What You Always Wanted to Know About Exceptions But Were Afraid to “Raise”

Currie Colket
The MITRE Corporation

Phone: (703) 883-7381
Email: colket@mitre.org | colket@acm.org

SIGAda 2002
9 December 2002

Acknowledgement and Thanks to Bill Thomas and Bill Bail for their help and ideas.
So What Are Exceptions?

Slide intentionally left blank😊
Overview

- Introduction
- Major Benefits of Exceptions
- Ada 95 RM View of Exceptions (with Ada 83 deltas)
- Designing Software With Exceptions
  - Design Considerations
  - Ada 95 Quality and Style Guide
  - Design Examples
  - Planning for Exceptions
- Issues Using Exceptions - Exception Gotchas
- Future of Ada Exceptions (Exception Workshop)
- Conclusion
Support fault tolerance and safety critical requirements

- Hardware faults
- Network errors
- Capacity faults
- Error conditions

Ada predefined exceptions & User defined exceptions
Handled by an exception handler

Allows user to identify what to do when things go wrong and return to some known safe state
Introduction
Ada Exceptions

- Exceptions are
  - *Declared*, like types, variables, etc.; there are also some predefined exceptions
  - *Raised*, explicitly (by a raise statement) or implicitly (by Ada runtime)
  - *Caught*, explicitly (by name) or implicitly ("when others =>")
  - *Propagated*, from called subprogram to calling subprogram
- In general, exceptions are an excellent feature
  - Reliability enhancing feature designed to help specify program behavior in the presence of unexpected events
  - Allows separation of error-handling code from normal code
- However, good design and coding guidelines are essential
  - Cannot infer from a subprogram specification what exceptions the subprogram may propagate
- And automated checking for conformance to guidelines is beneficial
Introduction
Simple Example of Exceptions in Ada

procedure Quadratic (A, B, C : in Float ; R1, R2 : out Float) is
    D : Float ;
begin
    D := B**2 - 4.0*A*C ;
    R1 := (-B - Sqrt (D) ) / (2.0*A) ;
    R2 := (-B + Sqrt (D) ) / (2.0*A) ;
exception
    when Negative_Sqrt =>
        R1 := 0.0; R2 := 0.0; raise No_real_solutions ;
    when others =>
        Put_Line ("Problem in Quadratic") ;
        raise ;
end Quadratic ;

function Sqrt (X : Float) return Float is
begin
    if X < 0.0 then
        raise Negative_Sqrt ;
    end if ;
    ...
end Sqrt;
Benefits of Using Ada Exceptions

- Provides a reliability-enhancing language feature designed to help specify program behavior in the presence of errors or unexpected events
- Supports the logical differentiation of normal and exceptional control paths, which can lead to clarity and ease of understanding
- Avoids "overloading" a parameter with multiple layers of abstraction (i.e., products of computation and status of that computation), which can cause confusion
- Supports the notion of error status that should not be ignored (a notorious practice in, e.g., much C software)
- Intuitive way of dealing with results of interactions among asynchronous independent activities
  - No amount of precondition testing can avoid chance of encountering error condition (e.g., jammed cable)
- Supports the return to some known safe state
Ada 95 RM
Clause 11 Exceptions

11.1 Exception Declarations
11.2 Exception Handlers
11.3 Raise Statements
11.4 Exception Handling
  11.4.1 The Package Ada.Exceptions
11.5 Suppressing Checks
11.6 Exceptions and Optimization

Major Changes for Ada 83 => Ada 95

- Numeric_Error is a renaming of Constraint_Error and is obsolete.
- Notion of Exception_Occurrence is introduced (Ada.Exceptions).
- Interaction between exceptions and optimization is clarified.
- Accept statement may now have a handler.

Ada 83 code is fully upward compatible to Ada 95
Ada 95 RM
11.1 Exception Declarations

Predefined exceptions declared in package Standard:
- Constraint_Error,
- Program_Error,
- Storage_Error, and
- Tasking_Error;

Declaration of name for a user-defined exception:

```
exception_declaration ::= defining_identifier_list : exception;
```

Examples of user-defined exception declarations:

```
Node_Not_DEFINED : exception;
My_Error          : exception;
Overflow, Underflow : exception;
```

Numeric_Error from Ada 83 is obsolete

Also declared in other packages
Response to exceptions is specified by `exception_handler`:

```ada
handled_sequence_of_statements ::= sequence_of_statements
[exception
   exception_handler
   {exception_handler}]
```

`exception_handler ::= when [choice_parameter_specification:]`

```ada
   exception_choice { | exception_choice} =>
   sequence_of_statements
```

```ada
choice_parameter_specification ::= defining_identifier
exception_choice ::= exception_name | others
```
11.2 Exception Handlers - 2

handled_sequence_of_statements found in Frames, i.e., block_statement, subprogram_body, package_body, task_body, accept_statement, entry_body

Simplifies syntax of all frames for Ada 95, e.g.,

**Ada 95:**

```plaintext
subprogram_body ::= subprogram_specification is declarative_part
                 begin
                 handled_sequence_of_statements
                 end [designator];
```

**Different in Ada 83:**

```plaintext
subprogram_body ::= subprogram_specification is [declarative_part]
                 begin
                 sequence_of_statements
                 [exception
                 exception_handler
                 {exception_handler}]
                 end [designator];
```

However, no change in source from Ada 83 => Ada 95 :-(
Example: [in Frame, have Frame_Name:= “My_Procedure”;]

begin
  Open(File, In_File, "input.txt"); -- Ada.Sequential_IO
exception
  when E : Name_Error =>
    Put("Cannot open input file in ");
    Put( Frame_Name); -- Should be provided with Information
    Put_Line(Exception_Occurrence(E)); -- fully qualified name
    Put_Line(Exception_Message(E)); -- short message
    Put_Line(Exception_Information(E)); -- implementation defined
      -- see package Ada.Exceptions
  raise;
end;

Ada.IO_Exceptions contains:
  Status_Error, Mode_Error, Name_Error, Use_Error,
  Device_Error, End_Error, Data_Error, Layout_Error.
When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is abandoned; i.e., Control is transferred to the handler.

