

# Exceptions

## Corrigé

### Objectifs

- Comprendre et savoir utiliser les exceptions
- Comprendre la différence entre programmation offensive et programmation défensive

Exercice 1 : Propagation et traitement d'exceptions .....	1
Exercice 2 : Utilisation des exceptions .....	11
Exercice 3 : Module Piles et programmation défensive .....	14

### Exercice 1 : Propagation et traitement d'exceptions

Dans cet exercice, plusieurs programmes sont fournis, ils correspondent à différents cas de mise en œuvre des exceptions. Pour chaque programme, il est demandé de préciser les éléments affichés et de décrire le comportement du programme.

1. Indiquer ce qui est affiché lorsque le programme suivant est exécuté alors que l'utilisateur saisit le caractère e.

Listing 1 – Le programme Exemple\_1

```
1  with text_io;           use text_io;
2  with ada.integer_text_io; use ada.integer_text_io;
3
4  procedure Exemple_1 is
5
6  -- spécification volontairement omise !
7  procedure Lire_Entier (FValeur : out Integer) is
8  begin
9      Put_Line ("Début lire_entier");
10     Get (FValeur);
11     Put_Line ("Fin lire_entier");
12 exception
13     when Data_Error =>
14         Put_Line ("Erreur de saisie dans lire_entier");
15 end Lire_Entier;
16
17 ----- Programme principal -----
18     Nb : Integer; -- le nombre à lire
```

```

19  begin
20      Put_Line ("Début instructions du programme Exemple 1");
21      Lire_Entier (Nb);
22      Put_Line ("Fin instructions du programme Exemple 1");
23  end Exemple_1;
    
```

**Solution :** Voici le résultat de l'exécution :

```

1  Début instructions du programme Exemple 1
2  Début lire_entier
3  Erreur de saisie dans lire_entier
4  Fin instructions du programme Exemple 1
    
```

Les instructions exécutées sont :

```

1  19 -- début bloc
2      20
3      21
4      7
5      8 -- début bloc
6          9
7          10 l'utilisateur saisit 'e' --> levée de l'exception Data_Error
8              => Interruption de l'exécution du bloc 8-12 et propagation de l'exception
9              => l'instruction 11 n'est donc pas exécutée
10             => Gestionnaire d'exception en ligne 12
11             => qui sait traiter Data_Error (ligne 13)
12             => l'exécution reprend en l. 14
13          14
14      15 -- fin bloc
15      22
16  23 -- fin bloc
    
```

2. Indiquer ce qui est affiché lorsque le programme suivant est exécuté alors que l'utilisateur saisit le caractère e.

Listing 2 – Le programme Exemple\_2

```

1  with text_io;           use text_io;
2  with ada.integer_text_io; use ada.integer_text_io;
3
4  procedure Exemple_2 is
5
6  -- spécification volontairement omise !
7  procedure Lire_Entier (FValeur : out Integer) is
8  begin
9      Put_Line ("Début de lire_entier");
10     Get (FValeur);
11     Put_Line ("Fin de lire_entier");
12 end lire_entier;
13
14 ----- Programme principal -----
15     Nb : Integer; -- le nombre à lire
16 begin
17     Put_Line ("Début de Exemple_2");
18     Lire_Entier (Nb);
19     Put_Line ("Fin de Exemple_2");
20 exception
21     when Data_Error =>
22         Put_Line ("Erreur de saisie");
23 end Exemple_2;
    
```

**Solution :** Voici le résultat de l'exécution :

```
1 Début de Exemple_2
2 Début de lire_entier
3 Erreur de saisie
```

Les instructions exécutées sont :

```
1 16
2 17
3 18
4 7
5 8
6 9
7 10 l'utilisateur saisit 'e' --> levée de l'exception Data_Error
8 => Interruption de l'exécution du bloc 8-12 et propagation de l'exception
9 => l'instruction 11 n'est donc pas exécutée
10 => Pas de gestionnaire d'exception dans ce bloc
11 12 -- propagation vers bloc précédent
12 => Propagation dans le bloc précédent 16-22
13 => L'instruction 19 n'est donc pas exécutée
14 => Gestionnaire d'exception dans ce bloc l. 20
15 => qui sait traiter Data_Error (l. 21)
16 => l'exécution reprend en l. 22
17 22
18 23
```

