Lecture 6: 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
Java exceptions

Exceptions are objects, descendants of Throwable:
Java: raising a programmer-defined 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;
}
```

The calling routine must handle the exception (even if the handling code does nothing).

To handle an exception: `try ... catch ...`
Checked vs unchecked exceptions

Checked: raised by program, caller must handle

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

A more rigorous basis:

- 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

\( r (...) \) is

require

... 

do

\begin{align*}
&op_1 \\
&op_2 \\
&... \\
&op_i \\
&... \\
&op_n
\end{align*}

ensure

... 

end
Not going according to plan

\[ r(...), is \]
require

\[ \text{...} \]
do

\[ \text{op}_1 \]
\[ \text{op}_2 \]
\[ \text{...} \]
\[ \text{...} \]
\[ \text{op}_i \]
ensure

\[ \text{...} \]
end

Fails, triggering an exception in \( r \) (\( r \) is \textit{recipient} of exception).
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 with no object attached to x)

In Eiffel & Spec#, will go away
Total functions

Let $A$ and $B$ be two sets

- A total function from $A$ to $B$ is a mechanism associating a member of $B$ with every member of $A$

- If $f$ is such a total function and $a \in A$, then the associated member of $B$ is written $f(a)$

- The set of all such members of $B$ is written range $f$

The set of total functions from $A$ to $B$ is written $A \rightarrow B$
Relations

A relation $r$ from $A$ to $B$ is a total function in

$$P(A) \rightarrow P(B)$$

such that $r(\emptyset) = \emptyset$ and for any subsets $X$ and $Y$ of $A$,

$$r(X \cup Y) = r(X) \cup r(Y)$$

The set of relations from $A$ to $B$ is also written

$$A \leftrightarrow B$$

For $r \in A \leftrightarrow B$

$X \subseteq A$

the set $r(X)$ is called the \textbf{image} of $X$ by $r$
A function from $A$ to $B$ is a total function from $X$ to $B$, for some $X \subseteq A$.

The set of functions from $A$ to $B$ is written $A \rightarrow B$.

The **domain** of a function $f \in A \rightarrow B$, written **domain** $f$, is the largest subset $X \subseteq A$ such that $f \in X \rightarrow B$. 
Total and partial functions

Theorem 1:

For any $f : A \to B$, there exists $X \subseteq A$ such that $f \in X \to B$

Theorem 2:

For any $f : A \to B$, for any $X \supseteq A$,

$$f \in X \leftrightarrow B$$
Convention:

For $f \in A \rightarrow B$ and $a \in A$, we may write $f(a)$ (as for a total function) if we prove that $a \in \text{domain } f$.
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)

\[
\text{sqrt}(x: \text{REAL}) \text{ return REAL is}
\]

\[
\begin{align*}
\text{begin} \\
\text{if } x < 0.0 \text{ then} \\
\text{raise Negative;} \\
\text{else} \\
\text{normal\_square\_root\_computation;} \\
\text{end} \\
\text{exception} \\
\text{when Negative =>} \\
\text{put("Negative argument");} \\
\text{return;} \\
\text{when others => ...} \\
\text{end; -- sqrt}
\end{align*}
\]
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)

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

\[\text{attempt_transmission (message: STRING) is}\]
\[\quad -- \text{Try to transmit message;}\]
\[\quad -- \text{if impossible in at most Max_attempts}\]
\[\quad -- \text{attempts, set failed to true.}\]

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

\[
\begin{aligned}
\text{rescue} \\
\quad \text{failures := failures + 1} \\
\text{retry}
\end{aligned}
\]

end
Another Ada textbook example

```ada
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
end main;
```

In Eiffel

```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

\[
\text{quasi\_inverse}\ (x: \text{REAL}): \text{REAL} \\
\quad \text{-- } 1/x \text{ if possible, otherwise 0}
\]

local
\[
\text{division\_tried: BOOLEAN}
\]
do
\[
\quad \text{if not } \text{division\_tried} \text{ then}
\quad \text{Result} := 1/x
\]
end
rescue
\[
\quad \text{division\_tried} := \text{True}
\]
retry
end
If no exception clause (1)

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

\[
\begin{align*}
  f(...) & \text{ is} \\
  & \text{ do} \\
  & \text{ end} \\
\end{align*}
\]

is an abbreviation for

\[
\begin{align*}
  f(...) & \text{ is} \\
  & \text{ do} \\
  & \text{ end} \\
  & \text{ rescue} \\
  & \text{ -- Nothing here} \\
\end{align*}
\]

(This is a provisional rule; see next.)
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 $cp$:

$$
\{\text{Pre}_{cp}\} \; \text{do}_cp \; \{\text{INV and } \text{Post}_{cp}\}
$$
Bank accounts

\[ balance := \text{deposits.total} - \text{withdrawals.total} \]
Exception correctness

For the normal body:

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

For the exception clause:

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

For the normal body:

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

For the exception clause:

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

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

is an abbreviation for

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

The task of `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)
- `is_assertion_violation`, etc.
- `raise` ("exception_name")

Inheritance from class *EXCEPTIONS* is replaced in ISO/ECMA Eiffel by the use of exception objects (class *EXCEPTION*).
Dealing with erroneous cases

Calling
  \(a \cdot f(y)\)
with
  \(f(x: T)\)
require
  \(x.\text{property}\)
do
  ...
ensure
  \(\text{Result.} other\_\text{property}\)
end

Normal way (a priori scheme) is either:

1. if \(y.\text{property}\) then \(a \cdot f(y)\) else ... end

2. ensure\_property: \(a \cdot f(y)\)
A posteriori scheme (from OOSC)

\[ a \cdot \text{invert} \ (b) \]

\[ \text{if } a \cdot \text{inverted} \text{ then } \]
\[ \quad x := a \cdot \text{inverse} \]

\[ \text{else} \]
\[ \quad \text{... Appropriate error action ...} \]

\[ \text{end} \]
Using agents (from *Standard Eiffel*)

Scheme 1:

```
action1
if ok1 then
  action2
    if ok2 then
      action3
        ... More processing, more nesting ...
    end
  end
end
```

Scheme 2:

```
controlled_execute ([
  agent action1,
  agent action2 (...),
  agent action3 (...)
])
if glitch then
  warning (glitch_message)
end
```
Another challenge today

Exceptions in a concurrent world

Another talk...
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
Complementary material

**OOSC2:**
- Chapter 11: Design by Contract