Then:

- If the construct is the sequence_of_statements of a handled_sequence_of_statements that has a handler with a choice covering the exception, the occurrence is handled by that handler; **Handler replaces abandoned portion of handled_sequence_of_statements**

- If the construct is a task_body or main program, the exception does not propagate further; The task dies; ditto for main program.

- Otherwise, the occurrence is propagated to the innermost dynamically enclosing execution, i.e., the occurrence is raised again in that context.
procedure A is
begin
  X;
  B;
  Y;
exception
  when My_Exception =>
    Log_Error;
end A;

procedure B is
begin
  P;
  C;
  Q;
exception
  when others =>
    raise;
end B;

procedure C is
begin
  R;
  raise My_Exception;
  S;
exception
  when My_Exception =>
    Log_Error;
    raise;
end C;

Calls to Y, Q, and S are not executed
Ada 95 RM
11.3 Raise Statements

Raise Statements “raise” an exception:

\[
\text{raise\_statement ::= raise [exception\_name];}
\]

Examples of raise statements:

\[
\begin{align*}
\text{raise Node\_Not\_Defined;} \\
\text{raise My\_Error;} \\
\text{raise Overflow;} \\
\text{raise;} \quad \text{-- called a re-raise (only allowed in a handler)}
\end{align*}
\]
package Ada.Exceptions is
    type Exception_Id is private;
    -- Attribute E'Identity returns the unique identity of the exception.
    Null_Id : constant Exception_Id;  -- Default initial value of objects of Exception_Id

type Exception_Occurrence is limited private;
    -- Each occurrence of an exception is represented by a value of Exception_Occurrence

    type Exception_Occurrence_Access is access all Exception_Occurrence;
    Null_Occurrence : constant Exception_Occurrence;

    function Exception_Identity(X : Exception_Occurrence) return Exception_Id;
    -- Returns full expanded name of the exception in upper case
    -- starting with a root library unit.
    -- Implementation-defined if exception declared in an unnamed block_statement

    function Exception_Name(Id : Exception_Id) return String;

    function Exception_Name(X : Exception_Occurrence) return String;
    -- Same as Exception_Name(Exception_Identity(X)).
procedure `Raise_Exception`(E : in Exception_Id; Message : in String := "");
-- Raises new occurrence. Default message is “Raise_Exception”

procedure `Reraise_Occurrence`(X : in Exception_Occurrence);
-- Similar to `Raise_Exception`, but reraises original Exception_Occurrence

function `Exception_Message`(X : Exception_Occurrence) return String;
-- Returns a one line nameless message associated with Exception_Occurrence

function `Exception_Information`(X : Exception_Occurrence) return String;
-- Returns implementation-defined information about the exception.

procedure `Save_Occurrence`(Target : out Exception_Occurrence;
Source : in Exception_Occurrence);
-- Copies Source to Target; Target can be used in `Reraise_Occurrence` later

function `Save_Occurrence`(Source : Exception_Occurrence)
return Exception_Occurrence_Access;
-- Creates new object, copies Source to object, returns access value designating
-- new object; result can be deallocated using _Unchecked_Deallocation

private
... -- not specified by the language
end Ada.Exceptions;
pragma Suppress gives *permission* to an implementation to omit certain *language-defined checks*

```
pragma Suppress(identifier [, [On =>] name]);
```

- Allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.
- For use within a package_specification, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification.

If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous.
11.5Suppressing Checks - 2

- **Identifiers identify checks:**
  - **Constraint_Error checks:** Access_Check, Discriminant_Check, Division_Check, Index_Check, Length_Check, Overflow_Check, Range_Check, Tag_Check
  - **Program_Error checks:** Elaboration_Check, Accessibility_Check
  - **Storage_Error checks:** Storage_Check
  - **Suppress all checks:** All_Checks

- **Examples:**
  
  pragma Suppress(Range_Check);
  
  pragma Suppress(Index_Check, On => Table);
  
  pragma Suppress(All_Checks);

---

Ada 83:
Checks for Numeric_Error Check N/A
Note Upward Compatible
The following additional permissions are granted to the implementation:

- An implementation need not always raise an exception when a language-defined check fails. Instead, the operation that failed the check can simply yield an undefined result. The exception need be raised by the implementation only if, in the absence of raising it, the value of this undefined result would have some effect on the external interactions of the program. In determining this, the implementation shall not presume that an undefined result has a value that belongs to its subtype, nor even to the base range of its type, if scalar. Having removed the raise of the exception, the canonical semantics will in general allow the implementation to omit the code for the check, and some or all of the operation itself.

- If an exception is raised due to the failure of a language-defined check, then upon reaching the corresponding exception_handler (or the termination of the task, if none), the external interactions that have occurred need reflect only that the exception was raised somewhere within the execution of the sequence_of_statements with the handler (or the task_body), possibly earlier (or later if the interactions are independent of the result of the checked operation) than that defined by the canonical semantics, but not within the execution of some abort-deferred operation or independent subprogram that does not dynamically enclose the execution of the construct whose check failed. An independent subprogram is one that is defined outside the library unit containing the construct whose check failed, and has no Inline pragma applied to it. Any assignment that occurred outside of such abort-deferred operations or independent subprograms can be disrupted by the raising of the exception, causing the object or its parts to become abnormal, and certain subsequent uses of the object to be erroneous, as explained in 13.9.1.

- The permissions granted by this clause can have an effect on the semantics of a program only if the program fails a language-defined check.
Design Considerations
Overview

- Thoughts on Exception Use Impacting Design
- Alternative Ways to Handle Exceptions
- Ada 95 Quality and Style Guide Recommendations
- Examples for Systems Design
Design Considerations
Thoughts on Exception Use Impacting Design

- Unhandled exceptions will propagate all the way up the calling sequence, causing termination of the task or program.
- Exception propagation can violate abstractions, forcing clients of abstractions to handle implementation details.
- Anonymous exceptions can result from the interaction of exception propagation (dynamic) with Ada scoping rules (static).
- Analysis of exception propagation can be difficult in large systems due to the mixture of dynamic and static aspects.
- Subtle interactions with other language features, e.g., propagation can be arbitrarily delayed waiting on termination of dependent tasks in a frame, propagation can expose parameter-passing mechanisms, etc.

