

## Exceptions

An exception is any unusual event, erroneous or not, that is detectable by either hardware or software and may require special processing.

In the beginning:

- hardware detection of erroneous operations (divide by zero, overflow, execution of a bad instruction)
- I/O errors in Fortran

```
READ (UNIT=5, FORMAT=1000, ERR=100, END=999) WEIGHT
```

Consistent with Fortran's level of abstraction for control structures: GOTOs. Fortran allows general use of LABEL parameters.

- Return codes, as is common for C library functions, and signals (from Unix/C)

As programming languages evolved, it became clear that these mechanisms weren't adequate to match with more structured control flow mechanisms and scope definition features than those in Fortran.

Note: nothing explicitly for exception handling was included in Algol 60 and 68, though the latter allows passing blocks of code as parameters. Recall this when we consider Clu later.

## **Design Issues**

1. How or where are exception handlers specified and what is their scope?
2. How is an exception occurrence bound to an exception handler?
3. Where does execution continue, if at all, after an exception handler completes its execution?
4. How are user-defined exceptions declared?
5. Should there be exception handlers for programs that do not provide their own?
6. Can built-in exceptions be raised explicitly?
7. How can exceptions be disabled, if at all?

## Exceptions in PL/I

PL/I pioneered the concept of allowing user programs to be directly involved in handling a full range of exceptions (with a long list of language-defined exceptions, of course). PL/I also introduced the concept of user-defined exceptions, which allows programmers to create software detected exceptions.

- Exception handlers in PL/I are executable code blocks that can appear anywhere an executable statement can appear (ON condition ...).
- Binding stays in effect until either a new ON statement for the same condition is executed or the block in which it occurs is exited.
- For some built-in exceptions, execution returns to the point where the exception was raised (CONVERSION), while for others, the program terminates (OVERFLOW). User-defined handlers can make control go anywhere, but they have no way of knowing where the exception was raised.

## Exceptions in Clu

A major step forward in the formulation of exceptions in structured languages occurred with the design of Clu, an important experimental language of the mid-1970s built around data abstraction concepts.

Procedure definition syntax in Clu:

```
procedure_name = proc (formal parameters)
                  signals (exception_1 (parameters),
                           . . .
                           exception_n (parameters))
  -- procedure body --
end procedure_name
```

Key points:

- Exceptions are part of interfaces.
- Exceptions can have parameters, which are sent to handlers.

Exception handlers can be attached to any statement in a Clu procedure; they handle only exceptions raised by subprograms called by that statement. The general syntactic form is:

```
statement
  except
    exception_name_1 (parameters): statement_1
    ...
    exception_name_n (parameters): statement_n
  end
```

An exception must be handled somewhere in the procedure that called the one that signals an exception. If no handler exists, the built-in exception failure is signaled.

If a handler is attached to a statement, execution flows to the next statement after the handler executes. If the handler is attached to the body of a procedure, the procedure returns after the handler is done (unless it signals another exception).

## Exceptions in Ada

Exceptions in Ada are much like those in Clu, with a couple of major differences:

- Exceptions can't have parameters.
- Exceptions are declared independently of subprogram interfaces.
- Exceptions can propagate through any number of program calls before being handled (no equivalent of **failure**).

Exceptions in Ada are declared like types or variables:

```
exception_name_list : exception;
```

Handlers are attached to blocks or unit bodies as follows:

```
begin
    -- statements of the block or unit body go here
exception
    when exception_name_list =>
        -- statements of the first handler go here
    when exception_name_list =>
        -- statements of the second handler go here
    . . .
    when others => -- optional
        -- statements of the others handler go here
end
```

In Ada, exceptions are "raised" as opposed to being "signaled" in PL/I and Clu or "thrown" in C++ and Java. Where does "throw" come from? It is from the Lisp world, where throw-catch exception handling was introduced quite a while ago. Note that it is really a lot like the Clu approach, though without the nice interface definition capabilities.

## Exceptions in Java

```
...
try {
    ... some code ..
} catch (X e){
...
} catch (Y e) {
...
} catch (Z e) {
...
}
```

Catch clauses are considered in the order they appear in the code. X, Y and Z in the code above are types derived from Error or Exception

Since exceptions are in a heirarchy, you can catch more specific exceptions first and less specific ones later.

Example:

```
...
try {
    ... some code ...
} catch ( myException e) {
... specific code if they throw myExceptin ...
} catch ( Exception e) {
... general error handling for any exception
}
```

## Exceptions in Eiffel

```
routine is
  require
    precondition
  local
    ... local entity declarations ...
  do
    body
  ensure
    postcondition
  rescue
    rescue_clause -- may include retry
end
```

If any failure occurs while executing the body, the rescue clause may attempt to adjust the state (making sure that the precondition is still true) and cause the body to be retried, or it may just ensure that any class invariants are still satisfied and then propagate the failure. Thus the result must be either a successful execution of the body or a routine failure.

Class EXCEPTIONS provides a set of integer codes corresponding to particular exceptions and queries to do particular handling of specific exceptions.

User-defined ("developer") exceptions are also supported by this class. They are signaled by

```
trigger (code: Integer; message: String)
```

An additional object (of type ANY) can be associated with a developer exception as a "context."