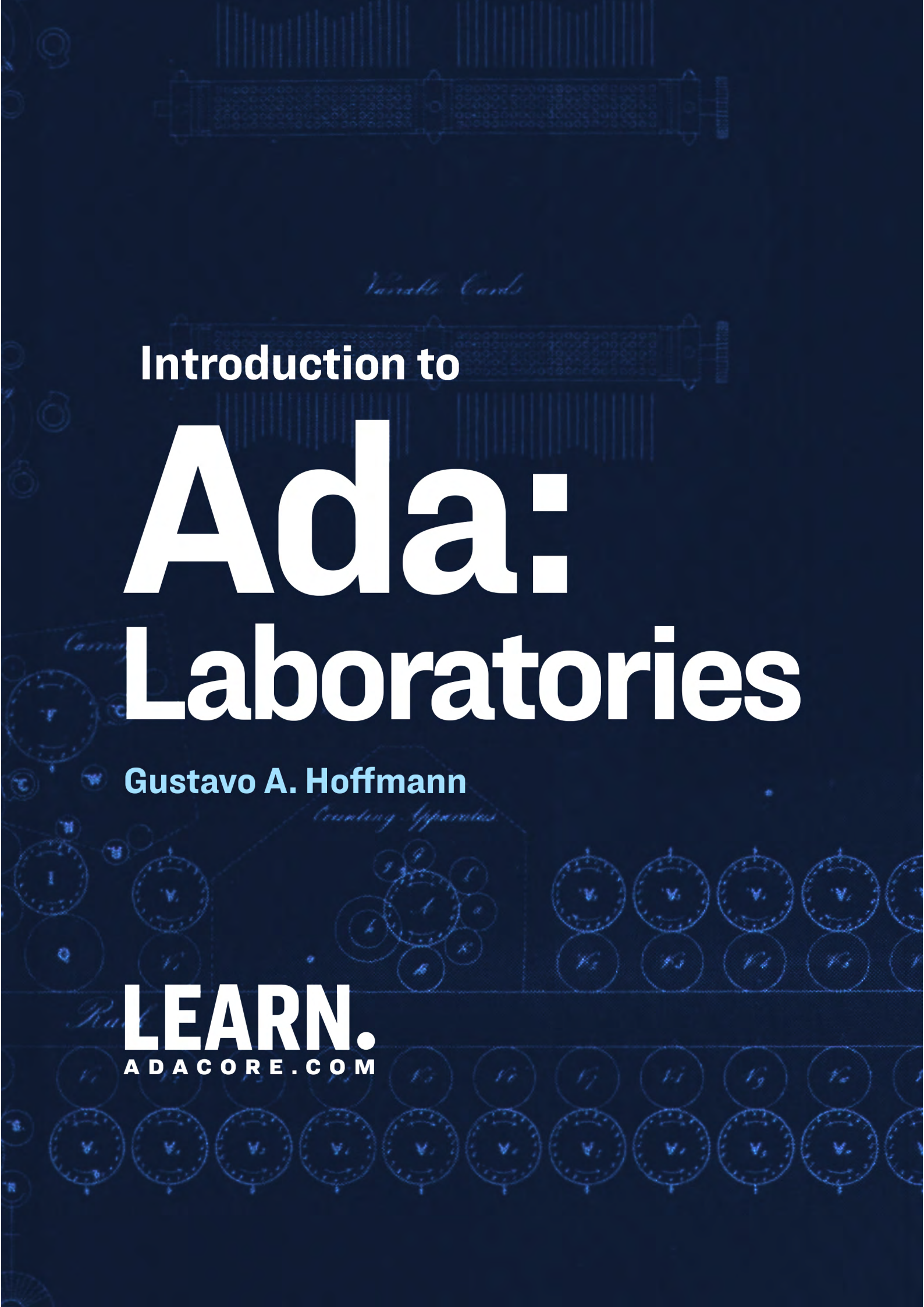




Introduction to

Ada: Laboratories

Gustavo A. Hoffmann

LEARN.
ADACORE.COM

**Introduction to Ada:
Laboratories**
Release 2025-12

Gustavo A. Hoffmann

Dec 27, 2025

CONTENTS:

1	Imperative language	3
1.1	Hello World	3
1.2	Greetings	3
1.3	Positive Or Negative	4
1.4	Numbers	5
2	Subprograms	7
2.1	Subtract procedure	7
2.2	Subtract function	8
2.3	Equality function	9
2.4	States	10
2.5	States #2	11
2.6	States #3	13
2.7	States #4	14
3	Modular Programming	17
3.1	Months	17
3.2	Operations	18
4	Strongly typed language	21
4.1	Colors	21
4.2	Integers	23
4.3	Temperatures	27
5	Records	31
5.1	Directions	31
5.2	Colors	33
5.3	Inventory	37
6	Arrays	41
6.1	Constrained Array	41
6.2	Colors: Lookup-Table	43
6.3	Unconstrained Array	46
6.4	Product info	49
6.5	String_10	52
6.6	List of Names	54
7	More About Types	59
7.1	Aggregate Initialization	59
7.2	Versioning	61
7.3	Simple todo list	63
7.4	Price list	65
8	Privacy	71
8.1	Directions	71

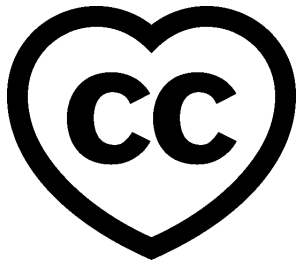
8.2	Limited Strings	73
8.3	Bonus exercise	77
8.3.1	Colors	77
8.3.2	List of Names	78
8.3.3	Price List	78
9	Generics	79
9.1	Display Array	79
9.2	Average of Array of Float	81
9.3	Average of Array of Any Type	83
9.4	Generic list	86
10	Exceptions	89
10.1	Uninitialized Value	89
10.2	Numerical Exception	90
10.3	Re-raising Exceptions	92
11	Tasking	95
11.1	Display Service	95
11.2	Event Manager	96
11.3	Generic Protected Queue	98
12	Design by contracts	101
12.1	Price Range	101
12.2	Pythagorean Theorem: Predicate	102
12.3	Pythagorean Theorem: Precondition	104
12.4	Pythagorean Theorem: Postcondition	106
12.5	Pythagorean Theorem: Type Invariant	108
12.6	Primary Color	110
13	Object-oriented programming	115
13.1	Simple type extension	115
13.2	Online Store	117
14	Standard library: Containers	123
14.1	Simple todo list	123
14.2	List of unique integers	125
15	Standard library: Dates & Times	129
15.1	Holocene calendar	129
15.2	List of events	130
16	Standard library: Strings	135
16.1	Concatenation	135
16.2	List of events	137
17	Standard library: Numerics	141
17.1	Decibel Factor	141
17.2	Root-Mean-Square	143
17.3	Rotation	145
18	Solutions	151
18.1	Imperative Language	151
18.1.1	Hello World	151
18.1.2	Greetings	151
18.1.3	Positive Or Negative	151
18.1.4	Numbers	152
18.2	Subprograms	153
18.2.1	Subtract Procedure	153
18.2.2	Subtract Function	154

18.2.3	Equality function	155
18.2.4	States	156
18.2.5	States #2	157
18.2.6	States #3	158
18.2.7	States #4	158
18.3	Modular Programming	159
18.3.1	Months	159
18.3.2	Operations	161
18.4	Strongly typed language	163
18.4.1	Colors	163
18.4.2	Integers	164
18.4.3	Temperatures	167
18.5	Records	170
18.5.1	Directions	170
18.5.2	Colors	171
18.5.3	Inventory	174
18.6	Arrays	176
18.6.1	Constrained Array	176
18.6.2	Colors: Lookup-Table	178
18.6.3	Unconstrained Array	180
18.6.4	Product info	182
18.6.5	String_10	185
18.6.6	List of Names	186
18.7	More About Types	189
18.7.1	Aggregate Initialization	189
18.7.2	Versioning	191
18.7.3	Simple todo list	192
18.7.4	Price list	194
18.8	Privacy	196
18.8.1	Directions	196
18.8.2	Limited Strings	198
18.9	Generics	201
18.9.1	Display Array	201
18.9.2	Average of Array of Float	202
18.9.3	Average of Array of Any Type	204
18.9.4	Generic list	206
18.10	Exceptions	208
18.10.1	Uninitialized Value	208
18.10.2	Numerical Exception	209
18.10.3	Re-raising Exceptions	211
18.11	Tasking	213
18.11.1	Display Service	213
18.11.2	Event Manager	214
18.11.3	Generic Protected Queue	216
18.12	Design by contracts	218
18.12.1	Price Range	218
18.12.2	Pythagorean Theorem: Predicate	219
18.12.3	Pythagorean Theorem: Precondition	221
18.12.4	Pythagorean Theorem: Postcondition	223
18.12.5	Pythagorean Theorem: Type Invariant	224
18.12.6	Primary Colors	226
18.13	Object-oriented programming	228
18.13.1	Simple type extension	228
18.13.2	Online Store	230
18.14	Standard library: Containers	233
18.14.1	Simple todo list	233
18.14.2	List of unique integers	234
18.15	Standard library: Dates & Times	236

18.15.1	Holocene calendar	236
18.15.2	List of events	237
18.16	Standard library: Strings	239
18.16.1	Concatenation	239
18.16.2	List of events	241
18.17	Standard library: Numerics	244
18.17.1	Decibel Factor	244
18.17.2	Root-Mean-Square	245
18.17.3	Rotation	248

Copyright © 2019 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)¹



These labs contain exercises for the Introduction to Ada course.

This document was written by Gustavo A. Hoffmann and reviewed by Michael Frank.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

¹ <http://creativecommons.org/licenses/by-sa/4.0>

IMPERATIVE LANGUAGE

For the exercises below (except for the first one), don't worry about the details of the Main procedure. You should just focus on implementing the application in the subprogram specified by the exercise.

1.1 Hello World

Goal: create a "Hello World!" application.

Steps:

1. Complete the Main procedure.

Requirements:

1. The application must display the message "Hello World!".

Remarks:

1. The part that you have to modify is indicated by the *-- Implement the application here!* comment in the source code.

Listing 1: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     -- Implement the application here!
6     null;
7 end Main;
```

1.2 Greetings

Goal: create an application that greets a person.

Steps:

1. Complete the Greet procedure.

Requirements:

1. Given an input string <name>, procedure Greet must display the message "Hello <name>!".

1. For example, if the name is "John", it displays the message "Hello John!".

Remarks:

1. You can use the concatenation operator (&).

2. The part that you have to modify is indicated by the *-- Implement the application here!* comment in the source code.

Listing 2: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Greet (Name : String) is
7     begin
8         -- Implement the application here!
9         null;
10    end Greet;
11
12 begin
13     if Argument_Count < 1 then
14         Put_Line ("ERROR: missing arguments! Exiting...");
15         return;
16     elsif Argument_Count > 1 then
17         Put_Line ("Ignoring additional arguments...");
18     end if;
19
20     Greet (Argument (1));
21 end Main;
```

1.3 Positive Or Negative

Goal: create an application that classifies integer numbers.

Steps:

1. Complete the Classify_Number procedure.

Requirements:

1. Given an integer number X, procedure Classify_Number must classify X as positive, negative or zero and display the result:
 1. If $X > 0$, it displays Positive.
 2. If $X < 0$, it displays Negative.
 3. If $X = 0$, it displays Zero.

