TUTORIAL ON

EXCEPTION HANDLING IN ADA (R)

Tuesday, March 25, 1986

Washington Ada Symposium

Benjamin M. Brosgol

Alsys, Inc.

1432 Main Street

Waltham, Mass. 02154

(617) 890-0030

(R) Ada is a registered trademark of the U.S. Government (AJPO)

OVERVIEW OF PRESENTATION

- Motivation and Basic Principles
- Exception Handlers and How to Find Them
- Programming with Exceptions
- Tasking and Exceptions
- Exceptions and Optimizations
- Conclusions

WHAT IS AN EXCEPTION HANDLING FACILITY

- Control structure to deal with run-time situations that are:
  - Rare
  - Inconvenient to test for at the point of occurrence
  - Not going to require that control return to the point of occurrence
  - Typically error conditions

- An <u>exception</u> is a name for such a situation
  - Predefined: CONSTRAINT_ERROR

    Situation: Array subscript out of bounds
  - User-declared: TABLE_FULL

    Situation: Inserting an element when no more room

- Activities comprise:
  - <u>Raising</u> the exception --

    Indicating that the situation has occurred
  - <u>Handling</u> the exception --

    Executing some actions in response

WHY EXCEPTION HANDLING

- Appropriate to application domain of real-time programming
  - Typical program is infinite loop, interacting with external environment
  - Even rare events will eventually occur
- Promotes program readability and efficiency
  - Program text shows the main processing clearly, not cluttered with checking of status flags
- Helps avoid language complexity
  - No need for "wild goto" or passing of error-handling procedures
- Steelman required it

RAISING AN EXCEPTION IMPLICITLY

An exception may be raised with no explicit indication in the source program

Examples:

```
X := Y+Z;            -- NUMERIC_ERROR if Y+Z overflows

X := Y/Z;            -- NUMERIC_ERROR if Z=0

I: POSITIVE := J;    -- CONSTRAINT_ERROR if J <= 0

X: array (1..10) of INTEGER;
.
.
.
X(J) := 0;           -- CONSTRAINT_ERROR if J not in 1..10

type PTR is access INTEGER;

P: PTR;
.
.
.
P.all := 0;          -- CONSTRAINT_ERROR if P = null
```

There may or may not be run-time code to check for the condition that will raise the exception

- o NUMERIC_ERROR
  - o Trap on overflow
  - o Check a status flag
- o CONSTRAINT_ERROR for access value dereference
  - o Trap on address fault
  - o Check vs. specific null value

Often the compiler can guarantee that there is no possibility for the condition to occur that will raise the exception:

```
I: INTEGER range 0..10 := 5;
```

and thus no checking code need be generated

RAISING AN EXCEPTION EXPLICITLY

This is done by a raise statement:

```
raise exception_name;
```

The exception name may be predefined:

```
raise TASKING_ERROR;
```

Or, more commonly, it may be a name declared by the programmer:

```
raise TABLE_FULL;
```

Regardless of whether the exception is raised explicitly (by a raise statement) or implicitly, the effect is the same:

- Abandon execution of the construct that caused the exception to be raised

- Try to find a handler for the exception, where execution will resume

PREDEFINED EXCEPTIONS

CONSTRAINT_ERROR

- o Violation of range, index, discriminant constraint
- o Dereferencing a null access value

NUMERIC_ERROR

- o Predefined numeric operation that cannot deliver correct value

PROGRAM_ERROR

- o "Access before elaboration"
- o Reaching the end of a function

STORAGE_ERROR

- o Exhausting storage during a declaration, allocation, subprogram call, task activation

TASKING_ERROR

- o Failure of task activation or communication

EXCEPTION HANDLERS

An exception handler specifies the actions to be taken in response to the raising of an exception

It occurs as part of a <u>frame</u> -- a block statement or the body of a subprogram, package, task, or generic unit; e.g.,

```
begin
     Sequence of Statements          -- Statement Part
exception
     -- Exception Part:
     when E1 | E2 =>
          Sequence of Statements     -- Handler 1
     when E3 =>
          Sequence of Statements     -- Handler 2
     when others =>
          Sequence of Statements     -- Handler 3
end;
```

EXCEPTION HANDLERS (Cont'd.)