3. Indiquer ce qui est affiché lorsque le programme suivant est exécuté alors que l'utilisateur saisit le caractère e.

Listing 3 – Le programme Exemple\_3

```

1  with text_io;           use text_io;
2  with ada.integer_text_io; use ada.integer_text_io;
3
4  procedure Exemple_3 is
5
6  -- spécification volontairement omise !
7  procedure lire_entier (FValeur : out Integer) is
8
9  -- spécification volontairement omise !
10 procedure Lire_Interne (FValeur_Interne : out Integer) is
11 begin
12     Put_Line ("Début de Lire_Interne");
13     Get (FValeur_Interne);
14     Put_Line ("Fin de Lire_Interne");
15 end lire_interne;
16
17 begin
18     Put_Line ("Début de lire_entier");
19     lire_interne (FValeur);
20     Put_Line ("Fin de lire_entier");
21 exception
22     when Data_Error =>
23         Put_Line ("Erreur de saisie dans Lire_Entier");
24 end lire_entier;
25
26 ----- Programme principal -----
27     Nb: Integer; -- le nombre à lire
28 begin
29     Put_Line ("Début de exemple_3");
30     Lire_Entier (Nb);
31     Put_Line ("Fin de exemple_3");
32 end Exemple_3;
    
```

**Solution :** Voici le résultat de l'exécution :

```

1  Début de exemple_3
2  Début de lire_entier
3  Début de Lire_Interne
4  Erreur de saisie dans Lire_Entier
5  Fin de exemple_3

```

Les instructions exécutées sont :

```

1  28
2      29
3      30
4      7
5      17
6          18
7          19
8          10
9          11
10             12
11             13 l'utilisateur saisit 'e' --> levée de l'exception Data_Error
12                 => Interruption de l'exécution du bloc 11-15 et propagation de l'exception
13                 => L'instruction 14 n'est donc pas exécutée
14                 => Pas de gestionnaire d'exception dans ce bloc
15             15 -- propagation vers le bloc précédent
16                 => Propagation dans le bloc précédent 17-24
17                 => L'instruction 20 n'est donc pas exécutée
18                 => Gestionnaire d'exception dans ce bloc l. 21
19                 => qui sait traiter Data_Error (l. 22)
20                 => l'exécution reprend en l. 23
21             23
22         24
23         31
24     32

```

4. Indiquer ce qui est affiché lorsque le programme suivant est exécuté alors que l'utilisateur saisit le caractère e.

Listing 4 – Le programme Exemple\_4

```

1  with text_io;           use text_io;
2  with ada.integer_text_io; use ada.integer_text_io;
3
4  procedure Exemple_4 is
5
6  -- spécification volontairement omise !
7  procedure Lire_Entier (FValeur : out Integer) is
8
9  -- spécification volontairement omise !
10 procedure Lire_Interne (FValeur_Interne : out Integer) is
11 begin
12     Put_Line ("Début de Lire_Interne");
13     Get (FValeur_Interne);
14     Put_Line ("Fin de Lire_Interne");
15 end lire_interne;
16
17 begin
18     Put_Line ("Début de lire_entier");
19     Lire_Interne (FValeur);
20     Put_Line ("Fin de lire_entier");
21 end lire_entier;
22
23 ----- Programme principal -----
24     Nb : Integer; -- le nombre à lire
25 begin
26     Put_Line ("Début de exemple_4");
27     Lire_Entier (Nb);
28     Put_Line ("Fin de exemple_4");
29 exception
30     when Data_Error =>
31         Put_Line ("Erreur de saisie");
32 end Exemple_4;
    
```

**Solution :** Voici le résultat de l'exécution :

```

1  Début de exemple_4
2  Début de lire_entier
3  Début de Lire_Interne
4  Erreur de saisie

```

Les instructions exécutées sont :

```

1  25
2      26
3      27
4      10
5      17
6          18
7          19
8          10
9          11
10             12
11             13 l'utilisateur saisit 'e' --> levée de l'exception Data_Error
12                 => Interruption de l'exécution du bloc 11-15 et propagation de l'exception
13                 => L'instruction 14 n'est donc pas exécutée
14                 => Pas de gestionnaire d'exception dans ce bloc
15             15 -- propagation vers le bloc précédent
16                 => Propagation dans le bloc précédent 17-21
17                 => L'instruction 20 n'est donc pas exécutée
18                 => Pas de gestionnaire d'exception dans ce bloc
19         21 -- propagation vers le bloc précédent
20             => Propagation dans le bloc précédent 25-32
21             => L'instruction 28 n'est donc pas exécutée
22             => Gestionnaire d'exception dans ce bloc l. 29
23             => qui sait traiter Data_Error (l. 30)
24             => l'exécution reprend en l. 31
25         31
26     32

```



5. Indiquer ce qui est affiché lorsque le programme suivant est exécuté alors que l'utilisateur saisit le caractère e puis l'entier 2.