Listing 3: classify_number.ads

```
1 procedure Classify_Number (X : Integer);
```

Listing 4: classify_number.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5     -- Implement the application here!
6     null;
7 end Classify_Number;
```

Listing 5: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Classify_Number;
5
6 procedure Main is
7   A : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");
14   end if;
15
16   A := Integer'Value (Argument (1));
17
18   Classify_Number (A);
19 end Main;

```

1.4 Numbers

Goal: create an application that displays numbers in a specific order.

Steps:

1. Complete the Display_Numbers procedure.

Requirements:

1. Given two integer numbers, Display_Numbers displays all numbers in the range starting with the smallest number.

Listing 6: display_numbers.ads

```

1 procedure Display_Numbers (A, B : Integer);

```

Listing 7: display_numbers.adb

```

1 procedure Display_Numbers (A, B : Integer) is
2 begin
3   -- Implement the application here!
4   null;
5 end Display_Numbers;

```

Listing 8: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Numbers;
5
6 procedure Main is
7   A, B : Integer;
8 begin
9   if Argument_Count < 2 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;

```

(continues on next page)

(continued from previous page)

```
12  elsif Argument_Count > 2 then
13      Put_Line ("Ignoring additional arguments...");
14  end if;
15
16  A := Integer'Value (Argument (1));
17  B := Integer'Value (Argument (2));
18
19  Display_Numbers (A, B);
20 end Main;
```

SUBPROGRAMS

2.1 Subtract procedure

Goal: write a procedure that subtracts two numbers.

Steps:

1. Complete the procedure Subtract.

Requirements:

1. Subtract performs the operation $A - B$.

Listing 9: subtract.ads

```
1  -- Write the correct parameters for the procedure below.
2  procedure Subtract;
```

Listing 10: subtract.adb

```
1  procedure Subtract is
2  begin
3      -- Implement the procedure here.
4      null;
5  end Subtract;
```

Listing 11: main.adb

```
1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Subtract;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Sub_10_1_Chk,
9           Sub_10_100_Chk,
10          Sub_0_5_Chk,
11          Sub_0_Minus_5_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14          Result : Integer;
15      begin
16          case TC is
17              when Sub_10_1_Chk =>
18                  Subtract (10, 1, Result);
19                  Put_Line ("Result: " & Integer'Image (Result));
20              when Sub_10_100_Chk =>
21                  Subtract (10, 100, Result);
```

(continues on next page)

(continued from previous page)

```

22     Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Subtract (0, 5, Result);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Subtract (0, -5, Result);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

2.2 Subtract function

Goal: write a function that subtracts two numbers.

Steps:

1. Rewrite the Subtract procedure from the previous exercise as a function.

Requirements:

1. Subtract performs the operation $A - B$ and returns the result.

Listing 12: subtract.ads

```

1  -- Write the correct signature for the function below.
2  -- Don't forget to replace the keyword "procedure" by "function."
3  procedure Subtract;

```

Listing 13: subtract.adb

```

1  procedure Subtract is
2  begin
3      -- Implement the function here!
4      null;
5  end Subtract;

```

Listing 14: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Subtract;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Sub_10_1_Chk,
9           Sub_10_100_Chk,
10          Sub_0_5_Chk,
11          Sub_0_Minus_5_Chk);

```

(continues on next page)

(continued from previous page)

```

12
13 procedure Check (TC : Test_Case_Index) is
14   Result : Integer;
15 begin
16   case TC is
17     when Sub_10_1_Chk =>
18       Result := Subtract (10, 1);
19       Put_Line ("Result: " & Integer'Image (Result));
20     when Sub_10_100_Chk =>
21       Result := Subtract (10, 100);
22       Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24       Result := Subtract (0, 5);
25       Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27       Result := Subtract (0, -5);
28       Put_Line ("Result: " & Integer'Image (Result));
29   end case;
30 end Check;
31
32 begin
33   if Argument_Count < 1 then
34     Put_Line ("ERROR: missing arguments! Exiting...");
35     return;
36   elsif Argument_Count > 1 then
37     Put_Line ("Ignoring additional arguments...");
38   end if;
39
40   Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

2.3 Equality function

Goal: write a function that compares two values and returns a flag.

Steps:

1. Complete the Is_Equal subprogram.

Requirements:

1. Is_Equal returns a flag as a **Boolean** value.
2. The flag must indicate whether the values are equal (flag is **True**) or not (flag is **False**).

Listing 15: is_equal.ads

```

1  -- Write the correct signature for the function below.
2  -- Don't forget to replace the keyword "procedure" by "function."
3  procedure Is_Equal;

```

Listing 16: is_equal.adb

```

1  procedure Is_Equal is
2  begin
3    -- Implement the function here!
4    null;
5  end Is_Equal;

```


Listing 17: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Is_Equal;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Equal_Chk,
9      Inequal_Chk);
10
11   procedure Check (TC : Test_Case_Index) is
12
13     procedure Display_Equal (A, B : Integer;
14                             Equal : Boolean) is
15     begin
16       Put (Integer'Image (A));
17       if Equal then
18         Put (" is equal to ");
19       else
20         Put (" isn't equal to ");
21       end if;
22       Put_Line (Integer'Image (B) & ".");
23     end Display_Equal;
24
25     Result : Boolean;
26   begin
27     case TC is
28       when Equal_Chk =>
29         for I in 0 .. 10 loop
30           Result := Is_Equal (I, I);
31           Display_Equal (I, I, Result);
32         end loop;
33       when Inequal_Chk =>
34         for I in 0 .. 10 loop
35           Result := Is_Equal (I, I - 1);
36           Display_Equal (I, I - 1, Result);
37         end loop;
38     end case;
39   end Check;
40
41 begin
42   if Argument_Count < 1 then
43     Put_Line ("ERROR: missing arguments! Exiting...");
44     return;
45   elsif Argument_Count > 1 then
46     Put_Line ("Ignoring additional arguments...");
47   end if;
48
49   Check (Test_Case_Index'Value (Argument (1)));
50 end Main;
```

2.4 States

Goal: write a procedure that displays the state of a machine.

Steps:

1. Complete the procedure `Display_State`.

Requirements:

1. The states can be set according to the following numbers:

Number	State
0	Off
1	On: Simple Processing
2	On: Advanced Processing

2. The procedure `Display_State` receives the number corresponding to a state and displays the state (indicated by the table above) as a user message.

Remarks:

1. You can use a case statement to implement this procedure.

Listing 18: `display_state.ads`

```
1 procedure Display_State (State : Integer);
```

Listing 19: `display_state.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_State (State : Integer) is
4 begin
5     null;
6 end Display_State;
```

Listing 20: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_State;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Display_State (State);
19 end Main;
```

2.5 States #2

Goal: write a function that returns the state of a machine.

Steps:

1. Implement the function `Get_State`.

Requirements:

1. Implement same state machine as in the previous exercise.
2. Function `Get_State` must return the state as a string.

Remarks:

1. You can implement a function returning a string by simply using quotes in a return statement. For example:

Listing 21: `get_hello.ads`

```
1 function Get_Hello return String;
```

Listing 22: `get_hello.adb`

```
1 function Get_Hello return String is
2 begin
3   return "Hello";
4 end Get_Hello;
```

Listing 23: `main.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Get_Hello;
3
4 procedure Main is
5   S : constant String := Get_Hello;
6 begin
7   Put_Line (S);
8 end Main;
```

2. You can reuse your previous implementation and replace it by a case expression.
 1. For values that do not correspond to a state, you can simply return an empty string (`""`).

Listing 24: `get_state.ads`

```
1 function Get_State (State : Integer) return String;
```

Listing 25: `get_state.adb`

```
1 function Get_State (State : Integer) return String is
2 begin
3   return "";
4 end Get_State;
```

Listing 26: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Get_State;
5
6 procedure Main is
7   State : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");
```

(continues on next page)

(continued from previous page)

```

14   end if;
15
16   State := Integer'Value (Argument (1));
17
18   Put_Line (Get_State (State));
19 end Main;

```

2.6 States #3

Goal: implement an on/off indicator for a state machine.

Steps:

1. Implement the function `Is_On`.
2. Implement the procedure `Display_On_Off`.

Requirements:

1. Implement same state machine as in the previous exercise.
2. Function `Is_On` returns:
 - **True** if the machine is on;
 - otherwise, it returns **False**.
3. Procedure `Display_On_Off` displays the message
 - "On" if the machine is on, or
 - "Off" otherwise.
4. `Is_On` must be called in the implementation of `Display_On_Off`.

Remarks:

1. You can implement both subprograms using if expressions.

Listing 27: `is_on.ads`

```

1 function Is_On (State : Integer) return Boolean;

```

Listing 28: `is_on.adb`

```

1 function Is_On (State : Integer) return Boolean is
2 begin
3   return False;
4 end Is_On;

```

Listing 29: `display_on_off.ads`

```

1 procedure Display_On_Off (State : Integer);

```

Listing 30: `display_on_off.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Is_On;
3
4 procedure Display_On_Off (State : Integer) is
5 begin
6   Put_Line ("");
7 end Display_On_Off;

```

Listing 31: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Display_On_Off;
5 with Is_On;
6
7 procedure Main is
8   State : Integer;
9 begin
10  if Argument_Count < 1 then
11    Put_Line ("ERROR: missing arguments! Exiting...");
12    return;
13  elsif Argument_Count > 1 then
14    Put_Line ("Ignoring additional arguments...");
15  end if;
16
17  State := Integer'Value (Argument (1));
18
19  Display_On_Off (State);
20  Put_Line (Boolean'Image (Is_On (State)));
21 end Main;
```

2.7 States #4

Goal: implement a procedure to update the state of a machine.

Steps:

1. Implement the procedure `Set_Next`.

Requirements:

1. Implement the same state machine as in the previous exercise.
2. Procedure `Set_Next` updates the machine's state with the next one in a *circular* manner:
 - In general, after a call to `Set_Next` for an integer variable `N` (`Set_Next (N)`), the new value of `N` must be the next number for that variable `N` (i.e., `N := N'Old + 1`).
 - In other words, a call to `Set_Next (N)` has the same effect as `N := N + 1`.
 - For example, after the statements `N := 1; Set_Next (N);`, we have that `N = 2`.
 - However, if the state is the last valid one for the machine (which, for this exercise, it's 2), the next state must be the first valid one (in this case: 0).
 - In other words, for `N := 2; Set_Next (N);`, we have that `N = 0`.

Remarks:

1. You can use an if expression to implement `Set_Next`.

Listing 32: set_next.ads

```
1 procedure Set_Next (State : in out Integer);
```

Listing 33: set_next.adb

```
1 procedure Set_Next (State : in out Integer) is
2 begin
3     null;
4 end Set_Next;
```

Listing 34: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Set_Next;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Set_Next (State);
19     Put_Line (Integer'Image (State));
20 end Main;
```


MODULAR PROGRAMMING

3.1 Months

Goal: create a package to display the months of the year.

Steps:

1. Convert the Months procedure below to a package.
2. Create the specification and body of the Months package.

Requirements:

1. Months must contain the declaration of strings for each month of the year, which are stored in three-character constants based on the month's name.
 - For example, the string "January" is stored in the constant Jan. These strings are then used by the Display_Months procedure, which is also part of the Months package.

Remarks:

1. The goal of this exercise is to create the Months package.
 1. In the code below, Months is declared as a procedure.
 - Therefore, we need to *convert* it into a real package.
 2. You have to modify the procedure declaration and implementation in the code below, so that it becomes a package specification and a package body.