- An exception name is not allowed to appear twice in the **when** clauses of the same exception part

- An **others** choice, if present, must appear alone in the last **when** clause

- A **goto** statement cannot transfer control into a handler from outside (e.g., from the statement part)

- A **goto** statement cannot transfer control from a handler to a statement in the statement part

- Nesting is permitted: a handler may itself contain inner frames that have handlers

FINDING AN EXCEPTION HANDLER WHEN AN EXCEPTION IS RAISED

Consider the context in which the exception was raised:

- o In a statement in the statement part of a frame
- o In a statement in an exception handler
- o In a declaration

Consider also the kind of unit in which the exception was raised:

- o Block
- o Package body
- o Subprogram body
- o Task body (Ignore this case for now)
- o Library package specification

FINDING A HANDLER IN THE CURRENT FRAME

Simple case:

Exception ALPHA is raised in the statement part, and a handler for ALPHA or for <u>others</u> appears in the frame's exception part

Effect:

- o Abandon execution of the statement part
- o Go to the sequence of statements in the appropriate handler

Note:

If no exception is raised in the statement part, then none of the handlers in the exception part is executed

EXAMPLE OF SIMPLE CASE

Compute the "Dot Product" of a vector with itself.

If overflow occurs, write the error to an error file and return the maximum value for the element type.

```
type ELEMENT_TYPE is range -MAX_VAL .. MAX_VAL;
type VECTOR  is array (NATURAL range <>) of ELEMENT_TYPE;

function DOT_SELF(V: VECTOR) return ELEMENT_TYPE is
     SUM: ELEMENT_TYPE := 0;
begin
     for I in V'RANGE loop
          SUM := SUM + V(I)**2;
     end loop;
     return SUM;
exception
     when NUMERIC_ERROR | CONSTRAINT_ERROR =>
       TEXT_IO.PUT_LINE(ERROR_FILE, "ERROR IN DOT_SELF");
       return ELEMENT_TYPE'LAST;
end DOT_SELF;
```

FINDING AN EXCEPTION HANDLER

A less simple case:

Exception BETA is raised in the statement part of a frame, but there is a handler neither for BETA nor for <u>others</u> in the frame's exception part

Effect:

- o Abandon execution of the statement part of the frame
- o <u>Propagate the exception</u> in these cases, based on the kind of frame:
  - o Block or package body --

    Raise BETA after the block or package body
  - o Subprogram body --

    Raise BETA after the point of call

EXAMPLE: PROPAGATING AN EXCEPTION OUT OF A BLOCK

```
-- Find the largest integer <= N whose factorial can be computed

function FIND_MAX_FACTORIALIZABLE(N: NATURAL) return NATURAL is
     I: NATURAL := 1;
begin
     declare
          FACT: NATURAL := 1;
     begin
          while I <= N loop
               FACT := FACT * I;
               I := I + 1;
          end loop;
     end;
     return N;       -- Here if loop completed normally
exception
     when NUMERIC_ERROR =>
       return I;     -- Here if exception raised in loop
end FIND_MAX_FACTORIALIZABLE;
```

PROPAGATING AN EXCEPTION OUT OF A SUBPROGRAM

```
procedure PUSH(MY_ELEMENT: ELEMENT; MY_STACK: in out STACK) is
begin
     if STACK_FULL(MY_STACK) then
          raise OVERFLOW;
     end if;
     STACK_INDEX := STACK_INDEX + 1;
     MY_STACK(STACK_INDEX) := MY_ELEMENT;
end PUSH;
```

If MY_STACK is full, then OVERFLOW is raised but not handled:

- o Execution of PUSH is abandoned
- o OVERFLOW is raised immediately after the call of PUSH, with the caller determined dynamically

EXCEPTIONS AND SUBPROGRAM CALLS/RETURNS

- Exception raised in subprogram call:
  Handle in caller, not callee

```
declare
    procedure HAMLET(I: NATURAL) is ... end HAMLET;
begin
    ...
    HAMLET(-1);    -- Raise CONSTRAINT_ERROR here
    ...
end;
```

