Programming in the large

Bertrand Meyer
Lecture 24: Exception handling in object-oriented programming
Causes of exceptions

Void call ($x.f$ with no object attached to $x$)

Operating system signal: arithmetic overflow, no more memory, interrupt ...

Assertion violation (if contracts are being monitored)
Exceptions

An exception is an “abnormal case” occurring program execution, causing a disruption of the default flow of control.
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
Exception vocabulary

- "Raise", "trigger" or "throw" an exception
- "Handle" or "catch" an exception
Java exceptions

Exceptions are objects, descendants of Throwable:
Java: raising an exception

Instruction:

```java
throw my_exception
```

The enclosing routine should be of the form

```java
my Routine (...) throws my_exception {
    ...
    if abnormal_condition
        throw my_exception;
}
```
Java: handling an exception

```java
try {
    instruction_1;
    instruction_2;
    ...
    instruction_n;
}
catch (Expected_exception_type e) {
    handling_code
}

(Possible "finally" clause to complete both cases)
```
Exception handling

The need for exceptions arises when a 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.
The original strategy

\[ r(...) \text{ is} \]
\[
\text{require} \]
\[
\ldots
\]
\[
\text{do}
\]
\[
\text{...}
\]
\[
\text{op}_1
\]
\[
\text{op}_2
\]
\[
\text{...}
\]
\[
\text{...}
\]
\[
\text{op}_i
\]
\[
\text{...}
\]
\[
\text{ensure}
\]
\[
\ldots
\]
\[
\text{end}
\]

Fails, triggering an exception in \( r \) (\( r \) is \textit{recipient} of exception).
Handling exceptions properly

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)

(Rare third case: false alarm)
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
```
The call chain
Exception mechanism

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)

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)

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

\[
\text{f (...) is}
\begin{array}{l}
\text{do} \\
\text{...}
\end{array}
\text{end}
\]

is an abbreviation for

\[
\text{f (...) is}
\begin{array}{l}
\text{do} \\
\text{...} \\
\text{rescue} \\
\text{-- Nothing here}
\end{array}
\text{end}
\]

(This is a provisional rule; see next.)
The correctness of a class

(1\text{-}n) For every exported routine $r$:

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

(1\text{-}m) For every creation procedure $cp$:

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

For the normal body:

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

For the exception clause:

\{ ??? \} \text{ rescue } \{ ??? \}
Exception correctness

For the normal body:

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

For the exception clause:

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

\[ f(...) \text{ is} \]
\[ \begin{array}{c}
  \text{do} \\
  \text{...}
\end{array} \]
\[ \text{end} \]

is an abbreviation for

\[ f(...) \text{ is} \]
\[ \begin{array}{c}
  \text{do} \\
  \text{...}
\end{array} \]
\[ \text{rescue} \]
\[ \text{default}\_\text{rescue} \]
\[ \text{end} \]

The task of \textit{default_rescue} is to restore the invariant.
For finer-grain exception handling

Use class EXCEPTIONS from the Kernel Library.

Some features:

- `exception` (code of last exception that was triggered).
- `assertion_violation`, etc.
- `raise` ("exception_name")