Listing 35: months.ads

```
1  -- Create specification for Months package, which includes
2  -- the declaration of the Display_Months procedure.
3  --
4  procedure Months;
```

Listing 36: months.adb

```
1  -- Create body of Months package, which includes
2  -- the implementation of the Display_Months procedure.
3  --
4  procedure Months is
5
6      procedure Display_Months is
7      begin
8          Put_Line ("Months:");
9          Put_Line ("- " & Jan);
10         Put_Line ("- " & Feb);
11         Put_Line ("- " & Mar);
12         Put_Line ("- " & Apr);
```

(continues on next page)

(continued from previous page)

```
13     Put_Line ("- " & May);
14     Put_Line ("- " & Jun);
15     Put_Line ("- " & Jul);
16     Put_Line ("- " & Aug);
17     Put_Line ("- " & Sep);
18     Put_Line ("- " & Oct);
19     Put_Line ("- " & Nov);
20     Put_Line ("- " & Dec);
21 end Display_Months;
22
23 begin
24     null;
25 end Months;
```

Listing 37: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Months;            use Months;
5
6  procedure Main is
7
8      type Test_Case_Index is
9          (Months_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Months_Chk =>
15                 Display_Months;
16         end case;
17     end Check;
18
19 begin
20     if Argument_Count < 1 then
21         Put_Line ("ERROR: missing arguments! Exiting...");
22         return;
23     elsif Argument_Count > 1 then
24         Put_Line ("Ignoring additional arguments...");
25     end if;
26
27     Check (Test_Case_Index'Value (Argument (1)));
28 end Main;
```

3.2 Operations

Goal: create a package to perform basic mathematical operations.

Steps:

1. Implement the Operations package.
 1. Declare and implement the Add function.
 2. Declare and implement the Subtract function.
 3. Declare and implement the Multiply: function.
 4. Declare and implement the Divide function.
2. Implement the Operations.Test package

1. Declare and implement the Display procedure.

Requirements:

1. Package Operations contains functions for each of the four basic mathematical operations for parameters of **Integer** type:
 1. Function Add performs the addition of A and B and returns the result;
 2. Function Subtract performs the subtraction of A and B and returns the result;
 3. Function Multiply performs the multiplication of A and B and returns the result;
 4. Function Divide performs the division of A and B and returns the result.
2. Package Operations.Test contains the test environment:
 1. Procedure Display must use the functions from the parent (Operations) package as indicated by the template in the code below.

Listing 38: operations.ads

```

1 package Operations is
2
3   -- Create specification for Operations package, including the
4   -- declaration of the functions mentioned above.
5   --
6
7 end Operations;
```

Listing 39: operations.adb

```

1 package body Operations is
2
3   -- Create body of Operations package.
4   --
5
6 end Operations;
```

Listing 40: operations-test.ads

```

1 package Operations.Test is
2
3   -- Create specification for Operations package, including the
4   -- declaration of the Display procedure:
5   --
6   -- procedure Display (A, B : Integer);
7   --
8
9 end Operations.Test;
```

Listing 41: operations-test.adb

```

1 package body Operations.Test is
2
3   -- Implement body of Operations.Test package.
4   --
5
6   procedure Display (A, B : Integer) is
7     A_Str : constant String := Integer'Image (A);
8     B_Str : constant String := Integer'Image (B);
9   begin
10    Put_Line ("Operations:");
11    Put_Line (A_Str & " + " & B_Str & " = ")
```

(continues on next page)

(continued from previous page)

```

12         & Integer'Image (Add (A, B))
13         & ","");
14     -- Use the line above as a template and add the rest of the
15     -- implementation for Subtract, Multiply and Divide.
16 end Display;
17
18 end Operations.Test;
```

Listing 42: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Operations;
5  with Operations.Test; use Operations.Test;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Operations_Chk,
11          Operations_Display_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when Operations_Chk =>
17                 Put_Line ("Add (100, 2) = "
18                     & Integer'Image (Operations.Add (100, 2)));
19                 Put_Line ("Subtract (100, 2) = "
20                     & Integer'Image (Operations.Subtract (100, 2)));
21                 Put_Line ("Multiply (100, 2) = "
22                     & Integer'Image (Operations.Multiply (100, 2)));
23                 Put_Line ("Divide (100, 2) = "
24                     & Integer'Image (Operations.Divide (100, 2)));
25             when Operations_Display_Chk =>
26                 Display (10, 5);
27                 Display ( 1, 2);
28         end case;
29     end Check;
30
31 begin
32     if Argument_Count < 1 then
33         Put_Line ("ERROR: missing arguments! Exiting...");
34         return;
35     elsif Argument_Count > 1 then
36         Put_Line ("Ignoring additional arguments...");
37     end if;
38
39     Check (Test_Case_Index'Value (Argument (1)));
40 end Main;
```

STRONGLY TYPED LANGUAGE

4.1 Colors

Goal: create a package to represent HTML colors in hexadecimal form and its corresponding names.

Steps:

1. Implement the `Color_Types` package.
 1. Declare the `HTML_Color` enumeration.
 2. Declare the `Basic_HTML_Color` enumeration.
 3. Implement the `To_Integer` function.
 4. Implement the `To_HTML_Color` function.

Requirements:

1. Enumeration `HTML_Color` has the following colors:
 - Salmon
 - Firebrick
 - Red
 - Darkred
 - Lime
 - Forestgreen
 - Green
 - Darkgreen
 - Blue
 - Mediumblue
 - Darkblue
2. Enumeration `Basic_HTML_Color` has the following colors: Red, Green, Blue.
3. Function `To_Integer` converts from the `HTML_Color` type to the HTML color code — as integer values in hexadecimal notation.
 - You can find the HTML color codes in the table below.
4. Function `To_HTML_Color` converts from `Basic_HTML_Color` to `HTML_Color`.
5. This is the table to convert from an HTML color to a HTML color code in hexadecimal notation:

Color	HTML color code (hexa)
Salmon	#FA8072
Firebrick	#B22222
Red	#FF0000
Darkred	#8B0000
Lime	#00FF00
Forestgreen	#228B22
Green	#008000
Darkgreen	#006400
Blue	#0000FF
Mediumblue	#0000CD
Darkblue	#00008B

Remarks:

1. In order to express the hexadecimal values above in Ada, use the following syntax: `16#<hex_value>#` (e.g.: `16#FFFFFF#`).
2. For function `To_Integer`, you may use a **case** for this.

Listing 43: color_types.ads

```
1 package Color_Types is
2
3   -- Include type declaration for HTML_Color!
4   --
5   -- type HTML_Color is [...]
6   --
7
8   -- Include function declaration for:
9   -- function To_Integer (C : HTML_Color) return Integer;
10
11  -- Include type declaration for Basic_HTML_Color!
12  --
13  -- type Basic_HTML_Color is [...]
14  --
15
16  -- Include function declaration for:
17  -- - Basic_HTML_Color => HTML_Color
18  --
19  -- function To_HTML_Color [...];
20  --
21 end Color_Types;
```

Listing 44: color_types.adb

```
1 package body Color_Types is
2
3   -- Implement the conversion from HTML_Color to Integer here!
4   --
5   -- function To_Integer (C : HTML_Color) return Integer is
6   -- begin
7   --   -- Hint: use 'case' for the HTML colors;
8   --   --       use 16#...# for the hexadecimal values.
9   -- end To_Integer;
10
11  -- Implement the conversion from Basic_HTML_Color to HTML_Color here!
12  --
13  -- function To_HTML_Color [...] is
```

(continues on next page)

(continued from previous page)

```

14  --
15  end Color_Types;

```

Listing 45: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Integer_Text_IO;
4
5  with Color_Types; use Color_Types;
6
7  procedure Main is
8      type Test_Case_Index is
9          (HTML_Color_Range,
10           HTML_Color_To_Integer,
11           Basic_HTML_Color_To_HTML_Color);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when HTML_Color_Range =>
17                 for I in HTML_Color'Range loop
18                     Put_Line (HTML_Color'Image (I));
19                 end loop;
20             when HTML_Color_To_Integer =>
21                 for I in HTML_Color'Range loop
22                     Ada.Integer_Text_IO.Put (Item => To_Integer (I),
23                                             Width => 6,
24                                             Base  => 16);
25                     New_Line;
26                 end loop;
27             when Basic_HTML_Color_To_HTML_Color =>
28                 for I in Basic_HTML_Color'Range loop
29                     Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30                 end loop;
31             end case;
32     end Check;
33
34     begin
35         if Argument_Count < 1 then
36             Put_Line ("ERROR: missing arguments! Exiting...");
37             return;
38         elsif Argument_Count > 1 then
39             Put_Line ("Ignoring additional arguments...");
40         end if;
41
42         Check (Test_Case_Index'Value (Argument (1)));
43     end Main;

```

4.2 Integers

Goal: implement a package with various integer types.

Steps:

1. Implement the Int_Types package.
 1. Declare the integer type I_100.
 2. Declare the modular type U_100.

3. Implement the `To_I_100` function to convert from the `U_100` type.
4. Implement the `To_U_100` function to convert from the `I_100` type.
5. Declare the derived type `D_50`.
6. Declare the subtype `S_50`.
7. Implement the `To_D_50` function to convert from the `I_100` type.
8. Implement the `To_S_50` function to convert from the `I_100` type.
9. Implement the `To_I_100` function to convert from the `D_50` type.

Requirements:

1. Types `I_100` and `U_100` have values between 0 and 100.
 1. Type `I_100` is an integer type.
 2. Type `U_100` is a modular type.
2. Function `To_I_100` converts from the `U_100` type to the `I_100` type.
3. Function `To_U_100` converts from the `I_100` type to the `U_100` type.
4. Types `D_50` and `S_50` have values between 10 and 50 and use `I_100` as a base type.
 1. `D_50` is a derived type.
 2. `S_50` is a subtype.
5. Function `To_D_50` converts from the `I_100` type to the `D_50` type.
6. Function `To_S_50` converts from the `I_100` type to the `S_50` type.
7. Functions `To_D_50` and `To_S_50` saturate the input values if they are out of range.
 - If the input is less than 10 the output should be 10.
 - If the input is greater than 50 the output should be 50.
8. Function `To_I_100` converts from the `D_50` type to the `I_100` type.

Remarks:

1. For the implementation of functions `To_D_50` and `To_S_50`, you may use the type attributes `D_50'First` and `D_50'Last`:
 1. `D_50'First` indicates the minimum value of the `D_50` type.
 2. `D_50'Last` indicates the maximum value of the `D_50` type.
 3. The same attributes are available for the `S_50` type (`S_50'First` and `S_50'Last`).
2. We could have implemented a function `To_I_100` as well to convert from `S_50` to `I_100`. However, we skip this here because explicit conversions are not needed for subtypes.

Listing 46: `int_types.ads`

```
1 package Int_Types is
2
3   -- Include type declarations for I_100 and U_100!
4   --
5   type I_100 is [...];
6   type U_100 is [...];
7   --
8
9   function To_I_100 (V : U_100) return I_100;
10
11  function To_U_100 (V : I_100) return U_100;
```

(continues on next page)

(continued from previous page)

```

12
13  -- Include type declarations for D_50 and S_50!
14  --
15  -- [...] D_50 is [...]
16  -- [...] S_50 is [...]
17  --
18
19  function To_D_50 (V : I_100) return D_50;
20
21  function To_S_50 (V : I_100) return S_50;
22
23  function To_I_100 (V : D_50) return I_100;
24
25 end Int_Types;

```

Listing 47: int_types.adb