Listing 5 – Le programme Exemple\_5

```

1  with text_io;           use text_io;
2  with ada.integer_text_io; use ada.integer_text_io;
3
4  procedure Exemple_5 is
5
6  -- spécification volontairement omise !
7  procedure Lire_Entier (FValeur : out Integer) is
8  begin
9      Put_Line ("Début lire_entier");
10     Get (FValeur);
11     Put_Line ("Fin lire_entier");
12 exception
13     when Data_Error =>
14         Put_Line ("Erreur de saisie dans lire_entier");
15         Skip_Line;
16         Lire_Entier (FValeur);
17 end Lire_Entier;
18
19 ----- Programme principal -----
20     Nb : Integer; -- le nombre à lire
21 begin
22     Put_Line ("Début de Exemple_5");
23     Lire_Entier (Nb);
24     Put_Line ("Fin de Exemple_5");
25 end Exemple_5;
    
```

**Solution :** Voici le résultat de l'exécution :

```

1  Début de Exemple_5
2  Début lire_entier
3  Erreur de saisie dans lire_entier
4  Début lire_entier
5  Fin lire_entier
6  Fin de Exemple_5

```

Les instructions exécutées sont :

```

1  21
2      22
3      23
4      7
5      8
6          9
7          10 l'utilisateur saisit 'e' --> levée de l'exception Data_Error
8              => Interruption de l'exécution du bloc 8-17 et propagation de l'exception
9              => L'instruction 11 n'est donc pas exécutée
10             => Gestionnaire d'exception dans ce bloc l. 12
11             => qui sait traiter Data_Error (l. 13)
12             => l'exécution reprend en l. 14
13          14
14          15 -- ce Skip_Line permet de vider le buffer d'entrée (supprimera 'e')
15          16
16          7
17          8
18              9
19              10 l'utilisateur saisit '2' --> valeur vaut donc 2
20              11
21          17
22      17
23      24
24  25

```

## Exercice 2 : Utilisation des exceptions

Considérons le programme du listing 6.

Listing 6 – Le programme Somme

```

1  with ada.text_io;           use ada.text_io;
2  with ada.integer_text_io;   use ada.integer_text_io;
3
4  -- Calculer la somme d'une suite d'entiers lus clavier. L'entier 0 marque la
5  -- fin de la série. Il n'en fait pas partie.
6  procedure Somme is
7      Somme : Integer;        -- la somme de valeurs lues au clavier
8      Valeur : Integer;       -- valeur lue au clavier
9  begin
10     -- calculer la somme d'une suite de valeurs entières, se terminant par 0
11     Somme := 0;
12     loop
13         Put ("Entrez une valeur entière : ");
14         Get (Valeur);
15         Somme := Somme + Valeur;
16     exit when Valeur = 0;
17     end loop;
18
19     -- afficher la somme
20     Put ("la somme vaut : ");
21     Put (Somme, 1);
22     New_Line;
23 end Somme;
```

1. Expliquer pourquoi ce programme n'est pas robuste.

**Solution :** Car l'utilisateur peut saisir autre chose qu'un entier ce qui provoquera en Ada une exception `Data_Error`.

2. Modifier le programme afin que la lecture d'une donnée de type incorrect provoque l'affichage du message « Saisie invalide » (et pas la somme).

**Solution :**

**Principe :** Il suffit de récupérer l'exception `Data_Error` dans le bloc du programme principal. Ainsi, on aura soit la somme affichée, soit le message « Saisie invalide ».

```

1  with ada.text_io;           use ada.text_io;
2  with ada.integer_text_io;   use ada.integer_text_io;
3
4  -- Calculer la somme d'une suite d'entiers lus clavier. L'entier 0 marque la
5  -- fin de la série. Il n'en fait pas partie.
6  -- Si l'utilisateur fait une erreur de saisie le message "Saisie invalide" est
7  -- affiché (et pas la somme).
8  procedure Somme_Invalide is
9      Somme : Integer;        -- la somme de valeurs lues au clavier
10     Valeur : Integer;       -- valeur lue au clavier
11 begin
```

```

12      -- calculer la somme d'une suite de valeurs entières, se terminant par 0
13      Somme := 0;
14      loop
15          Put ("Entrez une valeur entière : ");
16          Get (Valeur);
17          Somme := Somme + Valeur;
18      exit when Valeur = 0;
19      end loop;
20
21      -- afficher la somme
22      Put ("la somme vaut : ");
23      Put (Somme, 1);
24      New_Line;
25  exception
26      when Data_Error =>
27          Put_Line ("Saisie invalide");
28  end Somme_Invalide;
    
```

3. Modifier le programme pour qu'il s'arrête sur la première saisie invalide et affiche la somme en précisant avant « Attention, somme partielle ! ».

