Séance 4

La gestion des Fichiers et des Exceptions

Objectifs :

- ✓ Les concepts de base pour la gestion des fichiers
- ✓ Exploration des paquetages Input/Output
- ✓ Comprendre la gestion des Exceptions.

Gestion des fichiers: Les concepts de base

- Les fichiers internes et externes
  - Interne: Clavier, Affichage sur écran.
  - Externes : Sur disque.
- Les modes d'ouverture de fichier
  - In : le programme peut lire les données à partir du fichier
  - Out : le programme peut écrire des données dans le fichier
- L'accès direct aux fichiers doit avoir le mode *in-out*.
- L'accès séquentiel doit avoir le mode *Append.*

Les paquetages Input/Output

- La lecture/ecriture des fichiers textuels utilise les paquetage :
  - Ada.Text_IO
  - Ada.Wide_Text_IO
- Les deux paquetages fournissent un type privé nommé «File_Type».
  - Type File_Mode is (In_File, Out_File, Append_File)
- Voir aussi Ada.sequential_IO et Ada.Direct_IO.
  - Type File_Mode is (In_File, InOut_File, Out_File)

Instanciation

- Package Sequential_Integer_IO is new Ada.Sequential_IO (Integer);
- Package Direct_Integer_IO is new Ada.Direct_IO (Integer);
- Exception possibles:
  - Status_Error: utilisation d'un fichier interne qui n'est pas associé à un fichier externe.
  - Mode_Error: opération input sur un fichier en mode out ou append.
  - Name_Error: Spécification d'un nom de fichier externe invalide au moment de l'association avec le fichier interne.
  - Use_Error: opération input/output sur un fichier externe dont l'environnement n'accepte pas cette opération.
  - Device_Error: Problème matériel, logiciel ou de média qui fournit le service input/output.
  - End_Error: tentative de lecture après la fin de ce dernier.
  - Data_Error: les données ne respectent pas la forme attendue.
  - Layout_Error: opération de formatage des données textuelles invalide.

Manipulation des fichiers externes

```
Procedure Create  (File: in out File_Type;
                   Mode : in File_Mode := Out_File;
                   Name : in String := "";
                   Form : in String := "");

Procedure Open    (File: in out File_Type;
                   Mode : in File_Mode;
                   Name : in String;
                   Form : in String := "");

--// La version fourni avec Ada_Direct_IO est déclarée de la même façon
--// avec Mode : Inout_File
```

- Name et Form décrivent le fichier externe
- File est le nom de fichier interne

```
begin
   Open (File, Out_File, Name, Form);

exception
   When Name_Error =>
      Create (File, Out_File, Name, Form);
```

```
procedure Close (File: in out File_Type);    -- Élimine l'association entre le fichier interne et externe
procedure Delete (File: in out File_Type);   -- Élimine l'association et efface le fichier externe.
```

Initialisation de l'ouverture du fichier

```
procedure Reset (File: in out File_Type; Mode : in File_Mode) -- avec changement du mode d'ouverture
procedure Reset (File: in out File_Type)  -- reouverture du fichier sans changer de mode.
```

- Récupération de l'état du fichier

```
function Is_Open(File: File_Type)    return boolean;
function Name(File: File_Type)       return String;
function Form(File: File_Type)       return String;
function Mode(File: File_Type)       return File_Mode;
--// Fixer les valeurs

procedure Set_Line_Length (File: in File_Type; To : in Count);
procedure Set_Page_Length (File: in File_Type; To : in Count);
procedure New_Line (File: in File_Type; Spacing: in Positive_Count : =1);
procedure New_page (File: in File_Type);
procedure Set_Col (File: in File_Type; To : in Positive_Count );
procedure Set_Line(File: in File_Type; To : in Positive_Count );

--// Recuperer les valeurs

function Page (File: in File_Type) return Positive_Count;
function Line (File: in File_Type) return Positive_Count;
function Col  (File: in File_Type) return Positive_Count;
```

Entrées/Sorties

```
procedure Get (File: in File_Type; item: out Character);
procedure Get (File: in File_Type; item: out String);

procedure Put (File: in File_Type; item: in Character);
procedure Put (File: in File_Type; item: in Character);
```

- Exemple

```
procedure Print_dans_fichier is
   Output_File: File_Type;
begin
   Create (Output_File,Name=> "2778.Dat");
   for B in 2 .. 20 loop
      Put (Output_File, Item => 2778, Base => B);
      New_Line (Output_File);
   end loop;
   Close (Output_File);

end Print_dans_fichier;
```

EXCEPTIONS

- Ada fournit un mécanisme d'exception qui permet de transférer le contrôle en dehors du cadre où un évènement se produit.
- Les évènements qui déclenchent des exceptions sont:
  - end_of_file
  - queue_overflow
  - divise_by_zero
  - storage_overflow
  - ....