```

1  package body Int_Types is
2
3      function To_I_100 (V : U_100) return I_100 is
4      begin
5          -- Implement the conversion from U_100 to I_100 here!
6          --
7          null;
8      end To_I_100;
9
10     function To_U_100 (V : I_100) return U_100 is
11     begin
12         -- Implement the conversion from I_100 to U_100 here!
13         --
14         null;
15     end To_U_100;
16
17     function To_D_50 (V : I_100) return D_50 is
18         Min : constant I_100 := I_100 (D_50'First);
19         Max : constant I_100 := I_100 (D_50'Last);
20     begin
21         -- Implement the conversion from I_100 to D_50 here!
22         --
23         -- Hint: using the constants above simplifies the checks needed for
24         --       this function.
25         --
26         null;
27     end To_D_50;
28
29     function To_S_50 (V : I_100) return S_50 is
30     begin
31         -- Implement the conversion from I_100 to S_50 here!
32         --
33         -- Remark: don't forget to verify whether an explicit conversion like
34         --       S_50 (V) is needed.
35         --
36         null;
37     end To_S_50;
38
39     function To_I_100 (V : D_50) return I_100 is
40     begin
41         -- Implement the conversion from I_100 to D_50 here!
42         --
43         -- Remark: don't forget to verify whether an explicit conversion like
44         --       I_100 (V) is needed.

```

(continues on next page)

(continued from previous page)

```

45     --
46     null;
47     end To_I_100;
48
49 end Int_Types;

```

Listing 48: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Int_Types;        use Int_Types;
5
6  procedure Main is
7      package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
8      package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
9      package D_50_IO  is new Ada.Text_IO.Integer_IO (D_50);
10
11     use I_100_IO;
12     use U_100_IO;
13     use D_50_IO;
14
15     type Test_Case_Index is
16         (I_100_Range,
17          U_100_Range,
18          U_100_Wraparound,
19          U_100_To_I_100,
20          I_100_To_U_100,
21          D_50_Range,
22          S_50_Range,
23          I_100_To_D_50,
24          I_100_To_S_50,
25          D_50_To_I_100,
26          S_50_To_I_100);
27
28     procedure Check (TC : Test_Case_Index) is
29     begin
30         I_100_IO.Default_Width := 1;
31         U_100_IO.Default_Width := 1;
32         D_50_IO.Default_Width  := 1;
33
34         case TC is
35             when I_100_Range =>
36                 Put (I_100'First);
37                 New_Line;
38                 Put (I_100'Last);
39                 New_Line;
40             when U_100_Range =>
41                 Put (U_100'First);
42                 New_Line;
43                 Put (U_100'Last);
44                 New_Line;
45             when U_100_Wraparound =>
46                 Put (U_100'First - 1);
47                 New_Line;
48                 Put (U_100'Last + 1);
49                 New_Line;
50             when U_100_To_I_100 =>
51                 for I in U_100'Range loop
52                     I_100_IO.Put (To_I_100 (I));
53                     New_Line;

```

(continues on next page)

(continued from previous page)

```

54     end loop;
55     when I_100_To_U_100 =>
56         for I in I_100'Range loop
57             Put (To_U_100 (I));
58             New_Line;
59         end loop;
60     when D_50_Range =>
61         Put (D_50'First);
62         New_Line;
63         Put (D_50'Last);
64         New_Line;
65     when S_50_Range =>
66         Put (S_50'First);
67         New_Line;
68         Put (S_50'Last);
69         New_Line;
70     when I_100_To_D_50 =>
71         for I in I_100'Range loop
72             Put (To_D_50 (I));
73             New_Line;
74         end loop;
75     when I_100_To_S_50 =>
76         for I in I_100'Range loop
77             Put (To_S_50 (I));
78             New_Line;
79         end loop;
80     when D_50_To_I_100 =>
81         for I in D_50'Range loop
82             Put (To_I_100 (I));
83             New_Line;
84         end loop;
85     when S_50_To_I_100 =>
86         for I in S_50'Range loop
87             Put (I);
88             New_Line;
89         end loop;
90     end case;
91 end Check;
92
93 begin
94     if Argument_Count < 1 then
95         Put_Line ("ERROR: missing arguments! Exiting...");
96         return;
97     elsif Argument_Count > 1 then
98         Put_Line ("Ignoring additional arguments...");
99     end if;
100
101     Check (Test_Case_Index'Value (Argument (1)));
102 end Main;

```

4.3 Temperatures

Goal: create a package to handle temperatures in Celsius and Kelvin.

Steps:

1. Implement the Temperature_Types package.
 1. Declare the Celsius type.
 2. Declare the Int_Celsius type.

3. Implement the `To_Celsius` function.
4. Implement the `To_Int_Celsius` function.
5. Declare the Kelvin type.
6. Implement the `To_Celsius` function to convert from the Kelvin type.
7. Implement the `To_Kelvin` function.

Requirements:

1. The custom floating-point types declared in `Temperature_Types` must use a precision of six digits.
2. Types `Celsius` and `Int_Celsius` are used for temperatures in Celsius:
 1. `Celsius` is a floating-point type with a range between -273.15 and 5504.85.
 2. `Int_Celsius` is an integer type with a range between -273 and 5505.
3. Functions `To_Celsius` and `To_Int_Celsius` are used for type conversion:
 1. `To_Celsius` converts from `Int_Celsius` to `Celsius` type.
 2. `To_Int_Celsius` converts from `Celsius` and `Int_Celsius` types:
4. Kelvin is a floating-point type for temperatures in Kelvin using a range between 0.0 and 5778.0.
5. The functions `To_Celsius` and `To_Kelvin` are used to convert between temperatures in Kelvin and Celsius.
 1. In order to convert temperatures in Celsius to Kelvin, you must use the formula $K = C + 273.15$, where:
 - K is the temperature in Kelvin, and
 - C is the temperature in Celsius.

Remarks:

1. When implementing the `To_Celsius` function for the `Int_Celsius` type:
 1. You'll need to check for the minimum and maximum values of the input values because of the slightly different ranges.
 2. You may use variables of floating-point type (**Float**) for intermediate values.
2. For the implementation of the functions `To_Celsius` and `To_Kelvin` (used for converting between Kelvin and Celsius), you may use a variable of floating-point type (**Float**) for intermediate values.

Listing 49: `temperature_types.ads`

```
1 package Temperature_Types is
2
3   -- Include type declaration for Celsius!
4   --
5   -- Celsius is [...];
6   -- Int_Celsius is [...];
7   --
8
9   function To_Celsius (T : Int_Celsius) return Celsius;
10
11  function To_Int_Celsius (T : Celsius) return Int_Celsius;
12
13  -- Include type declaration for Kelvin!
14  --
15  -- type Kelvin is [...];
```

(continues on next page)

(continued from previous page)

```

16  --
17
18  -- Include function declarations for:
19  -- - Kelvin => Celsius
20  -- - Celsius => Kelvin
21  --
22  -- function To_Celsius [...];
23  -- function To_Kelvin  [...];
24  --
25  end Temperature_Types;

```

Listing 50: temperature_types.adb

```

1  package body Temperature_Types is
2
3      function To_Celsius (T : Int_Celsius) return Celsius is
4      begin
5          null;
6      end To_Celsius;
7
8      function To_Int_Celsius (T : Celsius) return Int_Celsius is
9      begin
10         null;
11     end To_Int_Celsius;
12
13     -- Include function implementation for:
14     -- - Kelvin => Celsius
15     -- - Celsius => Kelvin
16     --
17     -- function To_Celsius [...] is
18     -- function To_Kelvin  [...] is
19     --
20  end Temperature_Types;

```

Listing 51: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Temperature_Types; use Temperature_Types;
5
6  procedure Main is
7      package Celsius_IO is new Ada.Text_IO.Float_IO (Celsius);
8      package Kelvin_IO is new Ada.Text_IO.Float_IO (Kelvin);
9      package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);
10
11      use Celsius_IO;
12      use Kelvin_IO;
13      use Int_Celsius_IO;
14
15      type Test_Case_Index is
16          (Celsius_Range,
17           Celsius_To_Int_Celsius,
18           Int_Celsius_To_Celsius,
19           Kelvin_To_Celsius,
20           Celsius_To_Kelvin);
21
22      procedure Check (TC : Test_Case_Index) is
23      begin
24          Celsius_IO.Default_Fore := 1;

```

(continues on next page)

(continued from previous page)

```

25   Kelvin_IO.Default_Fore := 1;
26   Int_Celsius_IO.Default_Width := 1;
27
28   case TC is
29       when Celsius_Range =>
30           Put (Celsius'First);
31           New_Line;
32           Put (Celsius'Last);
33           New_Line;
34       when Celsius_To_Int_Celsius =>
35           Put (To_Int_Celsius (Celsius'First));
36           New_Line;
37           Put (To_Int_Celsius (0.0));
38           New_Line;
39           Put (To_Int_Celsius (Celsius'Last));
40           New_Line;
41       when Int_Celsius_To_Celsius =>
42           Put (To_Celsius (Int_Celsius'First));
43           New_Line;
44           Put (To_Celsius (0));
45           New_Line;
46           Put (To_Celsius (Int_Celsius'Last));
47           New_Line;
48       when Kelvin_To_Celsius =>
49           Put (To_Celsius (Kelvin'First));
50           New_Line;
51           Put (To_Celsius (0));
52           New_Line;
53           Put (To_Celsius (Kelvin'Last));
54           New_Line;
55       when Celsius_To_Kelvin =>
56           Put (To_Kelvin (Celsius'First));
57           New_Line;
58           Put (To_Kelvin (Celsius'Last));
59           New_Line;
60   end case;
61   end Check;
62
63   begin
64       if Argument_Count < 1 then
65           Put_Line ("ERROR: missing arguments! Exiting...");
66           return;
67       elsif Argument_Count > 1 then
68           Put_Line ("Ignoring additional arguments...");
69       end if;
70
71       Check (Test_Case_Index'Value (Argument (1)));
72   end Main;

```

RECORDS

5.1 Directions

Goal: create a package that handles directions and geometric angles.

Steps:

1. Implement the Directions package.
 1. Declare the Ext_Angle record.
 2. Implement the Display procedure.
 3. Implement the To_Ext_Angle function.

Requirements:

1. Record Ext_Angle stores information about the extended angle (see remark about *extended angles* below).
2. Procedure Display displays information about the extended angle.
 1. You should use the implementation that has been commented out (see code below) as a starting point.
3. Function To_Ext_Angle converts a simple angle value to an extended angle (Ext_Angle type).

Remarks:

1. We make use of the algorithm implemented in the Check_Direction procedure (chapter on imperative language).
2. For the sake of this exercise, we use the concept of *extended angles*. This includes the actual geometric angle and the corresponding direction (North, South, Northwest, and so on).

Listing 52: directions.ads

```
1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northeast,
8      East,
9      Southeast,
10     South,
11     Southwest,
12     West,
13     Northwest);
14
```

(continues on next page)

(continued from previous page)