#### Solution :

**Principe :** Il faut toujours afficher la somme mais éventuellement afficher un message si une donnée est incorrecte. On ajoute donc un bloc autour du calcul de la somme pour récupérer l'exception et afficher le message.

**Remarque :** On aurait pu prendre la même stratégie qu'à la question précédente en ajoutant l'affichage de la somme dans le gestionnaire d'exception. Le problème de cette solution c'est que l'on a du code redondant : la somme est affichée à deux endroits.

```

1  with ada.text_io;           use ada.text_io;
2  with ada.integer_text_io;   use ada.integer_text_io;
3
4  -- Calculer la somme d'une suite d'entiers lus clavier. L'entier 0 marque la
5  -- fin de la série. Il n'en fait pas partie.
6  -- On s'arrête sur la première erreur de saisie en affichant "Somme partielle".
7  procedure Somme_Partielle is
8      Somme : Integer; -- la somme de valeurs lues au clavier
9      Valeur : Integer; -- valeur lue au clavier
10  begin
11      -- calculer la somme d'une suite de valeurs entières, se terminant par 0
12      begin
13          Somme := 0;
14          loop
15              Put ("Entrez une valeur entière : ");
16              Get (Valeur);
17              Somme := Somme + Valeur;
18          exit when Valeur = 0;
19          end loop;
20      exception
21          when Data_Error =>
    
```

```

22         Put_Line ("Attention, somme partielle !");
23     end;
24
25     -- afficher la somme
26     Put ("la somme vaut : ");
27     Put (Somme, 1);
28     New_Line;
29 end Somme_Partielle;
    
```

4. Modifier le programme pour qu'il ignore les saisies invalides (il affichera juste « saisie invalide... mais on continue ! ») et affiche la somme des entiers.

#### Solution :

**Principe :** Quand l'utilisateur fait une erreur de saisie, on ne veut pas interrompre la boucle de manière à pouvoir continuer les saisies. Il faut donc récupérer l'exception à l'intérieur de la boucle.

```

1  with ada.text_io;           use ada.text_io;
2  with ada.integer_text_io;   use ada.integer_text_io;
3
4  -- Calculer la somme d'une suite d'entiers lus au clavier. L'entier 0 marque la
5  -- fin de la série. Il n'en fait pas partie.
6  -- En cas d'erreur de saisie, un message est affiché et la saisie est refaite.
7  procedure Somme_Ignorer is
8      Somme : Integer;        -- la somme de valeurs lues au clavier
9      Valeur : Integer;       -- valeur lue au clavier
10 begin
11     -- calculer la somme d'une suite de valeurs entières, se terminant par 0
12     Somme := 0;
13     loop
14         begin
15             Put ("Entrez une valeur entière : ");
16             Get (Valeur);
17             Somme := Somme + Valeur;
18         exception
19             when Data_Error =>
20                 Put_Line ("Saisie invalide... mais on continue !");
21                 Skip_Line;
22                 Valeur := 1; -- Ne pas sortir de la boucle si le premier Get échoue :
23                             -- Valeur non initialisée dans ce cas donc indéterminée
24         end;
25     exit when Valeur = 0;
26     end loop;
27
28     -- afficher la somme
29     Put ("la somme vaut : ");
30     Put (Somme, 1);
31     New_Line;
32 end Somme_Ignorer;
    
```

**Remarque :** Il ne faut pas oublier le Skip\_Line pour supprimer du tampon d'entrée les caractères qui ont provoqué l'erreur précédente.

**Attention :** Il ne faut pas mettre que l’instruction `Get (Valeur)`; dans le bloc car on va alors augmenter la somme de la valeur précédemment saisie !

**Remarque :** Est-ce que le programme est bien écrit ? Plus précisément, est-ce qu’il respecte sa spécification ?

En fait, non. Imaginons une petite évolution dans l’énoncé : au lieu de se terminer par 0, la série se termine par -1 (ou par un nombre négatif). Comment faut-il faire évoluer le programme ?

Logiquement, il suffirait de modifier la condition du Répéter en remplaçant `Valeur = 0` par `Valeur = -1` (ou `Valeur < 0`).

Mais ici, on n’aura pas le bon résultat. Pourquoi ?

Car on a exécuté l’instruction `Somme := Somme + Valeur`; sur la dernière valeur lue qui, d’après la spécification, ne fait pas partie de la série.

On pourrait corriger en retranchant la dernière valeur. Mais c’est une rustine, une bidouille. Est-ce qu’on pourrait apporter une telle correction si on voulait aussi calculer la plus petite valeur saisie ?