Proper Design Can Result in Highly Effective Results
Design Considerations
Alternative Ways To Handle Exceptions

- How is the exception caught?
  - Explicitly ("when My_Exception =>")
  - Implicitly ("when others =>")
  - Not at all

- What is propagated out of the handler?
  - Nothing (stops)
  - Same exception (passes)
  - Different exception (maps)
Guideline:

- Use a name that indicates the kind of problem the exception represents.

Example:

Invalid_Name: exception;
Stack_Overflow: exception;

Rationale:

- Naming exceptions according to the kind of problem they are detecting enhances the readability of the code.
Guideline:
- Describe the complete interface to the program unit, including any exceptions it can raise and any global effects it can have.

Example:
-- Exceptions:
-- Node_Not_DEFINED – A node must be defined ...

Rationale:
- The purpose of a header comment on the specification of a program unit is to help the user understand how to use the program unit.
- None of this information can be determined from the Ada specification of the program unit.
Guideline:

- Comment on all data types, objects, and exceptions unless their names are self-explanatory.
  - The conditions under which an exception is raised should be commented.

Example:

```ada
Node_Already_Defined : exception;
  --| Raised when an attempt is made to define
  --| a node with an identifier which already
  --| defines a node.
```

Rationale:

- The reader has no other way to find out the exact meaning of the exception (without reading the code in the package body).
Guidelines:
- For unavoidable internal errors for which no user recovery is possible, declare a single user-visible exception. Inside the abstraction, provide a way to distinguish between the different internal errors.
- Do not borrow an exception name from another context.
- Export (declare visibly to the user) the names of all exceptions that can be raised.
- In a package, document which exceptions can be raised by each subprogram and task entry.
- Do not raise exceptions for internal errors that can be avoided or corrected within the unit.
- Do not raise the same exception to report different kinds of errors that are distinguishable by the user of the unit.
Guidelines (continued):
- Provide interrogative functions that allow the user of a unit to avoid causing exceptions to be raised.
- When possible, avoid changing state information in a unit before raising an exception.
- Catch and convert or handle all predefined and compiler-defined exceptions at the earliest opportunity.
- Do not explicitly raise predefined or implementation-defined exceptions.
- Never let an exception propagate beyond its scope.
Abstraction Using Ada Package Specification & Exceptions

package QUEUE is
  procedure DEQUEUE (Object : out Object_type);
  procedure ENQUEUE (Object : in   Object_type);
  function Is_Empty return Boolean;
  function Is_Full return Boolean;
  OVERFLOW, UNDERFLOW : exception;
end QUEUE;

From application:  ENQUEUE(Object);
                    DEQUEUE(Object);
Guideline:
• Provide handlers for Storage_Error.

Example:
   when Storage_Error =>
       Do_Some_Garbage_Collection;
       raise; -- Allowing subprogram to be called again in loop.

Rationale:
• Prepare handlers for the exception Storage_Error, and consider carefully what alternatives you may be able to include in the program for each such situation.
**Guideline:**

- Use blocks to define local exception handlers.

**Example:**

```ada
for I in 1 .. Sample_Limit loop
    Get_Samples_And_Ignore_Invalid_Data:
        declare
            -- declarations
        begin
            Get_Value(Current);
        exception
            when Constraint_Error =>
                null; -- Continue trying
            when Accelerometer_Device.Failure =>
                raise Accelerometer_Device_Failed;
        end Get_Samples_And_Ignore_Invalid_Data;
    exit when Current <= 0.0; -- Slowing down
end loop;
```

**Rationale:**

Local exception handlers can catch exceptions close to the point of origin and allow them to be either handled, propagated, or converted.
“Ada exceptions are a reliability-enhancing language feature designed to help specify program behavior in the presence of errors or unexpected events.

“Exceptions are not intended to provide a general purpose control construct.”

“Further, liberal use of exceptions should not be considered sufficient for providing full software fault tolerance (Melliar-Smith and Randell 1987).”

Sections =>
5.8.1 Handling Versus Avoiding Exceptions
5.8.2 Handlers for Others
5.8.3 Propagation
5.8.4 Localizing the Cause of an Exception
Guidelines:

- When it is easy and efficient to do so, avoid causing exceptions to be raised.
- Provide handlers for exceptions that cannot be avoided.
- Use exception handlers to enhance readability by separating fault handling from normal execution.
- Do not use exceptions and exception handlers as goto statements.
- Do not evaluate the value of an object (or a part of an object) that has become abnormal because of the failure of a language-defined check.
Example:
- Test an integer for 0 before dividing by it
- Call an interrogative function Stack_Is_Empty before invoking the pop procedure of a stack package
- Use exceptions where such a test is too expensive [Entry_Exists before each call to the procedure Modify_Entry simply to avoid raising the exception Entry_Not_Found] or
  - or too unreliable [Concurrent Operations]

Rationale:
- In many cases, it is possible to detect easily and efficiently that an operation you are about to perform would raise an exception.
- When fault handling and only fault handling code is included in exception handlers, the separation makes the code easier to read.
Guidelines:

- When writing an exception handler for `when others`, capture and return additional information about the exception through the `Exception_Name`, `Exception_Message`, or `Exception_Information` subprograms declared in the predefined package `Ada.Exceptions`.
- Use others only to catch exceptions you cannot enumerate explicitly, preferably only to flag a potential abort.
- During development, trap `others`, capture the exception being handled, and consider adding an explicit handler for that exception.

