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:
    throw my_exception;

The enclosing routine should be of the form
    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

```plaintext
r(...) is 
require ...
do ...
  op1 ...
  op2 ...
  opn ...
ensure ...
end ...
```
Not going according to plan

```plaintext
r(...) is require do op1 op2 ... end ensure op Fails, triggering an exception in r (r is 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 $\mathcal{P}(A) \to \mathcal{P}(B)$ 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 image of $X$ by $r$.

Functions (possibly partial)

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 $A \to B$.

The domain of a function $f \in A \to B$, written $\text{domain } f$, is the largest subset $X \subseteq A$ such that $f \in X \to B$.

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 $X \supseteq A$, $f \in X \leftrightarrow B$. 
Using partial functions

**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)

```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

Routine call

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
```

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
end
end main;
```

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
```
If no exception clause (1)

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

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

is an abbreviation for

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

\[ \text{Nothing here} \]

(This is a provisional rule; see next.)

The correctness of a class

For every exported routine \( r \):

\[ \{ \text{INV and Prer} \} \text{ do }_{r} \{ \text{INV and Post}_{r} \} \]

For every creation procedure \( cp \):

\[ \{ \text{Pre}_{cp} \} \text{ do }_{cp} \{ \text{INV and Post}_{cp} \} \]

Bank accounts

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

\[ \text{(A2)} \]

\[ \text{deposits} \quad \text{withdrawals} \quad \text{balance} \]
Exception correctness

For the normal body:

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

For the exception clause:

\[
(\text{???) rescue}_r (\text{???)}
\]

---

Exception correctness

For the normal body:

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

For the exception clause:

\[
(\text{True}) \text{ rescue}_r (INV)
\]

---

If no exception clause (2)

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

\[
(... \text{ is do} \text{ ... end})
\]

is an abbreviation for

\[
(... \text{ is do rescue default rescue end})
\]

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.f(y)
with
  f(x : T)
  require
    x.property
  do
    …
  ensure
    Result, other_property
end
```

Normal way (a priori scheme) is either:
1. if y.property then a.f(y) else … end
2. `ensure_property; a.f(y)`

A posteriori scheme (from OOSC)

```
a.invert(b)
if a.inverted then
  x := a.inverse
else
  … Appropriate error action …
end
```
Using agents (from Standard Eiffel)

Scheme 1:
action1
if ok1 then
  action2
if ok2 then
  action3
  ... More processing, more nesting ...
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...

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
Complementary material

OOSC2:
> Chapter 11: Design by Contract