```

15  function To_Direction (N: Angle_Mod) return Direction;
16
17  -- Include type declaration for Ext_Angle record type:
18  --
19  -- NOTE: Use the Angle_Mod and Direction types declared above!
20  --
21  -- type Ext_Angle is [...]
22  --
23
24  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
25
26  procedure Display (N : Ext_Angle);
27
28  end Directions;

```

Listing 53: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          -- Uncomment the code below and fill the missing elements
8          --
9          -- Put_Line ("Angle: "
10             & Angle_Mod'Image (____)
11             & " => "
12             & Direction'Image (____)
13             & ".");
14          null;
15      end Display;
16
17      function To_Direction (N : Angle_Mod) return Direction is
18      begin
19          case N is
20              when 0      => return North;
21              when 1 .. 89 => return Northeast;
22              when 90     => return East;
23              when 91 .. 179 => return Southeast;
24              when 180    => return South;
25              when 181 .. 269 => return Southwest;
26              when 270    => return West;
27              when 271 .. 359 => return Northwest;
28          end case;
29      end To_Direction;
30
31      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
32      begin
33          -- Implement the conversion from Angle_Mod to Ext_Angle here!
34          --
35          -- Hint: you can use a return statement and an aggregate.
36          --
37          null;
38      end To_Ext_Angle;
39
40  end Directions;

```

Listing 54: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Directions;       use Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13         when Direction_Chk =>
14             Display (To_Ext_Angle (0));
15             Display (To_Ext_Angle (30));
16             Display (To_Ext_Angle (45));
17             Display (To_Ext_Angle (90));
18             Display (To_Ext_Angle (91));
19             Display (To_Ext_Angle (120));
20             Display (To_Ext_Angle (180));
21             Display (To_Ext_Angle (250));
22             Display (To_Ext_Angle (270));
23         end case;
24     end Check;
25
26 begin
27     if Argument_Count < 1 then
28         Put_Line ("ERROR: missing arguments! Exiting...");
29         return;
30     elsif Argument_Count > 1 then
31         Put_Line ("Ignoring additional arguments...");
32     end if;
33
34     Check (Test_Case_Index'Value (Argument (1)));
35 end Main;

```

5.2 Colors

Goal: create a package to represent HTML colors in RGB format using the hexadecimal form.

Steps:

1. Implement the Color_Types package.
 1. Declare the RGB record.
 2. Implement the To_RGB function.
 3. Implement the Image function for the RGB type.

Requirements:

1. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

Color	Red	Green	Blue
Salmon	#FA	#80	#72
Firebrick	#B2	#22	#22
Red	#FF	#00	#00
Darkred	#8B	#00	#00
Lime	#00	#FF	#00
Forestgreen	#22	#8B	#22
Green	#00	#80	#00
Darkgreen	#00	#64	#00
Blue	#00	#00	#FF
Mediumblue	#00	#00	#CD
Darkblue	#00	#00	#8B

2. The hexadecimal information of each HTML color can be mapped to three color elements: red, green and blue.
 1. Each color element has a value between 0 and 255, or 00 and FF in hexadecimal.
 2. For example, for the color *salmon*, the hexadecimal value of the color elements are:
 - red = FA,
 - green = 80, and
 - blue = 72.
3. Record RGB stores information about HTML colors in RGB format, so that we can retrieve the individual color elements.
4. Function To_RGB converts from the HTML_Color enumeration to the RGB type based on the information from the table above.
5. Function Image returns a string representation of the RGB type in this format:
 - "(Red => 16#..#, Green => 16#...#, Blue => 16#...#)"

Remarks:

1. We use the exercise on HTML colors from the previous lab on *Strongly typed language* (page 21) as a starting point.

Listing 55: color_types.ads

```
1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   function To_Integer (C : HTML_Color) return Integer;
17
18   type Basic_HTML_Color is
19     (Red,
```

(continues on next page)

(continued from previous page)

```

20     Green,
21     Blue);
22
23     function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25     subtype Int_Color is Integer range 0 .. 255;
26
27     -- Replace type declaration for RGB record below
28     --
29     -- - NOTE: Use the Int_Color type declared above!
30     --
31     -- type RGB is [...]
32     --
33     type RGB is null record;
34
35     function To_RGB (C : HTML_Color) return RGB;
36
37     function Image (C : RGB) return String;
38
39 end Color_Types;

```

Listing 56: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_Integer (C : HTML_Color) return Integer is
6      begin
7          case C is
8              when Salmon    => return 16#FA8072#;
9              when Firebrick => return 16#B22222#;
10             when Red        => return 16#FF0000#;
11             when Darkred    => return 16#8B0000#;
12             when Lime       => return 16#00FF00#;
13             when Forestgreen => return 16#228B22#;
14             when Green      => return 16#008000#;
15             when Darkgreen  => return 16#006400#;
16             when Blue       => return 16#0000FF#;
17             when Mediumblue => return 16#0000CD#;
18             when Darkblue   => return 16#00008B#;
19          end case;
20
21      end To_Integer;
22
23      function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
24      begin
25          case C is
26              when Red    => return Red;
27              when Green  => return Green;
28              when Blue   => return Blue;
29          end case;
30      end To_HTML_Color;
31
32      function To_RGB (C : HTML_Color) return RGB is
33      begin
34          -- Implement the conversion from HTML_Color to RGB here!
35          --
36          return (null record);
37      end To_RGB;
38

```

(continues on next page)

(continued from previous page)

```

39  function Image (C : RGB) return String is
40      subtype Str_Range is Integer range 1 .. 10;
41      SR : String (Str_Range);
42      SG : String (Str_Range);
43      SB : String (Str_Range);
44  begin
45      -- Replace argument in the calls to Put below
46      -- with the missing elements (red, green, blue)
47      -- from the RGB record
48      --
49      Ada.Integer_Text_IO.Put (To   => SR,
50                             Item => 0,   -- REPLACE!
51                             Base => 16);
52      Ada.Integer_Text_IO.Put (To   => SG,
53                             Item => 0,   -- REPLACE!
54                             Base => 16);
55      Ada.Integer_Text_IO.Put (To   => SB,
56                             Item => 0,   -- REPLACE!
57                             Base => 16);
58      return ("(Red => " & SR
59             & ", Green => " & SG
60             & ", Blue => " & SB
61             & ")");
62  end Image;
63
64  end Color_Types;

```

Listing 57: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_To_RGB);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when HTML_Color_To_RGB =>
14                 for I in HTML_Color'Range loop
15                     Put_Line (HTML_Color'Image (I) & " => "
16                             & Image (To_RGB (I)) & ".");
17                 end loop;
18             end case;
19     end Check;
20
21  begin
22      if Argument_Count < 1 then
23          Put_Line ("ERROR: missing arguments! Exiting...");
24          return;
25      elsif Argument_Count > 1 then
26          Put_Line ("Ignoring additional arguments...");
27      end if;
28
29      Check (Test_Case_Index'Value (Argument (1)));
30  end Main;

```

5.3 Inventory

Goal: create a simplified inventory system for a store to enter items and keep track of assets.

Steps:

1. Implement the Inventory_Pkg package.
 1. Declare the Item record.
 2. Implement the Init function.
 3. Implement the Add procedure.

Requirements:

1. Record Item collects information about products from the store.
 1. To keep it simple, this record only contains the name, quantity and price of each item.
 2. The record components are:
 - Name of Item_Name type;
 - Quantity of **Natural** type;
 - Price of **Float** type.
2. Function Init returns an initialized item (of Item type).
 1. Function Init must also display the item name by calling the To_String function for the Item_Name type.
 - This is already implemented in the code below.
3. Procedure Add adds an item to the assets.
 1. Since we want to keep track of the assets, the implementation must accumulate the total value of each item's inventory, the result of multiplying the item quantity and its price.

Listing 58: inventory_pkg.ads

```

1 package Inventory_Pkg is
2
3   type Item_Name is
4     (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);
5
6   function To_String (I : Item_Name) return String;
7
8   -- Replace type declaration for Item record:
9   --
10  type Item is null record;
11
12  function Init (Name      : Item_Name;
13                Quantity  : Natural;
14                Price     : Float) return Item;
15
16  procedure Add (Assets : in out Float;
17                I       : Item);
18
19 end Inventory_Pkg;
```

Listing 59: inventory_pkg.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Inventory_Pkg is
4
5     function To_String (I : Item_Name) return String is
6     begin
7         case I is
8             when Ballpoint_Pen      => return "Ballpoint Pen";
9             when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
10            when Feather_Quill_Pen   => return "Feather Quill Pen";
11        end case;
12    end To_String;
13
14    function Init (Name      : Item_Name;
15                  Quantity  : Natural;
16                  Price     : Float) return Item is
17    begin
18        Put_Line ("Item: " & To_String (Name) & ".");
19
20        -- Replace return statement with the actual record initialization!
21        --
22        return (null record);
23    end Init;
24
25    procedure Add (Assets : in out Float;
26                  I       : Item) is
27    begin
28        -- Implement the function that adds an item to the inventory here!
29        --
30        null;
31    end Add;
32
33 end Inventory_Pkg;
```

Listing 60: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Inventory_Pkg;     use Inventory_Pkg;
5
6 procedure Main is
7     -- Remark: the following line is not relevant.
8     F : array (1 .. 10) of Float := (others => 42.42);
9
10    type Test_Case_Index is
11        (Inventory_Chk);
12
13    procedure Display (Assets : Float) is
14        package F_IO is new Ada.Text_IO.Float_IO (Float);
15
16        use F_IO;
17    begin
18        Put ("Assets: $");
19        Put (Assets, 1, 2, 0);
20        Put (".");
21        New_Line;
22    end Display;
23
```

(continues on next page)

(continued from previous page)

```

24  procedure Check (TC : Test_Case_Index) is
25      I      : Item;
26      Assets : Float := 0.0;
27
28      -- Please ignore the following three lines!
29      pragma Warnings (Off, "default initialization");
30      for Assets'Address use F'Address;
31      pragma Warnings (On, "default initialization");
32  begin
33      case TC is
34      when Inventory_Chk =>
35          I := Init (Ballpoint_Pen,      185,  0.15);
36          Add (Assets, I);
37          Display (Assets);
38
39          I := Init (Oil_Based_Pen_Marker, 100,  9.0);
40          Add (Assets, I);
41          Display (Assets);
42
43          I := Init (Feather_Quill_Pen,    2, 40.0);
44          Add (Assets, I);
45          Display (Assets);
46      end case;
47  end Check;
48
49  begin
50      if Argument_Count < 1 then
51          Put_Line ("ERROR: missing arguments! Exiting...");
52          return;
53      elsif Argument_Count > 1 then
54          Put_Line ("Ignoring additional arguments...");
55      end if;
56
57      Check (Test_Case_Index'Value (Argument (1)));
58  end Main;

```


ARRAYS

6.1 Constrained Array

Goal: declare a constrained array and implement operations on it.

Steps:

1. Implement the `Constrained_Arrays` package.
 1. Declare the range type `My_Index`.
 2. Declare the array type `My_Array`.
 3. Declare and implement the `Init` function.
 4. Declare and implement the `Double` procedure.
 5. Declare and implement the `First_Elem` function.
 6. Declare and implement the `Last_Elem` function.
 7. Declare and implement the `Length` function.
 8. Declare the object `A` of `My_Array` type.

Requirements:

1. Range type `My_Index` has a range from 1 to 10.
2. `My_Array` is a constrained array of **Integer** type.
 1. It must make use of the `My_Index` type.
 2. It is therefore limited to 10 elements.
3. Function `Init` returns an array where each element is initialized with the corresponding index.
4. Procedure `Double` doubles the value of each element of an array.
5. Function `First_Elem` returns the first element of the array.
6. Function `Last_Elem` returns the last element of the array.
7. Function `Length` returns the length of the array.
8. Object `A` of `My_Array` type is initialized with:
 1. the values 1 and 2 for the first two elements, and
 2. 42 for all other elements.

Listing 61: constrained_arrays.ads