Rationale:

- Writing a handler without these subprograms limits the amount of error information you may see.
- Programming a handler for others requires caution. You should name the exception in the handler.
Example

```ada
with Ada.Exceptions; ...
function Valid_Choice return Positive is
   subtype Choice_Range is Positive range 1..3;
   Choice : Choice_Range;
begin
   Put ("Please enter your choice: 1, 2, or 3: ");
   Ada.Integer_Text_IO.Get (Choice);
   if Choice in Choice_Range then
      -- else garbage returned
      return Choice;
   end if;
   exception
      when Out_of_Bounds : Constraint_Error =>
         Put_Line ("Input choice not in range: ");
         Put_Line (Ada.Exceptions.Exception_Name (Out_of_Bounds));
         Skip_Line;
      when The_Error : others =>
         Put_Line ("Unexpected error: ");
         Put_Line (Ada.Exceptions.Exception_Information (The_Error));
         Skip_Line;
   end Valid_Choice;
```

Input choice not in range: Constraint_Error

Unexpected error: Program_Error
Guidelines:

- Handle all exceptions, both user and predefined.
- For every exception that might be raised, provide a handler in suitable frames to protect against undesired propagation outside the abstraction.

Rationale:

- The statement that “it can never happen” is not an acceptable programming approach.
- You must assume it can happen and be in control when it does. You should provide defensive code routines for the “cannot get here” conditions.
- You should catch the exception and propagate it or a substitute only if your handler is at the wrong abstraction level to effect recovery.
Ada 95 Quality & Style Guide – Section 5.8.4
Localizing the Cause of an Exception

Guidelines:

- Do not rely on being able to identify the fault-raising, predefined, or implementation-defined exceptions.
- Use the facilities defined in Ada.Exceptions to capture as much information as possible about an exception.
- Use blocks to associate localized sections of code with their own exception handlers.

Rationale:

- In an exception handler, it is very difficult to determine exactly which statement and which operation within that statement raised an exception.
- The predefined exceptions are candidates for conversion and propagation to higher abstraction levels for handling there.
- User-defined exceptions, being more closely associated with the application, are better candidates for recovery within handlers.
Section 5.9 is: ERRONEOUS EXECUTION AND BOUNDED ERRORS

Guidelines:
- Do not suppress exception checks during development.
- Minimize suppression of exception checks during operation.
- If necessary, during operation, introduce blocks that encompass the smallest range of statements that can safely have exception checking removed.

Rationale:
- If you disable exception checks and program execution results in a condition in which an exception would otherwise occur, the program execution is erroneous. The results are unpredictable.
- Further, you must still be prepared to deal with the suppressed exceptions if they are raised in and propagated from the bodies of subprograms, tasks, and packages you call.
- By minimizing the code that has exception checking removed, you increase the reliability of the program.
Section 5.9 is: ERRONEOUS EXECUTION AND BOUNDED ERRORS

Guideline:

- Prevent exceptions from propagating outside any user-defined Finalize or Adjust procedure by providing handlers for all predefined and user-defined exceptions at the end of each procedure.

Rationale:

- Using Finalize or Adjust to propagate an exception results in a bounded error (Ada Reference Manual 1995, §7.6.1). Either the exception will be ignored or a Program_Error exception will be raised.

[For Controlled Types - See RM §7.6 for Ada.Finalization]
Guidelines:
- Provide a handler for exception Program_Error whenever you cannot avoid a selective accept statement whose alternatives can all be closed (Honeywell 1986).
- Make systematic use of handlers for Tasking_Error.
- Be prepared to handle exceptions during a rendezvous.
- Consider using a when others exception handler.

Example:

```ada
Accelerate:
begin
    Throttle.Increase(Step);
exception
    when Tasking_Error  =>  ...
    when Program_Error  =>  ...
    when Throttle_Too_Wide =>  ...
    ...                
    when others =>  ...
end Accelerate;
```

Rationale:

- The exception `Program_Error` is raised if a selective `accept` statement is reached, all of whose alternatives are closed (i.e., the guards evaluate to `False` and there are no alternatives without guards), unless there is an else part. When all alternatives are closed, the task can never again progress, so there is by definition an error in its programming.

- Because an else part cannot have a guard, it can never be closed off as an alternative action; thus, its presence prevents `Program_Error`. However, an else part, a delay alternative, and a terminate alternative are all mutually exclusive, so you will not always be able to provide an else part.
Rationale (continued):

- The exception `Tasking_Error` can be raised in the calling task whenever it attempts to communicate. There are many situations permitting this. Few of them are preventable by the calling task.

- If an exception is raised during a rendezvous and not handled in the `accept` statement, it is propagated to both tasks and must be handled in two places.

- The handling of the `others` exception can be used to avoid propagating unexpected exceptions to callers (when this is the desired effect) and to localize the logic for dealing with unexpected exceptions in the rendezvous. After handling, an unknown exception should normally be raised again because the final decision of how to deal with it might need to be made at the outermost scope of the task body.
Guideline:
- Consider using an exception handler for a rendezvous within the main loop inside each task.

Rationale:
- Allowing a task to terminate might not support the requirements of the system. Without an exception handler for the rendezvous within the main task loop, the functions of the task might not be performed.
- The use of an exception handler is the only way to guarantee recovery from an entry call to an abnormal task. Use of the 'Terminated attribute to test a task’s availability before making the entry call can introduce a race condition where the tested task fails after the test but before the entry call.
Example:

```ada
loop
  Recognize_Degraded_Mode:
  begin
    case Mode is
    when Primary =>
      select
        CPP.Request_New_Coordinates (X, Y);
      or
        delay 0.25;
        -- Decide whether to switch modes;
    end select;
    when Degraded =>
      CPB.Request_New_Coordinates (X, Y);
    end case;
    ...
  exception
    when Tasking_Error | Program_Error =>
      Mode := Degraded;
  end Recognize_Degraded_Mode;
  end loop;
```

---

Ada 95 Quality & Style Guide – Section 6.3.1
Avoiding Undesired Task Termination - 2
Abnormal Task Termination - 1

Guidelines:

- Place an exception handler for others at the end of a task body.
- Consider having each exception handler at the end of a task body report the task’s demise.
- Do not rely on the task status to determine whether a rendezvous can be made with the task.