La bonne correction est donc d’ajouter une décision avant l’ajout de `Valeur` à `Somme` : `if Valeur /= 0 then` (dans le programme initial). On constate avec le Répéter que l’on écrit donc deux fois la condition. On pourrait prendre un TantQue pour éviter la redondance de la condition mais on va alors doubler l’instruction `Get (Valeur)`, une fois avant le TantQue une fois à sa fin.

C’est souvent ce qu’on a : le Répéter duplique une condition (que l’on peut limiter en utilisant une variable booléenne) et le TantQue duplique une instruction. Il faut alors choisir la structure qui minimise la redondance !

C’est aussi ce qui justifie qu’en Ada on puisse placer le `when exit` n’importe où. Ici, on le mettant après la saisie, on ne dupliquerait ni instruction, ni condition.

### Exercice 3 : Module Piles et programmation défensive

Dans cet exercice, nous partons du module *Piles* (listing 7 pour son interface et 8 pour son implantation) qui a été écrit dans un style de programmation dite *offensive*. On souhaite le modifier pour adopter un style de programmation dite *défensive*.

#### 1. Comparer programmation défensive et programmation offensive.

**Solution :** Programmation offensive : les préconditions et postconditions expriment les obligations de l’appelant et de l’appelé. C’est l’appelant qui doit vérifier la précondition du sous-programme appelé. C’est l’obligation de l’appelant. Le sous-programme appelé part du principe que la précondition est vraie. C’est son bénéfice. De manière symétrique, la postcondition est l’obligation de l’appelé et le bénéfice de l’appelant.

La programmation offensive suppose que l’appelé peut faire confiance à l’appelant. Si on n’est pas dans ce contexte, on ne peut pas faire de programmation offensive.

La programmation défensive consiste à ne pas faire confiance à l’appelant et, donc, à prévoir tous les cas anormaux. Pour signaler, un cas anormal, il est généralement conseillé de lever une

Listing 7 – L'interface du module *Piles*

```

1  -- Spécification du module Piles.
2
3  generic
4      Capacite : Integer;    -- Nombre maximal d'éléments qu'une pile peut contenir
5      type T_Element is private; -- Type des éléments de la pile
6
7  package Piles is
8
9      type T_Pile is private;
10
11     -- Initialiser une pile. La pile est vide.
12     procedure Initialiser (Pile : out T_Pile) with
13         Post => Est_Vide (Pile);
14
15     -- Est-ce que la pile est vide ?
16     function Est_Vide (Pile : in T_Pile) return Boolean;
17
18     -- Est-ce que la pile est pleine ?
19     function Est_Pleine (Pile : in T_Pile) return Boolean;
20
21     -- L'élément en sommet de la pile.
22     function Sommet (Pile : in T_Pile) return T_Element with
23         Pre => not Est_Vide (Pile);
24
25     -- Empiler l'élément en sommet de la pile.
26     procedure Empiler (Pile : in out T_Pile; Element : in T_Element) with
27         Pre => not Est_Pleine (Pile),
28         Post => Sommet (Pile) = Element;
29
30     -- Supprimer l'élément en sommet de pile
31     procedure Depiler (Pile : in out T_Pile) with
32         Pre => not Est_Vide (Pile);
33
34 private
35
36     type T_Tab_Elements is array (1..Capacite) of T_Element;
37
38     type T_Pile is
39         record
40             Elements : T_Tab_Elements;    -- les éléments de la pile
41             Taille: Integer;                -- Nombre d'éléments dans la pile
42         end record;
43
44 end Piles;
```

Listing 8 – L’implantation du module *Piles*

```

1  -- Implantation du module Piles.
2
3  package body Piles is
4
5      procedure Initialiser (Pile : out T_Pile) is
6      begin
7          Pile.Taille := 0;
8      end Initialiser;
9
10     function Est_Vide (Pile : in T_Pile) return Boolean is
11     begin
12         return Pile.Taille = 0;
13     end Est_Vide;
14
15     function Est_Pleine (Pile : in T_Pile) return Boolean is
16     begin
17         return Pile.Taille >= Capacite;
18     end Est_Pleine;
19
20     function Sommet (Pile : in T_Pile) return T_Element is
21     begin
22         return Pile.Elements (Pile.Taille);
23     end Sommet;
24
25     procedure Empiler (Pile : in out T_Pile; Element : in T_Element) is
26     begin
27         Pile.Taille := Pile.Taille + 1;
28         Pile.Elements (Pile.Taille) := Element;
29     end Empiler;
30
31     procedure Depiler (Pile : in out T_Pile) is
32     begin
33         Pile.Taille := Pile.Taille - 1;
34     end Depiler;
35
36 end Piles;
    
```



exception. Sinon, on pourrait décider de faire un traitement par défaut (par exemple dépiler un pile vide pourrait ne rien faire).