- Exception raised in function return:
  Handle in the function, if possible

```
function OPHELIA return NATURAL is
begin
    ...
    return -1;     -- Raise CONSTRAINT_ERROR here
    ...
end OPHELIA;
```

EXCEPTIONS AND SUBPROGRAM CALLS/RETURNS (Cont'd.)

o Exception raised on procedure return (assignment to copy-out parameter):

Handle in caller, not callee

```
declare
     I: INTEGER range 0..10 := 6;
     procedure POLONIUS(PARM: in out INTEGER) is
     begin
          PARM := 2*PARM;
          return;   *---------------------------
                                               |
     exception                                 |
                                               |
          when CONSTRAINT_ERROR => PARM := 0;  |
                                               |
     end POLONIUS;                             |
                                               |
begin                                          |
                                               |
     POLONIUS(I):                              |
                                               |
     <------------------------------------------
     *------------------------------------------
                                               |
exception                                      |
                                               |
     when CONSTRAINT_ERROR =>       <-----------
               I := 1;
end;
```

EXCEPTIONS AND UPDATING OF PARAMETERS

If an exception is propagated out of a procedure, then out and in out parameters passed by copy are not updated

Scalar and Access Types:

```
procedure CLAUDIUS(I: in out INTEGER) is
begin
     I := 0;
     raise CURTAIN;        -- Actual parameter not updated
end CLAUDIUS;

procedure GERTRUDE(I: in out INTEGER) is
begin
     I := 0;
     raise CURTAIN;
exception
     when others => null;     -- Actual parameter updated
end GERTRUDE;
```

Array and Record Types

- Compiler may choose either reference or copy
- Program erroneous if it matters

```
declare

     SWORD: EXCEPTION;
     subtype STRINGLET is STRING(1..4);
     S: STRINGLET := "ABCD";

     procedure LAERTES(STR: out STRINGLET) is
     begin
          STR := "SOFT";
          raise SWORD;
     end LAERTES;

begin
     LAERTES(S);
exception
     when SWORD =>
          if S = "SOFT" then
               PUT_LINE("By Reference");
          else
               PUT_LINE("By Copy");
          end if;
end;
```

PROPAGATING AN EXCEPTION OUT OF A HANDLER

```
procedure PUSH(MY_ELEMENT: ELEMENT; MY_STACK: in out STACK) is
begin
     STACK_INDEX := STACK_INDEX + 1;
     MY_STACK(STACK_INDEX) := MY_ELEMENT;
exception
     when CONSTRAINT_ERROR =>
          raise OVERFLOW;
end PUSH;
```

When the stack is full, CONSTRAINT_ERROR is raised by one of the statements in PUSH's statement part

The statement part is abandoned, and the handler for CONSTRAINT_ERROR in PUSH's exception part is executed

This in turn raises OVERFLOW, which is not handled by any inner handler

Thus OVERFLOW is propagated (to the point of subprogram call)

PROPAGATING AN EXCEPTION ANONYMOUSLY

A special form of raise statement, omitting the exception name, can be used only within a handler

Its effect is to propagate the "current exception", based on the kind of frame

It is useful in a handler for <u>others</u>; e.g.,

```
procedure GO_FOR_IT is
begin
     ...
exception
     when CONSTRAINT_ERROR =>
       ...  -- Take appropriate action and return
     when others =>
       PUT_LINE("Unexpected exception in GO_FOR_IT");
       raise;
end GO_FOR_IT;
```

HANDLING AN EXCEPTION RAISED IN A DECLARATION

- Look for the innermost frame containing the declaration. If one exists, then:

  - Abandon execution of the frame -- do not look for a handler in this frame's exception part

  - Propagate the exception based on the kind of frame

- If there is no such innermost frame, then the declaration occurs within a library package specification; thus:

  - Abandon elaboration of the package specification

  - Propagate the exception to the environment task (the main subprogram does not get called)