Rationale:

- A task will terminate if an exception is raised within it for which it has no handler. In such a case, the exception is not propagated outside of the task (unless it occurs during a rendezvous). The task simply dies with no notification to other tasks in the program. Therefore, providing exception handler for others, ensures that a task can regain control after an exception occurs.
Ada 95 Quality & Style Guide – Section 6.3.4
Abnormal Task Termination - 2

Example:

```ada
task type Track (My_Index : Track_Index) is

... end Track;

---------------------------------------------------------------------

task body Track is
    Neutral : Boolean := True;
begin -- Track
    select
        accept ...
            ...
            or
            terminate;
    end select;

... exception
    when others =>
        if not Neutral then Station(My_Index).Status := Dead;
    end if;
end Track;
```

Should the task terminate due to an exception, it signals the fact in one of its parent’s data structures, “Station”.
Guideline:

- Avoid using the exception `Numeric_Error`

Rationale:

- Ten years of reflection on the use of Ada 83 led to the conclusion that some features of the original language are not as useful as originally intended.
- It would have been desirable to remove the obsolescent features completely, but that would have prevented the upward compatible transition from Ada 83 to Ada 95.
- Thus, the obsolescent features remain in the language and are explicitly labeled as such in Annex J of the Ada Reference Manual (1995). The features listed in Annex J are candidates for removal from the language during its next revision.
Guidelines:

- Do not depend on the exact locations at which predefined exceptions are raised.
- Do not rely on the behavior of Ada.Exceptions beyond the minimum defined in the language.
- Do not raise implementation-specific exceptions.
- Convert implementation-specific exceptions within interface packages to visible user-defined exceptions.

Rationale:

- Predefined exceptions may be raised from different locations. Predefined exceptions are associated with a variety of conditions.
- No exception defined specifically by an implementation can be guaranteed to be portable to other implementations.

Do not allow yourself to be forced to find and change the name of every handler you have written for these exceptions when the program is ported.
Ada 95 Quality & Style Guide – Section 8.2.7
Reusability - Exceptions

**Guideline:**
- Propagate exceptions out of reusable parts. Handle exceptions within reusable parts only when you are certain that the handling is appropriate in all circumstances.
- Propagate exceptions raised by generic formal subprograms after performing any cleanup necessary to the correct operation of future invocations of the generic instantiation.
- Leave state variables in a valid state when raising an exception.
- Leave parameters unmodified when raising an exception.

**Rationale:**
- When an exception is raised by a generic formal subprogram, the generic unit is in no position to understand why or to know what corrective action to take.
- The generic unit must first clean up after itself, restoring its internal data structures to a correct state so that future calls may be made to it after the caller has dealt with the current exception.
Design Considerations
Example 1: System Example

Where handled?
Same Strategy across system?
What is approach for tasking and protected objects?
What are repercussions?
Is critical data protected?

How reported?
Are there differences between test & production?
What is criticality of error?
Any Operator decisions?

System Control

Navigation
Sensor Control
Tracking
Comm

Radar
Infra Red
ESM
Acoustics

What do we do if exception raised Here?
Or Here?
### Design Considerations

**Example 1: Planning For Exceptions - 1**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Type of Error</th>
<th>Located in System</th>
<th>Located in Module</th>
<th>Impact of Error</th>
<th>Notes on Method of Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>INS_1_Timeout_Error</td>
<td>Initial 1 Fail</td>
<td>AOP</td>
<td>Navigation</td>
<td>Navigation Degraded</td>
<td>Timeout = 50 milliseconds</td>
</tr>
<tr>
<td>INS_2_Timeout_Error</td>
<td>Initial 1 Fail</td>
<td>AOP</td>
<td>Navigation</td>
<td>Navigation Degraded</td>
<td>Timeout = 50 milliseconds</td>
</tr>
<tr>
<td>INS_3_Timeout_Error</td>
<td>Initial 1 Fail</td>
<td>AOP</td>
<td>Navigation</td>
<td>Navigation Degraded</td>
<td>Timeout = 50 milliseconds</td>
</tr>
<tr>
<td>Voting_Plane_INS_1_Bad_Error</td>
<td>Initial 1 Computation Bad</td>
<td>AOP</td>
<td>Navigation</td>
<td>Only 2 INS's agree</td>
<td>Threshold = 0.000001 Seconds</td>
</tr>
<tr>
<td>Voting_Plane_INS_2_Bad_Error</td>
<td>Initial 2 Computation Bad</td>
<td>AOP</td>
<td>Navigation</td>
<td>Only 2 INS's agree</td>
<td>Threshold = 0.000001 Seconds</td>
</tr>
<tr>
<td>Voting_Plane_INS_3_Bad_Error</td>
<td>Initial 3 Computation Bad</td>
<td>AOP</td>
<td>Navigation</td>
<td>Only 2 INS's agree</td>
<td>Threshold = 0.000001 Seconds</td>
</tr>
<tr>
<td>INS_1_Out_To_Lunch_Error</td>
<td>Initial 1 Computation Bad 10 Times</td>
<td>AOP</td>
<td>Navigation</td>
<td>INS_1 Needs Initialization</td>
<td>Raised by Voting_Plane_INS_1_Bad_Error</td>
</tr>
<tr>
<td>INS_2_Out_To_Lunch_Error</td>
<td>Initial 2 Computation Bad 10 Times</td>
<td>AOP</td>
<td>Navigation</td>
<td>INS_2 Needs Initialization</td>
<td>Raised by Voting_Plane_INS_2_Bad_Error</td>
</tr>
<tr>
<td>INS_3_Out_To_Lunch_Error</td>
<td>Initial 3 Computation Bad 10 Times</td>
<td>AOP</td>
<td>Navigation</td>
<td>INS_3 Needs Initialization</td>
<td>Raised by Voting_Plane_INS_3_Bad_Error</td>
</tr>
<tr>
<td>FTP_invalid_error</td>
<td>Operator entered Fly To Point Invalid</td>
<td>AOP</td>
<td>Navigation</td>
<td>Operator Must Reenter</td>
<td>FTP net within 500 nm of current position</td>
</tr>
<tr>
<td>NavigationSubsystem_error</td>
<td>Variety of Error Conditions</td>
<td>AOP</td>
<td>Navigation</td>
<td>All Nav Reinitialized</td>
<td>Variety of Error Conditions</td>
</tr>
</tbody>
</table>