**2. Modifier le module *Piles* pour qu'il mette en œuvre la programmation défensive.**

**Solution :** Le principe est de supprimer les préconditions des sous-programmes de l'interface du module. Elles seront donc vraies. Si la condition correspondant à la précondition supprimée n'est pas vraie, il faudra lever une exception qu'il faudra documenter dans la spécification du sous-programme.

La nouvelle interface du module pile est disponible au listing 9. Dans le corps du module, il faudra lever les exceptions correspondantes (listing 10).

Les modifications apportées sont donc les suivantes :

1. Identifier les exceptions qui peuvent se produire. Les préconditions nous les donnent. Ici on aura deux exceptions : `Pile_Vide_Error` et `Pile_Pleine_Error`.

Il est préférable de définir nos propres exception plutôt que réutiliser celles qui sont prédéfinies. Ceci facilitera l'identification de l'origine d'une exception récupérée dans un gestionnaire d'exception et donc l'écriture du traitement associé.

2. Les déclarer dans l'interface du module car l'utilisateur (autre module ou programme) doit y avoir accès car il est susceptible de les récupérer pour les traiter.
3. Supprimer les préconditions des sous-programmes de l'interface de la pile et compléter le commentaire pour dire que l'exception peut se produire en précisant sous quelles conditions.
4. Les lever effectivement dans l'implantation du module.

Dans le sous-programme, il est plus sûr de faire un test explicite pour savoir si la pile est vide (ou pleine) pour lever l'exception adéquate. Notons, que pour ce Si, on ne met pas de Sinon et on garde le code nominal du sous-programme au premier niveau (on le cache pas dans le else). Si on passe dans le Si, l'exception sera levée et le reste du corps ne sera pas exécuté.

On pourrait aussi récupérer l'exception `Constraint_Error` qui se produira si l'indice est invalide. Cependant cette solution est moins sûre car, si le code était plus long, plusieurs instructions pourraient lever cette exception bas niveau pour d'autres raisons qu'un problème de pile vide ou pleine (par exemple une bête erreur de programmation avec un mauvais indice) et on pourrait lever à tort l'exception `Pile_Vide_Error` ou `Pile_Pleine_Error`.

**Remarque :** Pour les **if** qui lèvent une exception, on n'a pas mis de **else**. L'objectif est de ne pas cacher le code intéressant du sous-programme dans ce **else** : il est au premier niveau, séparé d'une ligne vide de la levée de l'exception. L'autre intérêt est de remarquer que l'on a les mêmes 3 lignes en début de `Sommet` et `Depiler`. On pourrait les factoriser dans une procédure privée au module appelée `Verifier_Pile_Non_Vide` qui prend en paramètre une pile. Il suffirait de l'appeler en début de `Sommet` et `Depiler`.

**3. Écrire un programme qui empile une suite d'entiers strictement positifs lus au clavier. Il s'arrête dès que l'utilisateur saisit un entier négatif ou nul. Ce programme devra afficher le message « Plus de place » lorsque la capacité de la pile est atteinte et demandera alors à l'utilisateur s'il**

Listing 9 – L'interface du module *Piles* en programmation défensive

```

1  -- Spécification du module Piles.
2
3  generic
4      Capacite : Integer;    -- Nombre maximal d'éléments qu'une pile peut contenir
5      type T_Element is private; -- Type des éléments de la pile
6
7  package Piles is
8
9      type T_Pile is limited private;    --! "très privé" en Algorithmique !
10         --! Sur un type privé, on a droit à l'affectation (:=) et l'égalité (=).
11         --! On perd ces opérations avec un type "limited private" (très privé).
12
13         Pile_Vide_Error: Exception;    -- en cas d'opération sur une pile vide
14         Pile_Pleine_Error: Exception;    -- dépassement de la capacité d'une pile
15
16         -- Initilaiser une pile. La pile est vide.
17         procedure Initialiser (Pile : out T_Pile) with
18             Post => Est_Vide (Pile);
19
20         -- Est-ce que la pile est vide ?
21         function Est_Vide (Pile : in T_Pile) return Boolean;
22
23         -- Est-ce que la pile est pleine ?
24         function Est_Pleine (Pile : in T_Pile) return Boolean;
25
26         -- L'élément en sommet de la pile.
27         -- Exception : Pile_Vide_Error si la pile est vide.
28         function Sommet (Pile : in T_Pile) return T_Element;
29
30         -- Empiler l'élément en sommet de la pile.
31         procedure Empiler (Pile : in out T_Pile; Element : in T_Element) with
32             Post => Sommet (Pile) = Element;
33
34         -- Supprimer l'élément en sommet de pile
35         -- Exception : Pile_Vide_Error si la pile est vide.
36         procedure Depiler (Pile : in out T_Pile);
37
38     private
39
40         type T_Tab_Elements is array (1..Capacite) of T_Element;
41
42         type T_Pile is
43             record
44                 Elements : T_Tab_Elements;    -- les éléments de la pile
45                 Taille: Integer;    -- Nombre d'éléments dans la pile
46             end record;
47
48     end Piles;
    
```