EXCEPTIONS PRÉDÉFINIES

- **Constraint_Error (Numeric_Error)** : qui correspond généralement à un dépassement des bornes, ce qui inclut le cas des problèmes en arithmétique tel que la division par zéro;
- **Program_Error** : qui apparaît si nous tentons de violer d'une certaine façon la structure de contrôle, comme lorsqu'on atteint un ***end*** dans une fonction, lorsqu l'on viole les règles d'accessibilité ou qu'on appelle un sous-programme dont le corps n'a pas encore été élaboré;
- **Storage_Error** : qui apparaîtra en cas de dépassement mémoire, par exemple, en cas d'appel de la fonction récursive Factorielle avec un paramètre trop grand;
- **Tasking_Error** : qui est levée si une tâche fait appel à une tâche avortée ou est le résultat d'un échec de communication entre 2 tâches

CADRE DE TRAITE D'EXCEPTION

- La syntaxe d'un bloc avec une partie traite_d'exception est la suivante:

```
begin
  Suite_D_Instructions
exception
  Traite_Exception
       {Traite_Exception}
end;

Traite_Exception ::=
  when Choix_D_Exception{ | Choix_D_Exception} =>
Suite_D_Instruction

  Choix_D_Exception ::= Nom_D_Exception | [paramètre_de_choix : ] others
```

CADRE DE TRAITE D'EXCEPTION (Suite)

- Exemple:

```
begin
  -- suite d'instructions
exception
  when Singularité | Numeric_Error =>
    Put ("La matrice est singulière");

 when Storage_Error =>
    Put (« Mémoire insuffisante");

  when others =>
    Put ("Erreur fatale");
    raise Erreur;
end;
```

BLOCS

- Ada permet des déclarations locales et le traitement d'exception dans une suite d'instructions par l'intermédiaire des instructions de bloc.
- La syntaxe est:

```
instruction_bloc ::=
        [identificateur_de_bloc:]
        [declare
                liste_de_déclaration]
        begin
                suite_d_instructions
        exception
                traite_d_exception
                {traite_d_exception}
        end;
```

- Les blocs sont utilisés pour optimiser un programme et pour traiter localement les exceptions.

TRAITE D'EXCEPTION (Suite)

```
function Demain ( aujourdhui : in T_Jour) return T_Jour is
begin
    return T_Jour'succ(aujourdhui);
exception
    when Constraint_Error =>
        return T_Jour'first;
end Demain ;
```

DÉCLARATION D'EXCEPTIONS

- Les exceptions peuvent être déclarées dans la partie déclarative d'un module.
- La syntaxe est:

  nom_d_exception : ***exception***;

DÉCLENCHEMENT D'EXCEPTIONS

- Une exception est déclenchée par l'énoncé ***raise***:

  raise *[nom_d_exception] [**with** littéral_chaîne];*

- Exemple :

  raise; *-- lever l'exception courante*
  raise *Une_Exception;*
  raise *Une_Autre_Exception **with** "Un message relié à l'exception";*

- La vérification des exceptions prédéfinies peut être supprimée à l'exécution par l'utilisation du ***pragma*** SUPPRESS.

  pragma Suppress (Index_Check, Service);
  pragma Suppress (Range_Check);

DÉCLENCHEMENT D'EXCEPTIONS *pragma* SUPPRESS

- La syntaxe du ***pragma*** *Suppress* est:

  pragma *Suppress (identificateur[, ON => nom]);*

  nom représente un nom d'objet ou de type

Identificateur	Exception
Access_Check Discriminant_Check Index_Check Length_Check Range_Check	Constraint_Error
Division_Check Overflow_Check	Numeric_Error
Elaboration_Check	Program_Error
Storage_Check	Storage_Error