---
Design Considerations

Example 1: Planning For Exceptions - 2

<table>
<thead>
<tr>
<th>Exception Handler Description</th>
<th>G4</th>
<th>H4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception</strong></td>
<td>Log</td>
<td>Alert Operator</td>
</tr>
<tr>
<td>INS 1_TIMEOUT_Error</td>
<td>Yes</td>
<td>Warning</td>
</tr>
<tr>
<td>INS 2_TIMEOUT_Error</td>
<td>Yes</td>
<td>Warning</td>
</tr>
<tr>
<td>INS 3_TIMEOUT_Error</td>
<td>Yes</td>
<td>Warning</td>
</tr>
<tr>
<td>Voting Plane INS 1_END_Error</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Voting Plane INS 2_END_Error</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Voting Plane INS 3_END_Error</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>INS 1_Out_TO_Lunch_Error</td>
<td>Yes</td>
<td>Warning</td>
</tr>
<tr>
<td>INS 2_Out_TO_Lunch_Error</td>
<td>Yes</td>
<td>Warning</td>
</tr>
<tr>
<td>INS 3_Out_TO_Lunch_Error</td>
<td>Yes</td>
<td>Warning</td>
</tr>
<tr>
<td>Voting Plane No_Agree_Error</td>
<td>Yes</td>
<td>Severe</td>
</tr>
<tr>
<td>FTP Switchoff_Error</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Navigation SubsystemError</td>
<td>Yes</td>
<td>Severe</td>
</tr>
<tr>
<td>Perhaps Other Columns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

53
Design Considerations
Example 2: ASIS Client - Server Example

- The Ada Semantic Interface Specification (ASIS) exception handling mechanism is a good example demonstrating the types of things useful to do in a design.

*The Ada Semantic Interface Specification (ASIS) is an interface between an Ada environment (as defined by ISO/IEC 8652:1995) and any tool or application requiring information from it. An Ada environment includes valuable semantic and syntactic information. ASIS is an open and published callable interface which gives CASE tool and application developers access to this information. ASIS has been designed to be independent of underlying Ada environment implementations, thus supporting portability of software engineering tools while relieving tool developers from needing to understand the complexities of an Ada environment's proprietary internal representation. In short, ASIS can provide the foundation for your code analysis activities.*
What is ASIS?

Ada Source Code → Compile → Ada Environment → Link → APPLICATION SYSTEM

ASIS Interface

propositional interface

standard interface

CASE Tool or Application

Provides Syntactic and Semantic Information from Ada Environment using a standard interface
Ada syntax is summarized in Ada 95 RM, Annex P as variant of Backus-Naur Form

For example:

object_declaration ::= 
defining_identifier_list : [aliased] [constant] subtype_indication [:= expression]; | ...

For the Ada object declaration =>

A, B: Latitude := 0.0;

Syntactic Element
Tree Representation =>

ASIS can extract desired syntactic information for every syntactic category
Of the 367 ASIS Queries, most support syntactic tree analysis
These mechanisms allow ASIS to traverse the syntactic tree like Hypertext allows one to traverse a document.
Package Asis provides:

- **Common types:**
  - ASIS_Integer, ASIS_Natural, ASIS_Positive,
  - List_Index,
  - Context,
  - Element, Element_List, Element Subtypes,
  - Element Kinds (collection of enumeration types),
  - Compilation_Units, Compilation_Unit_List,
  - Unit Kinds (collection of enumeration types),
  - Traverse_Control, and
  - Program_Text (subtype of Wide_String)
- Queries via 20 visible child packages
- Ada Exceptions for errors with Status & Diagnostic information

Asis and child packages encapsulate vendor dependencies
*Designed to be portable for all implementations*
ASIS Package Architecture

Tool or Application using ASIS

ASIS

Ada_Environments
  Containers

Compilation_Units
  Times
  Relations

Implementation
  Permissions

Text

Statements

Clauses

Elements

Iterator

Declarations

Expressions

Definitions

Exceptions

Errors

Ids

Data_Decomposition (optional)

Portable_Transfer
  Portable_Constrained_Subtype

Portable_Unconstrained_Record_Type
  Portable_Array_Type_1
  Portable_Array_Type_2
  Portable_Array_Type_3

Asis. Exceptions for Exceptions to Application

Asis. Errors for Error Types

Asis. Implementations for Status & Diagnostic Info
Design Considerations

ASIS Exceptions - Example

package Asis.Exceptions is  -- Clause 5 in ISO/IEC 15291: 1999
  ASIS_Inappropriate_Context : exception;
  ASIS_Inappropriate_Container : exception;
  ASIS_Inappropriate_Compilation_Unit : exception;
  ASIS_Inappropriate_Element : exception;
  ASIS_Inappropriate_Line : exception;
  ASIS_Inappropriate_Line_Number : exception;
  ASIS_Failed : exception;

-- This is a catch-all exception that may be raised for different reasons
-- in different ASIS implementations. All ASIS routines may raise ASIS_Failed
-- whenever they cannot normally complete their operation. This exception
-- will typically indicate a failure of the underlying ASIS implementation.
end Asis.Exceptions;
Design Considerations - Example
ASIS Exceptions used with ASIS Errors