```
1 package Constrained_Arrays is
2
3     -- Complete the type and subprogram declarations:
4     --
5     -- type My_Index is [...]
6     --
7     -- type My_Array is [...]
8     --
9     -- function Init ...
10    --
11    -- procedure Double ...
12    --
13    -- function First_Elem ...
14    --
15    -- function Last_Elem ...
16    --
17    -- function Length ...
18    --
19    -- A : ...
20
21 end Constrained_Arrays;
```

Listing 62: constrained_arrays.adb

```
1 package body Constrained_Arrays is
2
3     -- Create the implementation of the subprograms!
4     --
5
6 end Constrained_Arrays;
```

Listing 63: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Constrained_Arrays; use Constrained_Arrays;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Range_Chk,
9          Array_Range_Chk,
10         A_Obj_Chk,
11         Init_Chk,
12         Double_Chk,
13         First_Elem_Chk,
14         Last_Elem_Chk,
15         Length_Chk);
16
17     procedure Check (TC : Test_Case_Index) is
18         AA : My_Array;
19
20         procedure Display (A : My_Array) is
21             begin
22                 for I in A'Range loop
23                     Put_Line (Integer'Image (A (I)));
24                 end loop;
25             end Display;
26
27     procedure Local_Init (A : in out My_Array) is
```

(continues on next page)

(continued from previous page)

```

28     begin
29         A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
30     end Local_Init;
31 begin
32     case TC is
33     when Range_Chk =>
34         for I in My_Index loop
35             Put_Line (My_Index'Image (I));
36         end loop;
37     when Array_Range_Chk =>
38         for I in My_Array'Range loop
39             Put_Line (My_Index'Image (I));
40         end loop;
41     when A_Obj_Chk =>
42         Display (A);
43     when Init_Chk =>
44         AA := Init;
45         Display (AA);
46     when Double_Chk =>
47         Local_Init (AA);
48         Double (AA);
49         Display (AA);
50     when First_Elem_Chk =>
51         Local_Init (AA);
52         Put_Line (Integer'Image (First_Elem (AA)));
53     when Last_Elem_Chk =>
54         Local_Init (AA);
55         Put_Line (Integer'Image (Last_Elem (AA)));
56     when Length_Chk =>
57         Put_Line (Integer'Image (Length (AA)));
58     end case;
59 end Check;
60
61 begin
62     if Argument_Count < 1 then
63         Put_Line ("ERROR: missing arguments! Exiting...");
64         return;
65     elsif Argument_Count > 1 then
66         Put_Line ("Ignoring additional arguments...");
67     end if;
68
69     Check (Test_Case_Index'Value (Argument (1)));
70 end Main;

```

6.2 Colors: Lookup-Table

Goal: rewrite a package to represent HTML colors in RGB format using a lookup table.

Steps:

1. Implement the Color_Types package.
 1. Declare the array type HTML_Color_RGB.
 2. Declare the To_RGB_Lookup_Table object and initialize it.
 3. Adapt the implementation of the To_RGB function.

Requirements:

1. Array type HTML_Color_RGB is used for the table.
2. The To_RGB_Lookup_Table object of HTML_Color_RGB type contains the lookup table.

- This table must be implemented as an array of constant values.
3. The implementation of the To_RGB function must use the To_RGB_Lookup_Table object.

Remarks:

1. This exercise is based on the HTML colors exercise from a previous lab ([Records](#) (page 31)).
2. In the previous implementation, you could use a **case** statement to implement the To_RGB function. Here, you must rewrite the function using a look-up table.
 1. The implementation of the To_RGB function below includes the case statement as commented-out code. You can use this as your starting point: you just need to copy it and convert the case statement to an array declaration.
 1. Don't use a case statement to implement the To_RGB function. Instead, write code that accesses To_RGB_Lookup_Table to get the correct value.
3. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

Color	Red	Green	Blue
Salmon	#FA	#80	#72
Firebrick	#B2	#22	#22
Red	#FF	#00	#00
Darkred	#8B	#00	#00
Lime	#00	#FF	#00
Forestgreen	#22	#8B	#22
Green	#00	#80	#00
Darkgreen	#00	#64	#00
Blue	#00	#00	#FF
Mediumblue	#00	#00	#CD
Darkblue	#00	#00	#8B

Listing 64: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   subtype Int_Color is Integer range 0 .. 255;
17
18   type RGB is record
19     Red   : Int_Color;
20     Green : Int_Color;
21     Blue  : Int_Color;
22   end record;
23

```

(continues on next page)

(continued from previous page)

```

24  function To_RGB (C : HTML_Color) return RGB;
25
26  function Image (C : RGB) return String;
27
28  -- Declare array type for lookup table here:
29  --
30  -- type HTML_Color_RGB is ...
31
32  -- Declare lookup table here:
33  --
34  -- To_RGB_Lookup_Table : ...
35
36  end Color_Types;

```

Listing 65: color_types.adb

```

1  with Ada.Integer_Text_IO;
2  package body Color_Types is
3
4      function To_RGB (C : HTML_Color) return RGB is
5      begin
6          -- Implement To_RGB using To_RGB_Lookup_Table
7          return (0, 0, 0);
8
9          -- Use the code below from the previous version of the To_RGB
10         -- function to declare the To_RGB_Lookup_Table:
11         --
12         -- case C is
13         --     when Salmon      => return (16#FA#, 16#80#, 16#72#);
14         --     when Firebrick   => return (16#B2#, 16#22#, 16#22#);
15         --     when Red         => return (16#FF#, 16#00#, 16#00#);
16         --     when Darkred     => return (16#8B#, 16#00#, 16#00#);
17         --     when Lime        => return (16#00#, 16#FF#, 16#00#);
18         --     when Forestgreen => return (16#22#, 16#8B#, 16#22#);
19         --     when Green       => return (16#00#, 16#80#, 16#00#);
20         --     when Darkgreen   => return (16#00#, 16#64#, 16#00#);
21         --     when Blue        => return (16#00#, 16#00#, 16#FF#);
22         --     when Mediumblue  => return (16#00#, 16#00#, 16#CD#);
23         --     when Darkblue    => return (16#00#, 16#00#, 16#8B#);
24         -- end case;
25
26     end To_RGB;
27
28     function Image (C : RGB) return String is
29         subtype Str_Range is Integer range 1 .. 10;
30         SR : String (Str_Range);
31         SG : String (Str_Range);
32         SB : String (Str_Range);
33     begin
34         Ada.Integer_Text_IO.Put (To    => SR,
35                                Item   => C.Red,
36                                Base   => 16);
37         Ada.Integer_Text_IO.Put (To    => SG,
38                                Item   => C.Green,
39                                Base   => 16);
40         Ada.Integer_Text_IO.Put (To    => SB,
41                                Item   => C.Blue,
42                                Base   => 16);
43
44         return ("(Red => " & SR
45                & ", Green => " & SG
46                & ", Blue => " & SB

```

(continues on next page)

(continued from previous page)

```

46         &"");
47     end Image;
48
49 end Color_Types;

```

Listing 66: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Color_Types;           use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Color_Table_Chk,
9           HTML_Color_To_Integer_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Color_Table_Chk =>
15                 Put_Line ("Size of HTML_Color_RGB: "
16                     & Integer'Image (HTML_Color_RGB'Length));
17                 Put_Line ("Firebrick: "
18                     & Image (To_RGB_Lookup_Table (Firebrick)));
19             when HTML_Color_To_Integer_Chk =>
20                 for I in HTML_Color'Range loop
21                     Put_Line (HTML_Color'Image (I) & " => "
22                         & Image (To_RGB (I)) & ".");
23                 end loop;
24             end case;
25     end Check;
26
27 begin
28     if Argument_Count < 1 then
29         Put_Line ("ERROR: missing arguments! Exiting...");
30         return;
31     elsif Argument_Count > 1 then
32         Put_Line ("Ignoring additional arguments...");
33     end if;
34
35     Check (Test_Case_Index'Value (Argument (1)));
36 end Main;

```

6.3 Unconstrained Array

Goal: declare an unconstrained array and implement operations on it.

Steps:

1. Implement the Unconstrained_Arrays package.
 1. Declare the My_Array type.
 2. Declare and implement the Init procedure.
 3. Declare and implement the Init function.
 4. Declare and implement the Double procedure.
 5. Declare and implement the Diff_Prev_Elem function.

Requirements:

1. My_Array is an unconstrained array (with a **Positive** range) of **Integer** elements.
2. Procedure Init initializes each element with the index starting with the last one.
 - For example, for an array of 3 elements where the index of the first element is 1 (My_Array (1 .. 3)), the values of these elements after a call to Init must be (3, 2, 1).
3. Function Init returns an array based on the length L and start index I provided to the Init function.
 1. I indicates the index of the first element of the array.
 2. L indicates the length of the array.
 3. Both I and L must be positive.
 4. This is its declaration: **function** Init (I, L : Positive) **return** My_Array;.
 5. You must initialize the elements of the array in the same manner as for the Init procedure described above.
4. Procedure Double doubles each element of an array.
5. Function Diff_Prev_Elem returns — for each element of an input array A — an array with the difference between an element of array A and the previous element.
 1. For the first element, the difference must be zero.
 2. For example:
 - **INPUT:** (2, 5, 15)
 - **RETURN** of Diff_Prev_Elem: (0, 3, 10), where
 - 0 is the constant difference for the first element;
 - $5 - 2 = 3$ is the difference between the second and the first elements of the input array;
 - $15 - 5 = 10$ is the difference between the third and the second elements of the input array.

Remarks:

1. For an array A, you can retrieve the index of the last element with the attribute 'Last'.
 1. For example: Y : **Positive** := A'Last;
 2. This can be useful during the implementation of procedure Init.
2. For the implementation of the Init function, you can call the Init procedure to initialize the elements. By doing this, you avoid code duplication.
3. Some hints about attributes:
 1. You can use the range attribute (A'Range) to retrieve the range of an array A.
 2. You can also use the range attribute in the declaration of another array (e.g.: B : My_Array (A'Range)).
 3. Alternatively, you can use the A'First and A'Last attributes in an array declaration.