Package Ada.Exceptions

```
with Ada.Streams;
with System.Standard_Library;
package Ada.Exceptions is
  type Exception_Id is private;
  Null_Id : constant Exception_Id;

  type Exception_Occurrence is limited private;
  type Exception_Occurrence_Access is access all Exception_Occurrence;

  Null_Occurrence : constant Exception_Occurrence;

  function Exception_Identity (X : Exception_Occurrence) return Exception_Id;
  function Exception_Name   (X : Exception_Occurrence) return String;
  function Exception_Name   (X : Exception_Id) return String;

  procedure Raise_Exception        (E : in Exception_Id; Message : in String := "");
  function    Exception_Message (X : Exception_Occurrence) return String;
  procedure Reraise_Occurrence (X : Exception_Occurrence);

  function Exception_Information (X : Exception_Occurrence) return String;

  procedure Save_Occurrence     (Target :   out Exception_Occurrence;
                                 Source : in    Exception_Occurrence);

  function Save_Occurrence      (Source : in Exception_Occurrence) return
     Exception_Occurrence_Access;
```

Package Ada.Exceptions

```
private
   package SSL renames System.Standard_Library;

   type Exception_Id is access all SSL.Exception_Data;
   Null_Id : constant Exception_Id := null;

   subtype Nat is Natural range 0 .. SSL.Exception_Message_Buffer'Last;
   type Exception_Occurrence  (Max_Length : Nat := SSL.Exception_Msg_Max)  is limited record
      Id                  : Exception_Id;
      Msg_Length : Natural;
      Msg                 : String (1 .. Max_Length);
   end record;

   procedure Set_Exception_Occurrence (Occ : Exception_Occurrence_Access);
   pragma Export (C, Set_Exception_Occurrence, "__set_except_occ");

   procedure Exception_Occurrence_Read (Stream : access Ada.Streams.Root_Stream_Type'Class;
                                        Item   : out Exception_Occurrence);
   procedure Exception_Occurrence_Write (Stream : access Ada.Streams.Root_Stream_Type'Class;
                                         Item   : in Exception_Occurrence);
   for Exception_Occurrence'Read  use Exception_Occurrence_Read;
   for Exception_Occurrence'Write use Exception_Occurrence_Write;
   function Exception_Occurrence_Access_Input
                                    (Stream : access Ada.Streams.Root_Stream_Type'Class)
                                        return   Exception_Occurrence_Access;
   for Exception_Occurrence_Access'Input use Exception_Occurrence_Access_Input;

   Null_Occurrence : constant Exception_Occurrence := (  Max_Length    => 0,
                                                         Id                => Null_Id,
                                                         Msg_Length  => 0,
                                                        Msg                => "");
   function Exception_Name_Simple (X : Exception_Occurrence) return String;

end Ada.Exceptions;
```

Package Ada.Exceptions (Exemple)

```
with Ada.Exceptions;
with Ada.Text_IO;
procedure Main is
   Index_Error    : exception;
   Index          : Integer   := 234;
   Maximum        : constant  := 100;
begin
   declare
      La_Plus_Récente_Occurrence: Ada.Exceptions.Exception_Occurrence;
      Pointeur_Occurrence   : Ada.Exceptions.Exception_Occurrence_Access;
   begin
      if Index > Maximum then
         Ada.Exceptions.Raise_Exception ( Index_Error'Identity,
            "Index" & Integer'Image ( Index ) & " dépasse le maximum " &
            Integer'Image ( Maximum ));
      end if;
      exception
         when E : Ada.Text_IO.Data_Error | Constraint_Error  =>
            Pointeur_Occurrence := Ada.Exceptions.Save_Occurrence ( E );
            raise;
         when  E : others =>
            Ada.Text_IO.Put_Line ( "Oh!, non" );
            Ada.Text_IO.Put ( Ada.Exceptions.Exception_Name ( E ));
            Ada.Text_IO.Put ( "levée dans la procédure Main et propagée" );
            Ada.Text_IO.New_Line;
            Ada.Exceptions.Save_Occurrence ( La_Plus_Récente_Occurrence, E );
            Ada.Exceptions.Reraise_Occurrence ( E );
   end;
end Main;
```