- ASIS reports all operational errors by raising an exception.
- Whenever an ASIS implementation raises one of the exceptions declared in package Asis.Exceptions, it will previously have set the values returned by the Status and Diagnosis queries to indicate the cause of the error.
- The possible values for Status are indicated in the definition of Error_Kinds, with suggestions for the associated contents of the Diagnosis string as a comment.
- The Diagnosis and Status queries are provided in the Asis.Implementation package to supply more information about the reasons for raising any exception.
- ASIS applications are encouraged to follow this same convention whenever they explicitly raise any ASIS exception--always record a Status and Diagnosis prior to raising the exception.
Design Considerations - Example
Package ASIS Errors defines the kinds of errors

package Asis.Errors is
  type Error_Kinds is (Not_An_Error, Value_Error, Initialization_Error, Environment_Error, Parameter_Error, Capacity_Error, Name_Error, Use_Error, Data_Error, Text_Error, Storage_Error, Obsolete_Reference_Error, Unhandled_Exception_Error, Not_Implemented_Error, Internal_Error);
end Asis.Errors;

-- Clause 4 in ISO/IEC 15291: 1999
-- No error is presently recorded
-- Routine argument value invalid
-- ASIS is uninitialized
-- ASIS could not initialize
-- Bad Parameter given to Initialize
-- Implementation overloaded
-- Context/unit not found
-- Context/unit not use/open-able
-- Context/unit bad/invalid/corrupt
-- The program text cannot be located
-- Storage_Error suppressed
-- Argument or result is invalid due to
-- and inconsistent compilation unit
-- Unexpected exception suppressed
-- Functionality not implemented
-- Implementation internal failure
Design Considerations - Example
Status Reporting


    function Status return Asis.Errors.Error_Kinds; -- Clause 6.9
        -- Returns the Error_Kinds value for the most recent error.
    function Diagnosis return Wide_String; -- Clause 6.10
        -- Returns a string value describing the most recent error.
        -- Will typically return a null string if Status = Not_An_Error.
    procedure Set_Status -- Clause 6.11
            Diagnosis : in Wide_String := """);
        -- Status    - Specifies the new status to be recorded
        -- Diagnosis - Specifies the new diagnosis to be recorded
        -- Sets (clears, if the defaults are used) the Status and Diagnosis
        -- information. Future calls to Status will return this Status (Not_An_Error)
        -- and this Diagnosis (a null string).
        -- Raises ASIS_Failed, with a Status of Internal_Error and a Diagnosis of
        -- a null string, if the Status parameter is Not_An_Error and the Diagnosis
        -- parameter is not a null string.
end Asis.Implementation;
Design Considerations - Example
Notional Exception Handler

exception
    when Asis.Exceptions.ASIS_Inappropriate_Context
        | Asis.Exceptions.ASIS_Inappropriate_Container
        | Asis.Exceptions.ASIS_Inappropriate_Compilation_Unit
        | Asis.Exceptions.ASIS_Inappropriate_Element
        | Asis.Exceptions.ASIS_Inappropriate_Line
        | Asis.Exceptions.ASIS_Inappropriate_Line_Number
        | Asis.Exceptions.ASIS_Failed
    =>
    Put (Asis.Implementation.Diagnosis);
    Put ("Status Value is ");

when others =>
    Put_Line ("Asis Application failed because of non-ASIS reasons");
end ASIS_Call_Tree_Example:

Full compilable example on ASIS Home Page:

http://www.acm.org/sigada/wg/asiswg
Design Considerations - Example Error Message

08:13:40 *** UNEXPECTED ERROR, EXCEPTION:

ASIS.EXCEPTIONS.ASIS_INAPPROPRIATE_ELEMENT

DIAGNOSIS: Raise location: #015407FC. Asis Diagnosis: Element is Nil. Internal => Asis_Declarations.Object_Declaration_Definition (SOL,120.0.2.1)>>STATUS: VALUE_ERROR, Version: 120.0.2.1, UNIT: /project/AAAV/aaav200101/gunner.ss/sun4_solaris2_ada95_3_2_0.wrk/unitss/battlesight_range_adjust_pkg.2.ada, On line: 541

08:12:31 *** Traversal of unit /project/AAAV/aaav200101/gunner.ss/sun4_solaris2_ada95_3_2_0.wrk/unitss/mailbox_pkg.2.ada terminated. Diagnosis: Exception: ADA.IO_EXCEPTIONS.DATA_ERROR (raised at #015407FC)

Internal =>
Asis_Elements.Traverse_Element.Pre_Operation/DN_SUBPROGRAM_BODY:
Asis_Elements.Walk'N(2)/DN_SUBPROGRAM_BODY:
Asis_Elements.Walk'N(1)/DN_SUBPROGRAM_BODY:
Asis_Elements.Children/DN_ITEM_S: Asis_Elements.Walk'N(1)/DN_ITEM_S:
Asis_Elements.Children/DN_BLOCK: Asis_Elements.Walk'N(2)/DN_BLOCK:
Asis_Elements.Walk'N(1)/DN_BLOCK:
Asis_Elements.Children/DN_PACKAGE_BODY:
Asis_Elements.Walk_First/DN_PACKAGE_BODY:
Asis_Elements.Traverse_Element (SOL,120.0.2.1) Status: UNHANDLED_EXCEPTION_ERROR
Exception Gotchas - 1
Function Handlers Must Raise Exception or Return

- Functions must either return a value or propagate an exception. Otherwise the exception Program_Error is raised. Six instances were detected where an exception handler would handle exceptions raised, but would neither return a known safe value or propagate an exception. The fact that the designer added the exception handler implied that exceptions were possible.

```ada
function Next_Packet return Rcv_Type is
  begin
    ... exception
      when Exception_Id: others =>
        Log_Error (Package_Id & ".Next_Packet: Unhandled exception: " &
                   Ada.Exceptions.Exception_Information (Exception_Id));
  end Next_Packet;
```

6 Violations in recent analysis
Exception Gotchas - 2
Unset Out Parameters

● When a handler does not propagate the exception, out parameters of a procedure and return values of functions should be set. In the case of a procedure, if the out parameters are not set appropriately, the calling program is likely to use them erroneously.

```ada
function Compute_Range return Integer is
    ... begin
    ... exception
        when Range_Not_Computable => return 0;
        when others =>
            Log_Error (Package_Id & ".Next_Packet: Unhandled exception: "
                         & Ada.Exceptions.Exception_Information (Exception_Id));
            raise;
    end Next_Packet;
```

Acceptable return value

Either prevents Program_Error

Acceptable to reraise the exception
Exception Gotchas - 3
Missing Storage Error Handler For Allocators

- Any allocator can exhaust the available space for the collection, the use of allocators should be limited and the "out of memory" case handled locally. An exception handler for Storage Error should be provided in the local scope for each allocator.

