-5-

Exception handling
What is an exception?

“An abnormal event”

Not a very precise definition

Informally: something that you don’t want to happen
Exception vocabulary

- “Raise”, “trigger” or “throw” an exception
- “Handle” or “catch” an exception
How to use exceptions?

Two opposite styles:

- Exceptions as a control structure: Use an exception to handle all cases other than the most favorable ones (e.g. a key not found in a hash table triggers an exception)

- Exceptions as a technique of last resort
How not to do it

(From an Ada textbook)

```ada
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
```
Dealing with erroneous cases

Calling \( a \cdot f(y) \) with

\[
f(x : T) \\
  \text{require} \\
  x \cdot \text{pre} \\
  \text{do} \\
  \quad \ldots \\
  \quad \text{ensure} \\
  \quad \text{Result} \cdot \text{pre} \\
\text{end}
\]

Normal way (\textit{a priori scheme}) is either:

1. \textbf{if} \( y \cdot \text{pre} \) \textbf{then} \( a \cdot f(y) \) \textbf{else} \ldots \text{Error case} \ldots \textbf{end}

2. \textbf{ensure pre ; a.f(y)}
A posteriori scheme

\[a.\text{try\_to\_invert}(b)\]

\[\text{if } a.\text{could\_invert} \text{ then}\]

\[x := a.\text{inverted}\]

\[\text{else}\]

\[\ldots \text{ Error case } \ldots\]

\[\text{end}\]

.
**C++, Java, C#: raising programmer-defined exception**

**Instruction:**

```java
throw my_exception;
try ...
catch (Exception e) ...
try ...
```

The enclosing routine should be of the form

```java
my_routine (...) throws my_exception {
 ...
 if abnormal_condition
  throw my_exception;
}
```

The calling routine must handle the exception (even if the handling code does nothing).
How not to do it: Java

```java
foo () {
    int b=1;
    while (true) {
        try {
            b++; // This line is redundant
            throw new Exception();
        }
        finally {
            b++; // This line is redundant
            break;
        }
    }
    b++; // This line is redundant
}
```

Does this program compile in C#?
Example: Return and finally

```c
foo () {
    try {
        return 1;
    }
    finally {
        return 2;
    }
}
```

Return 2
Example: Return and finally

```c
foo () {
    b = 0;
    try {
        b++; 
        return b;
    }
    finally {
        b++; 
    }
}
```

`b = 2`
Checked vs unchecked exceptions

Checked: raised by program, caller must handle

Unchecked: usually raised by external sources, don’t have to be handled
Exception handling: another approach

Introduce notion of contract

The need for exceptions arises when a contract is broken by either of its parties (client, supplier)

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.
The original strategy

\[ r(\ldots) \]

require
\[ \ldots \]
do
\[ \text{op}_1 \]
\[ \text{op}_2 \]
\[ \ldots \]
\[ \text{op}_i \]
\[ \ldots \]
\[ \text{op}_n \]
ensure
\[ \ldots \]
end
Not going according to plan

\[
\begin{align*}
&\text{Fails, triggering an exception in } r \\
&\text{(} r \text{ is } \textit{recipient} \text{ of exception).}
\end{align*}
\]
Causes of exceptions in O-O programming

Four major kinds:

- Operating system signal: arithmetic overflow, no more memory, interrupt ...
- Assertion violation (if contracts are being monitored)
- Void call \((x.f\text{ with no object attached to } x)\)
- Programmer-triggered

Not any more in Eiffel & Spec#
Handling exceptions properly

Exception handling principle

There are only two acceptable ways to react for the recipient of an exception:

- **Failure**: Concede impossibility of fulfilling contract, and trigger an exception in the caller (also called Organized Panic).
- **Retry**: Try again, using a different strategy (or repeating the same one)

(Rare third case: false alarm)
The call chain

Routine call
Exception mechanism

Two constructs:
- A routine may contain a\textit{ rescue} clause.
- A rescue clause may contain a\textit{ retry} instruction.

A \textit{rescue} clause that does not execute a \textit{retry} leads to failure of the routine (this is the organized panic case).
Exception mechanism (2)

\[ f(...) \]
\[
\begin{align*}
\text{require} & \quad \textit{precondition} \\
\text{local} & \quad ... \textit{local entity declarations}... \\
\text{do} & \quad \textit{body} \\
\text{ensure} & \quad \textit{postcondition} \\
\text{rescue} & \quad \textit{rescue} \_\textit{clause} \\
\text{end}
\end{align*}
\]
If no exception clause (1)

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

\[
  f(...) \\
  \quad \text{do} \\
  \quad \text{...} \\
  \quad \text{end}
\]

is an abbreviation for

\[
  f(...) \\
  \quad \text{do} \\
  \quad \text{...} \\
  \quad \text{rescue} \\
  \quad \text{--- Nothing here} \\
  \quad \text{end}
\]

(This is a provisional rule; see next.)
Transmitting over an unreliable line (1)