EXAMPLE: HANDLING AN EXCEPTION RAISED IN A DECLARATION

```
declare
   procedure WHY_NOT is
        N: INTEGER range 0..10 := 20;  -- CONSTRAINT_ERROR---
        ALPHA: array (1..N) of FLOAT;                        |
   begin                                                     |
        ...                                                  |
   exception                                                 |
        when CONSTRAINT_ERROR => ALPHA := (1..N => 0.0);     |
   end WHY_NOT;                                              |
begin                                                        |
   WHY_NOT;                                                  |
        <-----------------------------------------------------
        *-----------------------------------------------------
   return;                                                   |
exception                                                    |
   when CONSTRAINT_ERROR =>     <-----------------------------
             PUT_LINE("No problem");
end;
```

This example shows why Ada rules do not allow an exception raised in a declaration to be handled in the same frame

EXCEPTION DECLARATIONS

An exception declaration looks somewhat like an object declaration, and has similar namescope properties, but there are important differences:

- There are only a static number of exceptions -- If an exception is declared in a recursive subprogram, only one exception is created

  Rationale: run-time efficiency

- Exceptions are not permitted as components of arrays or records; they cannot be passed as parameters

  Rationale: no need for added generality

PROPAGATING AN EXCEPTION BEYOND ITS SCOPE

It can happen! E.g.,

```
procedure YOU_ASKED_FOR_IT is
     GLORP: EXCEPTION;
begin
     raise GLORP;
end YOU_ASKED_FOR_IT;
```

Why allow this?

- Disallowing it would add complexity, run-time cost
- Exceptions are not error situations -- they are <u>names</u> for error situations
  - Leaving the scope of an exception does not make the error condition disappear
  - The propagated exception may still be handled by an <u>others</u> handler

PROGRAMMING WITH EXCEPTIONS - OVERVIEW

- Handle a Terminating Condition
- Try Alternative Technique
- Retry an Operation
- Clean-Up ("Last Wishes")
- Use with Data Abstraction

HANDLING A TERMINATING CONDITION

Example: Looking up an item in a sequential file and returning its frequency count

```
package MUMBLE_SEQ_IO is new SEQUENTIAL_IO(MUMBLE);
use MUMBLE_SEQ_IO;

function COUNT_MUMBLES(MUMBLE_FILE: FILE_TYPE; PATTERN: MUMBLE)
          return NATURAL is
     COUNT: NATURAL := 0;
     CURRENT_MUMBLE: MUMBLE;
begin
     -- Assume that MUMBLE_FILE has already been opened
     loop
          READ(MUMBLE_FILE, CURRENT_MUMBLE);
          -- Raises END_ERROR when attempting to read EOF
          if CURRENT_MUMBLE = PATTERN then
               COUNT := COUNT + 1;
          end if;
     end loop;
exception
     when END_ERROR => return COUNT;
     -- Uses an exception as a normal condition for termination,
     -- not as an error
end COUNT_MUMBLES;
```

TRYING AN ALTERNATIVE TECHNIQUE

Find average of an array of INTEGER values; result is a FLOAT.

Use integer arithmetic, for efficiency, to sum the values.

Use floating point arithmetic if get integer overflow.

```
type VECTOR is array (NATURAL range <>) of INTEGER;

function AVERAGE(SAMPLES: VECTOR) return FLOAT is
     SUM: INTEGER := 0;
begin
     if SAMPLES'LENGTH = 0 then return 0.0; end if;
     for INDEX in SAMPLES'RANGE loop
          SUM := SUM + SAMPLES(INDEX);
     end loop;
     return FLOAT(SUM)/FLOAT(SAMPLES'LENGTH);
exception
     when NUMERIC_ERROR =>
          declare
               REAL_SUM: FLOAT := 0.0;
          begin
               for INDEX in SAMPLES'RANGE loop
                    REAL_SUM := REAL_SUM + FLOAT(SAMPLES(INDEX));
               end loop;
               return REAL_SUM/FLOAT(SAMPLES'LENGTH);
          end;
end AVERAGE;
```

RETRYING AN OPERATION

Example: Try to read a block of data from a tape.