143 Violations in recent analysis
Exception Gotchas - 4
Exception handler with "when others => null"

- Use of a "when others" whose statement body is "null" may be inappropriate in that they catch all exceptions but provide no further processing of conditions that led to the exception.

- Exceptions should be used to trap expected problems and revert to some known safe state. These are normally classified as serious errors since should an exception be raised, processing continues which is likely degraded.

```java
when others =>
  null;
```

Catches the exception, but erroneous problem is not resolved, resulting in erroneous execution, which some folks call “Graceful Degradation”

13 Violations in recent analysis
Exception Gotchas - 5
Use of When Others in an Exception Handler

- Such handlers are a catchall and may be inappropriate in some cases. It prevents the opportunity to return the system to a known safe state based on the exception named. Typical action is to log the exception and to allow the system to perform in a degraded state. It is better practice to handle all potential exceptions explicitly. It should be noted that it is quite valuable to have such an exception handler within a looped block for tasks and main programs, as the absence there can result in a system crash.

```ada
when Exception_Id: others =>
  Log_Error (Package_Id & ".Next_Packet: Unhandled exception: " &
             Ada.Exceptions.Exception_Information (Exception_Id));
```

Logs the exception, but erroneous problem is not resolved allowing for “Graceful Degradation”

659 Violations in recent analysis

Useful to Have “when others” at Subsystem/System Level with operator Decision to Reinitialize Subsystem/System
Exception Gotchas - 6
Raised Exceptions Non Propagating

- Explicit raising of exceptions that are caught in the local scope is similar in nature to a Goto statement. Use of exceptions in this manner represents another form of an unstructured program jump. It makes programs harder to understand, test, and modify.

- If the problem can be resolved at the local level, perhaps the use of exceptions is the wrong abstraction.

7 Violations in recent analysis
Exception Gotchas - 7
Raise Predefined Exceptions

- Raising predefined exceptions adds confusion as to the source of the exception. The declaration of application exceptions keeps system run-time errors and application errors separate. This is considered to be a poor programming practice.

```
Raise Constraint_Error;
```

8 Violations in recent analysis all for Constraint_Error
This is the declaration of an exception in a non-visible part of the program. Non-visible declarations can be very dangerous as they can be only handled within the scope of the declaration (except with a `when others`). Unintended propagation outside this scope may impact remote sections of the code and be a difficult error to find.

```plaintext
package QUEUE is
    procedure DEQUEUE (Object : out Object_type);
    procedure ENQUEUE (Object : in   Object_type);
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
end QUEUE;

and in body:
    OVERFLOW, UNDERFLOW : exception;
```

11 Violations in recent analysis

Declarations of exceptions in body is not useful for desired abstractions
Subprograms should not raise exceptions that are outside the visible scope of calling programs. This creates a serious problem where the exception cannot be handled by name to take the appropriate action for the raised exception. The raised exception can only be handled by a "when others =>" option which cannot distinguish which exception has been raised.

```vhdl
procedure Erroneous_Propagation_Demo is
  My_Exception: exception;
begin
  ...
  raise My_Exception;
exception
  when My_Exception => raise;
end Erroneous_Propagation_Demo;
```

4 Violations in recent analysis

Propagates Outside of Scope
Future of Ada Exceptions - 1

The aims of the workshop were:

- To share experience on how to build modern systems that have to deal with abnormal situations;
- To discuss how solutions to those needs can be developed employing standard Ada features including the current exception handling paradigm; and
- To propose new exception handling mechanisms / paradigms that can be included in future revisions of the Ada language and also fit high integrity language profiles for safety critical systems.
Future of Ada Exceptions - 2

Some of the issues addressed at Workshop:

- Exception handling model remains based on Ada 83 while Ada 95 is object oriented.
- Exceptions and concurrency are not well integrated
  - Task with an unhandled exception dies silently
  - Asynchronous transfer of control for passing exceptions asynchronously between tasks
- New fault tolerance schemes based on existing exception handling facilities have been developed in research environments
  - Higher level abstractions providing more advanced mechanisms
Future of Ada Exceptions - 3

Ideas Emerged:

- Extensions to Exception Handling should be downward compatible with Ada95
- Exceptions can be (generic) parameters
- Unhandled Exceptions must be declared in the subprogram header (signature)
- Exceptions are types
- It should be possible to know if (for a controlled type) “Finalize” is called with a pending exception
- Visibility Control should be controlled
- There should be some notification mechanism for tasks that complete by raising an exception
- And as a study topic: Instead of associating exceptions with subprograms, it would be nice to be able to associate exceptions with classes, types, objects, etc.
Future of Ada Exceptions - 4

For more Information:

Ada Letters Volume XXI, Number 1, September 2001
Special Issue -
Exception Handling for a 21st Century
Programming Language Proceedings
Conclusion

- Ada 95 Exceptions can be effective in supporting situations that might otherwise result in erroneous execution.
- Handled exceptions can be used to return the system to some known safe state or to propagate the exception to a handler that can return the system to some known safe state.
- The Ada.Exceptions package provides excellent support to help debug a program. If you are not currently using it, plan to use it.
- Exceptions must be used intelligently as improper use can result in serious errors including Program_Error.
- Sound Software Engineering includes planning for Exceptions during the Design Phase.
- Code Analysis Tools can be essential.
- There are likely to be enhancements to exceptions for Ada 0X.