Listing 10 – L’implantation du module *Piles* en programmation défensive

```

1  -- Implantation du module Piles.
2
3  package body Piles is
4
5      procedure Initialiser (Pile : out T_Pile) is
6      begin
7          Pile.Taille := 0;
8      end Initialiser;
9
10     function Est_Vide (Pile : in T_Pile) return Boolean is
11     begin
12         return Pile.Taille = 0;
13     end Est_Vide;
14
15     function Est_Pleine (Pile : in T_Pile) return Boolean is
16     begin
17         return Pile.Taille >= Capacite;
18     end Est_Pleine;
19
20     function Sommet (Pile : in T_Pile) return T_Element is
21     begin
22         if Est_Vide (Pile) then
23             raise Pile_Vide_Error;
24         end if;
25
26         return Pile.Elements (Pile.Taille);
27     end Sommet;
28
29     procedure Empiler (Pile : in out T_Pile; Element : in T_Element) is
30     begin
31         if Pile.Taille >= Capacite then
32             raise Pile_Pleine_Error;
33         end if;
34
35         Pile.Taille := Pile.Taille + 1;
36         Pile.Elements (Pile.Taille) := Element;
37     end Empiler;
38
39     procedure Depiler (Pile : in out T_Pile) is
40     begin
41         if Est_Vide (Pile) then
42             raise Pile_Vide_Error;
43         end if;
44
45         Pile.Taille := Pile.Taille - 1;
46     end Depiler;
47
48 end Piles;
    
```

veut continuer en lui proposant de dépiler un nombre (demandé à l'utilisateur) d'éléments pour continuer. Le programme devra être robuste.

**Solution :** Gardons les bonnes habitudes...

```

1  R0 : Saisir une pile          pile: out T_Pile, une pile d'entiers
2
3  Exemple avec une pile de capacité 5
4
5      Valeur : 1
6      Valeur : 2
7      Valeur : 3
8      Valeur : 4
9      Valeur : 5
10     Valeur : 6    ~^v~> Pile_Pleine_Error
11     La pile est pleine !
12     Nombre d'éléments à dépiler : 3
13     Valeur : 7
14     La pile contient :
15         7
16         6
17         2
18         1
19     ----- fond de pile
20
21     Autres cas à envisager :
22     - l'utilisateur demande à dépiler trop d'éléments
23     - l'utilisateur saisit autre chose qu'un entier pour la valeur
24     - l'utilisateur saisit autre chose qu'un entier pour le nombre d'éléments
25
26     R1 : Comme « Saisir une pile » ?
27     Initialiser la pile
28     Demander un entier          Valeur : out Entier
29     TantQue Valeur > 0 Faire
30         Empiler la valeur dans la pile    pile : in out, valeur : in
31         -- si la pile est pleine, on demandera à l'utilisateur
32         -- combien d'élément supprimer
33         Demander un entier          Valeur : out Entier
34     FinTQ
35
36     R2 : Comment « Saisir une valeur »
37     Saisie_OK <- FAUX
38     Répéter
39         Début
40             Écrire ("Valeur : ")
41             Lire (Valeur)
42             Saisie_OK <- TRUE
43         Exception
44             Data_Error =>
45                 Écrire ("Il faut saisir un entier.")
46                 Effacer les caractères en attente (Skip_Line)
47     Fin
    
```

```

48     JusquÀ Saisie_OK
49
50     Cette action complexe sera utile dans plusieurs contextes : à chaque fois
51     qu'on demande un entier, en particulier, quand on demandera le nombre
52     d'éléments à dépiler. On a donc intérêt à en faire un sous-programme.
53     Pour que le sous-programme fonctionne dans différent contexte, il faut
54     éventuellement le généraliser. Par exemple, ici le message "Valeur : "
55     afficher à l'utilisateur n'est pas forcément adapté. Il faut donc en faire un
56     paramètre du sous-programme. On peut l'« appeler consigne ».

1      -- Saisir de manière robuste et conviviale un entier
2      Procedure Lire_Robuste (Valeur : out Entier ; Consigne : in Chaine);

1      R2 : Comment « Empiler la valeur dans la pile »
2          Début
3              Empiler la valeur dans la pile
4          Exception
5              Pile_Pleine_Error =>
6                  ÉcrireLn ("La pile est pleine. Il faut supprimer des éléments")
7                  Supprimer plusieurs éléments
8                  Empiler la valeur
9          Fin
10
11     R3 : Comment « Supprimer plusieurs éléments »
12         Demander le nombre d'éléments à dépiler      Quantité : out Entier
13         { Quantité > 0 }
14         Dépiler les éléments
15
16     R4 : Comment « Demander le nombre d'éléments à saisir »
17         Répéter
18             Lire_Robuste (Quantité, "Nombre d'éléments à dépiler : ")
19             Si Quantité <= 0 Alors
20                 Écrire("L'entier doit être strictement positif.")
21             Sinon
22                 Rien
23             FinSi
24         JusquÀ Quantité > 0
25
26     R4 : Comment « Dépiler les éléments »
27         Début
28             Pour I de 1 à Quantité Faire
29                 Depiler (Pile)
30             FinPour
31         Exception
32             Pile_Vide_Error =>
33                 Rien;    -- rien à faire
34         Fin
    
```