Pragmas associés aux exceptions

- **Pragmas**
  - **Vérifier si une condition est satisfaite, dans le cas contraire, lever une exception**

    pragma **Assert (Expression_Booléenne, Chaîne_Caractères);**
  - **Contrôler le comportement du *pragma* Assert**

    pragma **Assertion_Policy (Policy_Identifier); -- Policy_Identifier = Check | Ignore**
  - **Contrôler les vérifications pour les exceptions du système**

    pragma **Suppress (***identificateur[, ON => nom]***); -- Déjà vu**

Paquetage Ada.Assertions

- Le **pragma** Assertion est supporté par le paquetage Ada.Assertions

Ada 2005

```
package Ada.Assertions is
      pragma Pure(Assertions);

   Assertion_Error : exception;

   procedure Assert(Check : in Boolean);
     procedure Assert(Check : in Boolean; Message : in String);

end Ada.Assertions;
```

Journal d'exception

- Exemple d'un journal d'exceptions :

```
with Ada.Exceptions;
with Reflection;
package body cFonctions is
   procedure Fonctions_1 is
      package m is new Reflection;
   begin
      …
   exception
      when Occurrence : others =>
         m.LogException ( Occurrence );
         Ada.Exceptions.Reraise_Occurence (Occurrence );
end cFonctions;
```

Journal d'exception

- Exemple d'un journal d'exceptions :

```
with Mixed_Case;
with Ada.Strings.Fixed;
with Ada.Tags;
with Ada.Exceptions;
 generic
§ package Reflection is
§   Unit_Name : constant String;
§    procedure LogExceptions ( Occurrence : in Ada.Exceptions.Exception_Occurrence );
§
§    function Get_Unit_Name return String;
§private -- Reflection
§   type T is abstract tagged null record;
§   Name : constant String := Ada.Tags.Expanded_Name (T'Tag);
§   Dot  : constant String := ".";
§   Direction : constant Ada.Strings.Direction := Ada.Strings.Backward;
§   Dot_Location : constant Natural :=
§    Ada.Strings.Fixed.Index (Source     => Name (Name'First .. Ada.Strings.Fixed.Index
                             (Source      => Name,
§                             Pattern     => Dot,
§                             Going       => Direction) - 1),
§                                 Pattern    => Dot,
§                                 Going      => Direction);
§-- 1st Index finds dot before type name
§-- 2nd Index finds dot before instantiation name
§   Unit_Name : constant String := Mixed_Case( Name (Name'First .. Dot_Location - 1) );
§   -- Trim off type name and instantiation name
 end Reflection;
```

CFONCTIONS.FONCTION_1.
M.T
Cfonctions.Fonction_1

Journal d'exception

- Exemple d'un journal d'exceptions :

```
with Ada.Text_IO;
   package body Reflection is
      procedure LogExceptions ( Occ : in Ada.Exceptions.Exception_Occurrence ) is
         Where : String := Get_Unit_Name;
      begin
         Ada.Text_IO.Put_Line ("Exception dans " & Where & " " &
                               Ada.Exceptions.Exception_Name(Occ)) ;
         Ada.Text_IO.Put_Line ("Exception dans " & Where & " "
                               & Ada.Exceptions.Exception_Information(Occ)) ;
         Ada.Text_IO.Put_Line ("Exception dans " & Where  & " "
                               & Ada.Exceptions.Exception_Message(Occ)) ;
      end LogExceptions;

      function Get_Unit_Name return String is
      begin
         return Unit_Name;
      end Get_Unit_Name;
   begin
      Ada.Text_IO.Put_Line (Get_Unit_Name);
   end Reflection;
```

Journal d'exception

- Exemple d'un journal d'exceptions :

```
with Ada.Characters.Handling;
   function  Mixed_Case ( Name : in  String ) return String is
   NameTmp      : string := Name;
   Dot          : constant Character := '.';
   UnderLine    : constant Character := '_';
   begin
   for c in Name'First + 1 .. Name'Last loop
      if Name ( Integer'Pred(c)) /= Dot and
           Name ( Integer'Pred(c)) /= UnderLine then
           NameTmp(c) := Ada.Characters.Handling.To_Lower(Name(c));
      end if;
   end loop;
   return NameTmp;
   end Mixed_Case;
```