If unsuccessful after ten tries, raise MALFUNCTION.

```
for I in 1..10 loop
     begin
          READ_TAPE(DATA);
          exit;     --------------------
                                       |
     exception                         |
                                       |
          when TAPE_ERROR =>           |
                                       |
               if I = 10 then          |
                                       |
                    raise MALFUNCTION; |
                                       |
               else                    |
                                       |
                    BACKSPACE;         |
                                       |
               end if;                 |
                                       |
     end;                              |
                                       |
end loop;                              |
                                       |
                    <------------------
```

CLEANING UP

Example: Allow a procedure to perform "last wishes" before propagating an exception:

Leave world in a consistent state for the caller

First approximation:

```
procedure OPERATE(NAME: STRING) is
     FILE: FILE_TYPE;
begin
     -- (1)  Initial actions
     OPEN(FILE, INOUT_FILE, NAME);
     -- (2)  Perform work on the file
     CLOSE(FILE);
     -- (3)  Final actions
end OPERATE;
```

Problem:

If an exception is raised during (2), FILE is left open

A Better Approach:

```
procedure SAFE_OPERATE(NAME: STRING) is
     FILE: FILE_TYPE;
begin
     --(1)  Initial actions
     OPEN(FILE, INOUT_FILE, NAME);
     begin
          -- (2) Perform work on the file
     exception
          when others =>
               CLOSE(FILE);
               raise;
     end;
     CLOSE(FILE);
     -- (3)  Final actions
end SAFE_OPERATE;
```

EXCEPTIONS AND ABSTRACT DATA TYPES

An abstract data type is a (private) type declared in a package specification together with the subprograms that operate on data of that type

- Declare exceptions that will be raised when the subprograms cannot complete normally
- Comment each subprogram specification by documenting both its normal and abnormal behavior
- Declare functions that the package user can call to see if an exception would be raised
- Program the visible subprograms so that they do not propagate anonymous or predefined exceptions; instead, propagate an explicitly declared exception

EXCEPTIONS AND ABSTRACT DATA TYPES: EXAMPLE

```
with STACK_INIT_PCKG;
package STACK_PCKG is
    type STACK is private;
    OVERFLOW, UNDERFLOW: EXCEPTION;

    procedure PUSH(MY_ELEMENT: ELEMENT; MY_STACK: in out STACK);
    -- Normal return: pushes MY_ELEMENT onto MY_STACK
    -- Abnormal return: raises OVERFLOW if stack is full

    function IS_FULL(MY_STACK: STACK) return BOOLEAN;
    -- Normal return: TRUE iff MY_STACK is full
    -- Abnormal return: none

    ... -- Analogous declarations for POP, IS_EMPTY, TOP
private
    MAX_STACK_SIZE: constant INTEGER := STACK_INIT_PCKG.MAX;
    type STACK is array (1..MAX_STACK_SIZE) of ELEMENT;
end STACK_PCKG;
```

TASKING AND EXCEPTIONS: OVERVIEW

- Exceptions raised during task activation
- Exceptions raised during execution of task statements
- Exceptions raised during task communication
    - Calling an entry of a completed task
    - Exception raised during execution of an accept statement
    - Communication with an "abnormal" task

EXCEPTIONS DURING TASK ACTIVATION

- Review of activation semantics

  - Parent unit suspended just after the <u>begin</u>
  - Activation: elaborate declarative parts of child tasks
  - Parent unit awakened, may execute in parallel with statements of children

- What if an exception is raised during activation of one (or more) of the children?

  - The child task becomes completed
  - When parent unit is awakened, TASKING_ERROR is raised (just after the <u>begin</u>)
  - If several children raise exceptions during their activation, TASKING_ERROR is raised only once

EXAMPLE: EXCEPTIONS DURING TASK ACTIVATION

```
declare
     task ALPHA;

     task body ALPHA is
          S: STRING(1..10) := "HI";  -- CONSTRAINT_ERROR
               -- CONSTRAINT_ERROR --> TASKING_ERROR ------
                                                            |
     begin                                                  |
          ...                                               |
     exception                                              |
          when CONSTRAINT_ERROR => ...;                     |
     end ALPHA;                                             |
begin                                                       |
     -- Activate ALPHA here                                 |
     -- When CONSTRAINT_ERROR is raised during activation,  |
     -- propagate TASKING_ERROR to this point               |
          <-------------------------------------------------
          *-------------------------------------------------
                                                            |
     ...                                                    |
exception                                                   |
     when TASKING_ERROR => ...     <------------------------
end;
```