Listing 67: unconstrained_arrays.ads

```

1 package Unconstrained_Arrays is
2
3   -- Complete the type and subprogram declarations:

```

(continues on next page)

(continued from previous page)

```

4      --
5      --  type My_Array is ...;
6      --
7      --  procedure Init ...;
8
9      function Init (I, L : Positive) return My_Array;
10
11     --  procedure Double ...;
12     --
13     --  function Diff_Prev_Elem ...;
14
15 end Unconstrained_Arrays;

```

Listing 68: unconstrained_arrays.adb

```

1 package body Unconstrained_Arrays is
2
3     --  Implement the subprograms:
4     --
5
6     --  procedure Init is...
7
8     --  function Init (L : Positive) return My_Array is...
9
10    --  procedure Double ... is...
11
12    --  function Diff_Prev_Elem ... is...
13
14 end Unconstrained_Arrays;

```

Listing 69: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Unconstrained_Arrays; use Unconstrained_Arrays;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Init_Chk,
9          Init_Proc_Chk,
10         Double_Chk,
11         Diff_Prev_Chk,
12         Diff_Prev_Single_Chk);
13
14     procedure Check (TC : Test_Case_Index) is
15         AA : My_Array (1 .. 5);
16         AB : My_Array (5 .. 9);
17
18         procedure Display (A : My_Array) is
19             begin
20                 for I in A'Range loop
21                     Put_Line (Integer'Image (A (I)));
22                 end loop;
23             end Display;
24
25         procedure Local_Init (A : in out My_Array) is
26             begin
27                 A := (1, 2, 5, 10, -10);
28             end Local_Init;

```

(continues on next page)

(continued from previous page)

```

29
30   begin
31       case TC is
32       when Init_Chk =>
33           AA := Init (AA'First, AA'Length);
34           AB := Init (AB'First, AB'Length);
35           Display (AA);
36           Display (AB);
37       when Init_Proc_Chk =>
38           Init (AA);
39           Init (AB);
40           Display (AA);
41           Display (AB);
42       when Double_Chk =>
43           Local_Init (AB);
44           Double (AB);
45           Display (AB);
46       when Diff_Prev_Chk =>
47           Local_Init (AB);
48           AB := Diff_Prev_Elem (AB);
49           Display (AB);
50       when Diff_Prev_Single_Chk =>
51           declare
52               A1 : My_Array (1 .. 1) := (1 => 42);
53           begin
54               A1 := Diff_Prev_Elem (A1);
55               Display (A1);
56           end;
57       end case;
58   end Check;
59
60   begin
61       if Argument_Count < 1 then
62           Put_Line ("ERROR: missing arguments! Exiting...");
63           return;
64       elsif Argument_Count > 1 then
65           Put_Line ("Ignoring additional arguments...");
66       end if;
67
68       Check (Test_Case_Index'Value (Argument (1)));
69   end Main;

```

6.4 Product info

Goal: create a system to keep track of quantities and prices of products.

Steps:

1. Implement the Product_Info_Pkg package.
 1. Declare the array type Product_Infos.
 2. Declare the array type Currency_Array.
 3. Implement the Total procedure.
 4. Implement the Total function returning an array of Currency_Array type.
 5. Implement the Total function returning a single value of Currency type.

Requirements:

1. Quantity of an individual product is represented by the Quantity subtype.

2. Price of an individual product is represented by the Currency subtype.
3. Record type Product_Info deals with information for various products.
4. Array type Product_Infos is used to represent a list of products.
5. Array type Currency_Array is used to represent a list of total values of individual products (see more details below).
6. Procedure Total receives an input array of products.
 1. It outputs an array with the total value of each product using the Currency_Array type.
 2. The total value of an individual product is calculated by multiplying the quantity for this product by its price.
7. Function Total returns an array of Currency_Array type.
 1. This function has the same purpose as the procedure Total.
 2. The difference is that the function returns an array instead of providing this array as an output parameter.
8. The second function Total returns a single value of Currency type.
 1. This function receives an array of products.
 2. It returns a single value corresponding to the total value for all products in the system.

Remarks:

1. You can use Currency (Q) to convert from an element Q of Quantity type to the Currency type.
 1. As you might remember, Ada requires an explicit conversion in calculations where variables of both integer and floating-point types are used.
 2. In our case, the Quantity subtype is based on the **Integer** type and the Currency subtype is based on the **Float** type, so a conversion is necessary in calculations using those types.

Listing 70: product_info_pkg.ads

```
1 package Product_Info_Pkg is
2
3     subtype Quantity is Natural;
4
5     subtype Currency is Float;
6
7     type Product_Info is record
8         Units : Quantity;
9         Price : Currency;
10    end record;
11
12    -- Complete the type declarations:
13    --
14    -- type Product_Infos is ...
15    --
16    -- type Currency_Array is ...
17
18    procedure Total (P : Product_Infos;
19                    Tot : out Currency_Array);
20
21    function Total (P : Product_Infos) return Currency_Array;
22
23    function Total (P : Product_Infos) return Currency;
```

(continues on next page)

(continued from previous page)

```

24
25 end Product_Info_Pkg;

```

Listing 71: product_info_pkg.adb

```

1 package body Product_Info_Pkg is
2
3   -- Complete the subprogram implementations:
4   --
5
6   -- procedure Total (P : Product_Infos;
7   --                  Tot : out Currency_Array) is ...
8
9   -- function Total (P : Product_Infos) return Currency_Array is ...
10
11  -- function Total (P : Product_Infos) return Currency is ...
12
13 end Product_Info_Pkg;

```

Listing 72: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Product_Info_Pkg; use Product_Info_Pkg;
5
6 procedure Main is
7   package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
8
9   type Test_Case_Index is
10     (Total_Func_Chk,
11      Total_Proc_Chk,
12      Total_Value_Chk);
13
14   procedure Check (TC : Test_Case_Index) is
15     subtype Test_Range is Positive range 1 .. 5;
16
17     P : Product_Infos (Test_Range);
18     Tots : Currency_Array (Test_Range);
19     Tot : Currency;
20
21     procedure Display (Tots : Currency_Array) is
22     begin
23       for I in Tots'Range loop
24         Currency_IO.Put (Tots (I));
25         New_Line;
26       end loop;
27     end Display;
28
29     procedure Local_Init (P : in out Product_Infos) is
30     begin
31       P := ((1, 0.5),
32            (2, 10.0),
33            (5, 40.0),
34            (10, 10.0),
35            (10, 20.0));
36     end Local_Init;
37
38   begin
39     Currency_IO.Default_Fore := 1;

```

(continues on next page)

(continued from previous page)

```

40     Currency_IO.Default_Aft := 2;
41     Currency_IO.Default_Exp := 0;
42
43     case TC is
44     when Total_Func_Chk =>
45         Local_Init (P);
46         Tots := Total (P);
47         Display (Tots);
48     when Total_Proc_Chk =>
49         Local_Init (P);
50         Total (P, Tots);
51         Display (Tots);
52     when Total_Value_Chk =>
53         Local_Init (P);
54         Tot := Total (P);
55         Currency_IO.Put (Tot);
56         New_Line;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

6.5 String_10

Goal: work with constrained string types.

Steps:

1. Implement the Strings_10 package.
 1. Declare the String_10 type.
 2. Implement the To_String_10 function.

Requirements:

1. The constrained string type String_10 is an array of ten characters.
2. Function To_String_10 returns constrained strings of String_10 type based on an input parameter of **String** type.
 - For strings that are more than 10 characters, omit everything after the 11th character.
 - For strings that are fewer than 10 characters, pad the string with ' ' characters until it is 10 characters.

Remarks:

1. Declaring String_10 as a subtype of **String** is the easiest way.
 - You may declare it as a new type as well. However, this requires some adaptations in the Main test procedure.
2. You can use **Integer'Min** to calculate the minimum of two integer values.

Listing 73: strings_10.ads

```

1 package Strings_10 is
2
3   -- Complete the type and subprogram declarations:
4   --
5
6   -- subtype String_10 is ...;
7
8   -- Using "type String_10 is..." is possible, too. However, it
9   -- requires a custom Put_Line procedure that is called in Main:
10  -- procedure Put_Line (S : String_10);
11
12  -- function To_String_10 ...;
13
14 end Strings_10;

```

Listing 74: strings_10.adb

```

1 package body Strings_10 is
2
3   -- Complete the subprogram declaration and implementation:
4   --
5   -- function To_String_10 ... is
6
7 end Strings_10;

```

Listing 75: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3
4 with Strings_10;          use Strings_10;
5
6 procedure Main is
7   type Test_Case_Index is
8     (String_10_Long_Chk,
9      String_10_Short_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12    SL  : constant String := "And this is a long string just for testing...";
13    SS  : constant String := "Hey!";
14    S_10 : String_10;
15
16  begin
17    case TC is
18      when String_10_Long_Chk =>
19        S_10 := To_String_10 (SL);
20        Put_Line (String (S_10));
21      when String_10_Short_Chk =>
22        S_10 := (others => ' ');
23        S_10 := To_String_10 (SS);
24        Put_Line (String (S_10));
25      end case;
26  end Check;
27
28 begin
29   if Argument_Count < 1 then
30     Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
31     return;
32   elsif Argument_Count > 1 then

```

(continues on next page)

(continued from previous page)

```
33     Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
34   end if;
35
36   Check (Test_Case_Index'Value (Argument (1)));
37 end Main;
```

6.6 List of Names

Goal: create a system for a list of names and ages.

Steps:

1. Implement the Names_Ages package.
 1. Declare the People_Array array type.
 2. Complete the declaration of the People record type with the People_A element of People_Array type.
 3. Implement the Add procedure.
 4. Implement the Reset procedure.
 5. Implement the Get function.
 6. Implement the Update procedure.
 7. Implement the Display procedure.

Requirements:

1. Each person is represented by the Person type, which is a record containing the name and the age of that person.
2. People_Array is an unconstrained array of Person type with a positive range.
3. The Max_People constant is set to 10.
4. Record type People contains:
 1. The People_A element of People_Array type.
 2. This array must be constrained by the Max_People constant.
5. Procedure Add adds a person to the list.
 1. By default, the age of this person is set to zero in this procedure.
6. Procedure Reset resets the list.
7. Function Get retrieves the age of a person from the list.
8. Procedure Update updates the age of a person in the list.
9. Procedure Display shows the complete list using the following format:
 1. The first line must be LIST OF NAMES:. It is followed by the name and age of each person in the next lines.
 2. For each person on the list, the procedure must display the information in the following format:

```
NAME: XXXX
AGE: YY
```

Remarks:

1. In the implementation of procedure `Add`, you may use an index to indicate the last valid position in the array — see `Last_Valid` in the code below.
2. In the implementation of procedure `Display`, you should use the `Trim` function from the `Ada.Strings.Fixed` package to format the person's name — for example: `Trim (P.Name, Right)`.
3. You may need the `Integer'Min` (A, B) and the `Integer'Max` (A, B) functions to get the minimum and maximum values in a comparison between two integer values A and B.
4. Fixed-length strings can be initialized with whitespaces using the `others` syntax. For example: `S : String_10 := (others => ' ');`
5. You may implement additional subprograms to deal with other types declared in the `Names_Ages` package below, such as the `Name_Type` and the `Person` type.
 1. For example, a function `To_Name_Type` to convert from `String` to `Name_Type` might be useful.
 2. Take a moment to reflect on which additional subprograms could be useful as well.

Listing 76: names_ages.ads

```

1 package Names_Ages is
2
3   Max_People : constant Positive := 10;
4
5   subtype Name_Type is String (1 .. 50);
6
7   type Age_Type is new Natural;
8
9   type Person is record
10     Name : Name_Type;
11     Age  : Age_Type;
12   end record;
13
14   -- Add type declaration for People_Array record:
15   --
16   -- type People_Array is ...;
17
18   -- Replace type declaration for People record. You may use the
19   -- following template:
20   --
21   -- type People is record
22   --   People_A : People_Array ...;
23   --   Last_Valid : Natural;
24   -- end record;
25   --
26   type People is null record;
27
28   procedure Reset (P : in out People);
29
30   procedure Add (P : in out People;
31                 Name : String);
32
33   function Get (P : People;
34                Name : String) return Age_Type;
35
36   procedure Update (P : in out People;
37                    Name : String;
38                    Age : Age_Type);
39
40   procedure Display (P : People);

```

(continues on next page)

(continued from previous page)

```

41
42 end Names_Ages;

```

Listing 77: names_ages.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Ada.Strings;      use Ada.Strings;
3  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
4
5  package body Names_Ages is
6
7      procedure Reset (P : in out People) is
8      begin
9          null;
10     end Reset;
11
12     procedure Add (P : in out People;
13                  Name : String) is
14     begin
15         null;
16     end Add;
17
18     function Get (P : People;
19                 Name : String) return Age_Type is
20     begin
21         return 0;
22     end Get;
23
24     procedure Update (P : in out People;
25                     Name : String;
26                     Age : Age_Type) is
27     begin
28         null;
29     end Update;
30
31     procedure Display (P : People) is
32     begin
33         null;
34     end Display;
35
36 end Names_Ages;

```

Listing 78: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Names_Ages;      use Names_Ages;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Names_Ages_Chk,
9           Get_Age_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     P : People;
13     begin
14         case TC is
15         when Names_Ages_Chk =>
16             Reset (P);

```

(continues on next page)

(continued from previous page)

```

17      Add (P, "John");
18      Add (P, "Patricia");
19      Add (P, "Josh");
20      Display (P);
21      Update (P, "John", 18);
22      Update (P, "Patricia", 35);
23      Update (P, "Josh", 53);
24      Display (P);
25      when Get_Age_Chk =>
26          Reset (P);
27          Add (P, "Peter");
28          Update (P, "Peter", 45);
29          Put_Line ("Peter is "
30                  & Age_Type'Image (Get (P, "Peter"))
31                  & " years old.");
32      end case;
33  end Check;
34
35  begin
36      if Argument_Count < 1 then
37          Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
38          return;
39      elsif Argument_Count > 1 then
40          Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
41      end if;
42
43      Check (Test_Case_Index'Value (Argument (1)));
44  end Main;

```


MORE ABOUT TYPES

7.1 Aggregate Initialization

Goal: initialize records and arrays using aggregates.