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
Transmitting over an unreliable line (2)

\[ \text{Max_attempts : INTEGER is 100} \]

\[ \text{failed : BOOLEAN} \]

\[ \text{attempt_transmission (message: STRING) is} \]

\[ -- \text{Try to transmit message;} \]
\[ -- \text{if impossible in at most Max_attempts} \]
\[ -- \text{attempts, set failed to true.} \]

\[ \text{local} \]
\[ \text{failures: INTEGER} \]

\[ \text{do} \]
\[ \text{if failures < Max_attempts then} \]
\[ \text{unsafe_transmit (message)} \]
\[ \text{else} \]
\[ \text{failed := True} \]
\[ \text{end} \]

\[ \text{end rescue} \]

\[ \text{failures := failures + 1} \]

\[ \text{retry} \]

\[ \text{end} \]
Another Ada textbook example

```
procedure attempt is begin
  <<Start>> -- Start is a label
  loop
    begin
      algorithm_1;
      exit; -- Alg. 1 success
    exception
      when others =>
        begin
          algorithm_2;
          exit; -- Alg. 2 success
        exception
          when others =>
            goto Start;
        end
    end loop
end main;
```

In Eiffel

```
attempt
local
  even: BOOLEAN
do
  if even then
    algorithm_2
  else
    algorithm_1
  end
rescue
  even := not even
Retry
end
```
Dealing with arithmetic overflow

quasi_inverse (x: REAL): REAL
    -- 1/x if possible, otherwise 0
    local
division_tried: BOOLEAN
    do
        if not division_tried then
            Result := 1/x
        end
    rescue
        division_tried := True
        Retry
    end
Transmitting over an unreliable line (3)

Max_attempts: INTEGER is 100
failed : BOOLEAN

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
    failed := true ...
  end
Pseudo code

from unsafe_transmit (message) until not exceptions loop
    failures := failures + 1
    if failures < Max_attempts then
        continue l
    end
    failed := true ... raise exception
    l: unsafe_transmit (message)
end

Retry allows to transfer control flow to the beginning of a routine without executing the rest of the rescue block
ETL3: Exception mechanism

Two constructs:
- A routine may contain a rescue clause
- A rescue clause may set the Retry boolean variable

A rescue clause that ends with Retry not set leads to failure of the routine ("organized panic").
ETL3: Transmitting over an unreliable line (1)

Max_tries: INTEGER = 100

attempt_transmission_1 (message: STRING )

-- Transmit message in at most Max_tries attempts.

local
  failures: INTEGER

do
  unsafe_transmit (message)
rescue
  failures := failures + 1
  Retry := (failures < Max_tries)
end
ETL3: Transmitting over an unreliable line (2)

\[ \text{Max}_\text{tries}: \text{INTEGER} = 100 \]

\[ \text{attempt}_\text{transmission}_2 (\text{message}: \text{STRING}) \]

-- Try to transmit message in at most Max_tries attempts; if impossible, set could_not to True.

local

\[ \text{failures}: \text{INTEGER} \]

do

if failures < Max_tries then

\[ \text{unsafe}_\text{transmit} (\text{message}) \]

else

\[ \text{could}_\text{not} := \text{True} \]

end

rescue

\[ \text{Retry} := \text{True} \]

end
quasi_inverse (x: REAL): REAL
    -- 1/x if possible, otherwise 0
    local
division_tried: BOOLEAN
    do
        if not division_tried then
            Result := 1/x
        end
    rescue
        division_tried := True
        Retry := True
    end
The correctness of a class

For every exported routine $r$:

$$\{\text{INV and } \text{Pre}_r\} \text{ do } r \{\text{INV and } \text{Post}_r\}$$

For every creation procedure $cp$:

$$\{\text{Pre}_{cp}\} \text{ do } cp \{\text{INV and } \text{Post}_{cp}\}$$
Exception correctness

For the normal body:

\[ \{ \text{INV and Pre}_r \} \; \text{do}_r \; \{ \text{INV and Post}_r \} \]

For the exception clause:

\[ \{ ??? \} \; \text{rescue}_r \; \{ ??? \} \]
Exception correctness

For the normal body:

\[
\{ \text{INV and Pre}_r \} \text{ do}_r \{ \text{INV and Post}_r \}
\]

For the exception clause:

\[
\{ \text{True} \} \text{ rescue}_r \{ \text{INV} \}
\]
Absence of a rescue clause is equivalent to a default rescue clause:

```plaintext
f(...) do ...
end
```

is an abbreviation for

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

The task of `default_rescue` is to restore the invariant.
Exceptions are objects
Library mechanisms to raise exceptions, distinguish between exception types, and access details of an exception
The full rule

\[ P \Rightarrow P' \]

\[ \{ \text{INV} \land P \} \quad b \quad \{ \text{INV} \land Q \mid Q' \} \]

\[ \{ Q' \} \quad r \quad \{ \text{INV} \land (\text{Retry} \Rightarrow \overline{P}) \land (\neg \text{Retry} \Rightarrow R) \mid \text{INV} \land R \} \]

\[ \{ \text{INV} \land P \} \quad \text{do } b \text{ rescue } r \text{ end } \quad \{ \text{INV} \land Q \mid \text{INV} \land R \} \]

Normal postcondition

Error postcondition

“Retry invariant”
Summary and conclusion

Exceptions as a control structure (internally triggered):
Benefits are dubious at best

An exception mechanism is needed for unexpected external events

Need precise methodology; must define what is “normal” and “abnormal”. Key notion is “contract”.

Next challenge is concurrency & distribution
Further reading


Chapter 12: When the contract is broken: exception handling

ECMA-367 Standard, Section 8.26