EXCEPTIONS AND TASK ACTIVATION: RATIONALE

- An exception raised in a declarative part of a (task) unit is never handled in that unit

  => Divide task execution into two phases

- If an exception is raised in the declarative part of a task body, the parent unit should be notified that activation did not complete successfully

  => Activate just after the begin

- But do not raise an exception asynchronously

  => Suspend parent at the activation of children tasks

- And don't forget that children tasks may raise different exceptions during their activation

  => Raise TASKING_ERROR in parent

EXCEPTIONS DURING TASK EXECUTION

- If an exception is raised during execution of the statements in a task body, look for a handler in the exception part of that task body

- If a handler is not found, the task becomes a completed task

- Do <u>not</u> propagate either that exception or TASKING_ERROR back to the parent unit

  Why not?

  Because we do not want to raise exceptions asynchronously

  It is good style to put a **when others** choice in the exception part of a task body, perhaps to rendezvous with an error reporting task

EXCEPTIONS DURING TASK COMMUNICATION

After a server task completes its execution, TASKING_ERROR is raised in any task attempting to call the server's entries

```
task SERVER is
     entry UPDATE(THIS_ITEM: in out ITEM);
end SERVER;

task body SERVER is
     ...
begin
     ...
     accept UPDATE(THIS_ITEM: in out ITEM) do
          ... -- actions to update the item
     end UPDATE;
     ...
end SERVER;

task USER;

task body USER is
     MY_ITEM: ITEM;
begin
     ...
     SERVER.UPDATE(MY_ITEM);
     -- TASKING_ERROR raised if SERVER completes without
     -- accepting this call
     ...
end USER;
```

EXCEPTIONS DURING RENDEZVOUS

```
task SERVER is                               |    task USER;
                                             |
     entry SYNCH;                            |
                                             |
end SERVER;                                  |
                                             |
                                             |
task body SERVER is                          |    task body USER is
                                             |
     ...                                     |         ...
                                             |
begin                                        |    begin
                                             |
     ...                                     |         ...
                                             |
     accept SYNCH do                         |         SERVER.SYNCH;
                                             |
          ... raise GERBILS; *------------------->     ...
                             |               |
     end SYNCH;              |               |    end USER;
                             |               |
          <-------------------               |
                                             |
     ...                                     |
                                             |
end SERVER;                                  |
```

- o Propagate the raised exception both to caller and callee, at the points following the call and accept
  - o Important that both be notified of the failure
  - o The propagation is not asynchronous (caller was suspended)
- o In each task, look for handler via normal search rules

THE ABORT STATEMENT

"An abort statement causes one or more tasks to become abnormal, thus preventing any further rendezvous with such tasks"

-- Ada Reference Manual, Section 9.10, Paragraph 1

Semantics of rendezvous with abnormal tasks has interaction with exception facility

Syntax of abort statement:

```
abort task_name {, task_name};
```

Effect:

- o Each named task (and dependents) not yet terminated becomes abnormal
- o An abnormal task becomes completed when it is suspended at any of several tasking operations

Note:

- o Aborting a task does not necessarily cause immediate (or even eventual) termination
- o Use the abort statement sparingly; e.g., for "rogue" tasks or to prevent deadlock

RENDEZVOUS WITH ABNORMAL TASKS

- If a task calls an entry of an abnormal task, TASKING_ERROR is raised in the caller
- What if a task becomes abnormal during the rendezvous?
  - Finish execution of the accept statement
  - Then, effect depends on whether it is the user (calling) task or the server (called) task that is abnormal
    - If <u>user</u> task becomes abnormal:
      - No effect on server
      - User task becomes completed
    - If <u>server</u> task becomes abnormal:
      - TASKING_ERROR raised in user
      - Server task becomes completed

