

```

Max_attempts: INTEGER is 100
failed: BOOLEAN
attempt_transmission (message: STRING) is
  -- Try to transmit message;
  -- if impossible in at most Max_attempts
  -- attempts, set failed to true.
  local
    failures: INTEGER
  do
    if failures < Max_attempts then
      unsafe_transmit (message)
    else
      failed := True
    end
  rescue
    failures := failures + 1
  retry
end
  
```

The correctness of a class 12

- (1-n) For every exported routine r :
 $\{INV \text{ and } Pre_r\} \text{ do } \{Post_r \text{ and } INV\}$
- (1-m) For every creation procedure cp :
 $\{Pre_{cp}\} \text{ do}_{cp} \{Post_{cp} \text{ and } INV\}$



Exception correctness: A quiz

13

- For the normal body:

`{INV and Pre.} do. {Post. and INV}`

- For the exception clause:

`{ ??? } rescue. { ??? }`

If no exception clause (2)

15

- Absence of a rescue clause is equivalent to a default rescue clause:

```
f (...) is
do
end ...
```

is an abbreviation for

```
f (...) is
do
...
rescue
  default_rescue
end
```

- The task of `default_rescue` is to restore the invariant.

Quiz answers

14

- For the normal body:

`{INV and Pre.} do. {Post. and INV}`

- For the exception clause:

`{True} rescue. {INV}`

For finer-grain exception handling

16

- Use class `EXCEPTIONS` from the Kernel Library.
- Some features:
 - `exception` (code of last exception that was triggered).
 - `assertion_violation`, etc.
 - `raise ("exception_name")`



End of lecture 15