Voici le programme correspondant en Ada. On ajoute un affichage après la saisie pour contrôler le contenu de la pile.

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3  with Piles;
4  with Afficher_Un_Entier;
5
6  procedure Saisir_Pile_Robuste is
7
8      -- Lire un entier de manière conviviale et robuste.
9      -- Paramètres :
10     --  Nombre : le nombre à saisir
11     --  Consigne : la consigne à afficher à l'utilisateur avant chaque saisie
12     procedure Lire_Entier_Robuste ( Nombre: out Integer ; Consigne: in String) is
13         Saisie_OK: Boolean;    -- Est-ce que la saisie a réussi ?
14     begin
15         Saisie_OK := False;
16         loop
17             begin
18                 Put (Consigne);
19                 Get (Nombre);
20                 Saisie_OK := True;
21             exception
22                 when Constraint_Error =>
23                     Put ("Il faut saisir un entier.");
24             end;
25             exit when Saisie_OK;
26         end loop;
27     end Lire_Entier_Robuste;
28
29     -- Lire un entier strictement positif de manière conviviale et robuste.
30     -- Paramètres :
31     --  Nombre : le nombre à saisir, il sera strictement positif
32     --  Consigne : la consigne à afficher à l'utilisateur avant chaque saisie
33     procedure Lire_Entier_Naturel_Strict (
34         Nombre: out Integer ;
35         Consigne: in String) with
36         Post => Nombre > 0
37     is
38         Nombre_Lu: Integer;
39         -- pour ne pas modifier Nombre avant d'avoir réussi à saisir
40         -- un entier strictement positif.
41     begin
42         loop
43             Lire_Entier_Robuste(Nombre_Lu, Consigne);
44             if Nombre_Lu <= 0 then
45                 Put_Line ("Le nombre doit être strictement positif.");
46             else
47                 null;
48             end if;
49             exit when Nombre_Lu > 0;
50         end loop;

```

```

51     Nombre := Nombre_Lu;
52 end Lire_Entier_Naturel_Strict;
53
54 package Piles_Entiers is
55     new Piles (Capacite => 5, T_Element => Integer);
56 use Piles_Entiers;
57
58 procedure Afficher is
59     new Piles_Entiers.Afficher (Afficher_Element => Afficher_Un_Entier);
60
61 Entiers : T_Pile;    -- les entiers lus au clavier
62 Valeur: Integer;    -- une valeur lue au clavier
63 Quantite: integer;  -- nombre d'éléments à dépiler
64 begin
65     -- Saisir la pile
66     Initialiser (Entiers);
67     Lire_Entier_Robuste (Valeur, "Une valeur : ");
68     while Valeur > 0 loop
69         begin
70             Empiler (Entiers, Valeur);
71         exception
72             when Pile_Pleine_Error =>
73                 Put_Line ("La pile est pleine. Il faut supprimer des éléments.");
74
75                 -- Supprimer quelques éléments
76                 Lire_Entier_Naturel_Strict (Quantite, "Nombre d'éléments à supprimer : ");
77                 -- dépiler les éléments
78                 begin
79                     for I in 1..Quantite loop
80                         Depiler (Entiers);
81                     end loop;
82                 exception
83                     when Pile_Vide_Error =>
84                         null;    -- Pas assez d'éléments dans la pile
85                 end;
86
87                 -- Empiler l'élément
88                 Empiler (Entiers, Valeur);
89             end;
90             Lire_Entier_Robuste (Valeur, "Une valeur : ");
91         end loop;
92
93     -- Afficher la pile
94     Put_Line ("La pile contient : ");
95     Afficher (Entiers);
96     New_Line;
97
98 end Saisir_Pile_Robuste;
    
```