EXCEPTIONS AND OPTIMIZATION

- o Elimination of unnecessary checks
  - o Pragma SUPPRESS
  - o Compiler optimization techniques
- o "Termination" rather than "Resumption" model
- o Allowed code motions by compiler
- o Allowed machine instruction effects
  - o When to test overflow
  - o Widening intermediate ranges

SUPPRESSING CHECKS

- Time-critical applications may require absence of generated code that checks for error conditions

```
type VECTOR is array (1..100) of INTEGER;
pragma SUPPRESS(INDEX_CHECK, ON => VECTOR);
-- Now no explicit index check when subscripting or
-- slicing objects of type VECTOR
```

- Checks for CONSTRAINT_ERROR:

  ACCESS_CHECK   DISCRIMINANT_CHECK   INDEX_CHECK

  LENGTH_CHECK   RANGE_CHECK

- Checks for NUMERIC_ERROR:

  DIVISION_CHECK   OVERFLOW_CHECK

- Check for PROGRAM_ERROR: ELABORATION_CHECK

- Check for STORAGE_ERROR: STORAGE_CHECK

- Program is erroneous if error situation occurs and run-time check is absent

- Inclusion of pragma SUPPRESS does not guarantee that the exception will not be raised
  - It may be raised by hardware
  - Pragma SUPPRESS simply advises compiler not to generate code to check for the error situation

ALLOWED CODE MOTIONS

- o Reference: Ada Reference Manual, Section 11.6

- o Code motion must not introduce an exception that would not otherwise have been raised:

```
ALPHA:  array (1..N) of FLOAT;
...
for I in ALPHA'RANGE loop
     ALPHA(I) := ALPHA(I) / FLOAT(ALPHA'LENGTH)
end loop;
```

cannot be transformed into

```
ALPHA: array (1..N) of FLOAT;
...
<TEMP> := 1.0 / FLOAT(ALPHA'LENGTH);
for I in ALPHA'RANGE loop
     ALPHA(I) := ALPHA(I) * <TEMP>;
end loop;
```

unless compiler can ensure that range not null

ALLOWED CODE MOTIONS (Cont'd.)

- Code that may raise an exception must not be moved to a place that would cause a different handler to be invoked (unless the program effect remains the same)

- Programmer should be careful about assuming where in the statements a particular exception was raised

```
begin
     I := 1;
     J := Expr;     -- may raise NUMERIC_ERROR
exception
     when others =>  PUT(I);  -- I = 1
end;
```

may be transformed into:

```
begin
     <TEMP> := Expr;     -- may raise NUMERIC_ERROR
     I := 1;
     J := <TEMP>;
exception
     when others => PUT(I);   -- I /= 1
end;
```

OPERATOR/OPERAND ASSOCIATIONS

- An expression may be rearranged to yield a correct mathematical result, even though this may cause an exception to be raised

- Programmer may prevent such rearrangements by parenthesizing

- Example: I + J + K

  Assume I = 1, J = -1, K = INTEGER'LAST

  Then the compiler may choose to evaluate this as

  (I + K) + J

  though this raises NUMERIC_ERROR

ALLOWED MACHINE INSTRUCTION EFFECTS

- If an exception is raised in an assignment statement, the target variable must not be modified

  Thus, compiling the statement

  ```
  J := J + 1;
  ```

  is an issue, if the machine INCREMENT instruction modifies the variable before giving an overflow indication (can't use such an instruction)

  Programmer can help here by declaring J in a subrange

- Widening the range of an intermediate result

  ```
  I, J, K: INTEGER;
  ...
  I := (I * J) / K;
  -- May compute I * J as LONG_INTEGER,  to take
  -- advantage of MULTIPLY and DIVIDE instructions
  ```

CONCLUSIONS

- Exception handling is a useful control facility for dealing with rare run-time events (usually error conditions)
- When an exception is raised, the current action is abandoned and a handler is sought
- Search rules take into account where the exception is raised and the kind of enclosing unit
- User-declared exceptions are valuable part of data abstraction
- Interactions between exceptions and tasking have been carefully considered; exceptions always synchronous
- Optimizers are permitted reasonable flexibility in transforming programs that may raise exceptions