Steps:

1. Implement the Aggregates package.
 1. Create the record type Rec.
 2. Create the array type Int_Arr.
 3. Implement the Init procedure that outputs a record of Rec type.
 4. Implement the Init_Some procedure.
 5. Implement the Init procedure that outputs an array of Int_Arr type.

Requirements:

1. Record type Rec has four components of **Integer** type. These are the components with the corresponding default values:
 - W = 10
 - X = 11
 - Y = 12
 - Z = 13
2. Array type Int_Arr has 20 elements of **Integer** type (with indices ranging from 1 to 20).
3. The first Init procedure outputs a record of Rec type where:
 1. X is initialized with 100,
 2. Y is initialized with 200, and
 3. the remaining elements use their default values.
4. Procedure Init_Some outputs an array of Int_Arr type where:
 1. the first five elements are initialized with the value 99, and
 2. the remaining elements are initialized with the value 100.
5. The second Init procedure outputs an array of Int_Arr type where:
 1. all elements are initialized with the value 5.

Listing 79: aggregates.ads

```
1 package Aggregates is
2
3     -- type Rec is ...;
4
5     -- type Int_Arr is ...;
6
7     procedure Init;
8
9     -- procedure Init_Some ...;
10
11     -- procedure Init ...;
12
13 end Aggregates;
```

Listing 80: aggregates.adb

```
1 package body Aggregates is
2
3     procedure Init is null;
4
5 end Aggregates;
```

Listing 81: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Aggregates; use Aggregates;
5
6 procedure Main is
7     -- Remark: the following line is not relevant.
8     F : array (1 .. 10) of Float := (others => 42.42)
9     with Unreferenced;
10
11     type Test_Case_Index is
12         (Default_Rec_Chk,
13          Init_Rec_Chk,
14          Init_Some_Arr_Chk,
15          Init_Arr_Chk);
16
17     procedure Check (TC : Test_Case_Index) is
18         A : Int_Arr;
19         R : Rec;
20         DR : constant Rec := (others => <>);
21     begin
22         case TC is
23             when Default_Rec_Chk =>
24                 R := DR;
25                 Put_Line ("Record Default:");
26                 Put_Line ("W => " & Integer'Image (R.W));
27                 Put_Line ("X => " & Integer'Image (R.X));
28                 Put_Line ("Y => " & Integer'Image (R.Y));
29                 Put_Line ("Z => " & Integer'Image (R.Z));
30             when Init_Rec_Chk =>
31                 Init (R);
32                 Put_Line ("Record Init:");
33                 Put_Line ("W => " & Integer'Image (R.W));
34                 Put_Line ("X => " & Integer'Image (R.X));
35                 Put_Line ("Y => " & Integer'Image (R.Y));
36                 Put_Line ("Z => " & Integer'Image (R.Z));
```

(continues on next page)

(continued from previous page)

```

37     when Init_Some_Arr_Chk =>
38         Init_Some (A);
39         Put_Line ("Array Init_Some:");
40         for I in A'Range loop
41             Put_Line (Integer'Image (I) & " "
42                     & Integer'Image (A (I)));
43         end loop;
44     when Init_Arr_Chk =>
45         Init (A);
46         Put_Line ("Array Init:");
47         for I in A'Range loop
48             Put_Line (Integer'Image (I) & " "
49                     & Integer'Image (A (I)));
50         end loop;
51     end case;
52 end Check;
53
54 begin
55     if Argument_Count < 1 then
56         Put_Line ("ERROR: missing arguments! Exiting...");
57         return;
58     elsif Argument_Count > 1 then
59         Put_Line ("Ignoring additional arguments...");
60     end if;
61
62     Check (Test_Case_Index'Value (Argument (1)));
63 end Main;

```

7.2 Versioning

Goal: implement a simple package for source-code versioning.

Steps:

1. Implement the Versioning package.
 1. Declare the record type Version.
 2. Implement the Convert function that returns a string.
 3. Implement the Convert function that returns a floating-point number.

Requirements:

1. Record type Version has the following components of **Natural** type:
 1. Major,
 2. Minor, and
 3. Maintenance.
2. The first Convert function returns a string containing the version number.
3. The second Convert function returns a floating-point value.
 1. For this floating-point value:
 1. the number before the decimal point must correspond to the major number, and
 2. the number after the decimal point must correspond to the minor number.
 3. the maintenance number is ignored.
 2. For example, version "1.3.5" is converted to the floating-point value 1.3.

3. An obvious limitation of this function is that it can only handle one-digit numbers for the minor component.

- For example, we cannot convert version "1.10.0" to a reasonable value with the approach described above. The result of the call `Convert ((1, 10, 0))` is therefore unspecified.
- For the scope of this exercise, only version numbers with one-digit components are checked.

Remarks:

1. We use overloading for the `Convert` functions.
2. For the function `Convert` that returns a string, you can make use of the `Image_Trim` function, as indicated in the source-code below — see package body of `Versioning`.

Listing 82: `versioning.ads`

```
1 package Versioning is
2
3     -- type Version is record...
4
5     -- function Convert ...
6
7     -- function Convert
8
9 end Versioning;
```

Listing 83: `versioning.adb`

```
1 with Ada.Strings; use Ada.Strings;
2 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3
4 package body Versioning is
5
6     function Image_Trim (N : Natural) return String is
7         S_N : constant String := Trim (Natural'Image (N), Left);
8     begin
9         return S_N;
10    end Image_Trim;
11
12    -- function Convert ...
13    --     S_Major : constant String := Image_Trim (V.Major);
14    --     S_Minor : constant String := Image_Trim (V.Minor);
15    --     S_Maint : constant String := Image_Trim (V.Maintenance);
16    -- begin
17    -- end Convert;
18
19    -- function Convert ...
20    -- begin
21    -- end Convert;
22
23 end Versioning;
```

Listing 84: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Versioning; use Versioning;
5
6 procedure Main is
```

(continues on next page)

(continued from previous page)

```

7  type Test_Case_Index is
8      (Ver_String_Chk,
9       Ver_Float_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12      V : constant Version := (1, 3, 23);
13  begin
14      case TC is
15          when Ver_String_Chk =>
16              Put_Line (Convert (V));
17          when Ver_Float_Chk =>
18              Put_Line (Float'Image (Convert (V)));
19      end case;
20  end Check;
21
22  begin
23      if Argument_Count < 1 then
24          Put_Line ("ERROR: missing arguments! Exiting...");
25          return;
26      elsif Argument_Count > 1 then
27          Put_Line ("Ignoring additional arguments...");
28      end if;
29
30      Check (Test_Case_Index'Value (Argument (1)));
31  end Main;

```

7.3 Simple todo list

Goal: implement a simple to-do list system.

Steps:

1. Implement the `Todo_Lists` package.
 1. Declare the `Todo_Item` type.
 2. Declare the `Todo_List` type.
 3. Implement the `Add` procedure.
 4. Implement the `Display` procedure.

Requirements:

1. `Todo_Item` type is used to store a to-do item.
 1. It should be implemented as an access type to strings.
2. `Todo_Items` type is an array of to-do items.
 1. It should be implemented as an unconstrained array with positive range.
3. `Todo_List` type is the container for all to-do items.
 1. This record type must have a discriminant for the maximum number of elements of the list.
 2. In order to store the to-do items, it must contain a component named `Items` of `Todo_Items` type.
 3. Don't forget to keep track of the last element added to the list!
 - You should declare a `Last` component in the record.
4. Procedure `Add` adds items (of `Todo_Item` type) to the list (of `Todo_List` type).

1. This requires allocating a string for the access type.
2. An item can only be added to the list if the list isn't full yet — see next point for details on error handling.
5. Since the number of items that can be stored on the list is limited, the list might eventually become full in a call to Add.
 1. You must write code in the implementation of the Add procedure that verifies this condition.
 2. If the procedure detects that the list is full, it must display the following message: "ERROR: list is full!".
6. Procedure Display is used to display all to-do items.
 1. The header (first line) must be T0-**DO** LIST.
 2. It must display one item per line.

Remarks:

1. We use access types and unconstrained arrays in the implementation of the Todo_Lists package.

Listing 85: todo_lists.ads

```
1 package Todo_Lists is
2
3   -- Replace by actual type declaration
4   type Todo_Item is null record;
5
6   -- Replace by actual type declaration
7   type Todo_Items is null record;
8
9   -- Replace by actual type declaration
10  type Todo_List is null record;
11
12  procedure Add (Todos : in out Todo_List;
13                Item : String);
14
15  procedure Display (Todos : Todo_List);
16
17 end Todo_Lists;
```

Listing 86: todo_lists.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Todo_Lists is
4
5   procedure Add (Todos : in out Todo_List;
6                 Item : String) is
7   begin
8     Put_Line ("ERROR: list is full!");
9   end Add;
10
11  procedure Display (Todos : Todo_List) is
12  begin
13    null;
14  end Display;
15
16 end Todo_Lists;
```

Listing 87: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Todo_Lists;       use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List (10);
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreadsheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27         end case;
28     end Check;
29
30 begin
31     if Argument_Count < 1 then
32         Put_Line ("ERROR: missing arguments! Exiting...");
33         return;
34     elsif Argument_Count > 1 then
35         Put_Line ("Ignoring additional arguments...");
36     end if;
37
38     Check (Test_Case_Index'Value (Argument (1)));
39 end Main;

```

7.4 Price list

Goal: implement a list containing prices

Steps:

1. Implement the Price_Lists package.
 1. Declare the Price_Type type.
 2. Declare the Price_List record.
 3. Implement the Reset procedure.
 4. Implement the Add procedure.
 5. Implement the Get function.
 6. Implement the Display procedure.

Requirements:

1. Price_Type is a decimal fixed-point data type with a delta of two digits (e.g. 0.01) and twelve digits in total.
2. Price_List is a record type that contains the price list.
 1. This record type must have a discriminant for the maximum number of elements of the list.
3. Procedure Reset resets the list.
4. Procedure Add adds a price to the list.
 1. You should keep track of the last element added to the list.
5. Function Get retrieves a price from the list using an index.
 1. This function returns a record instance of Price_Result type.
 2. Price_Result is a variant record containing:
 1. the Boolean component Ok, and
 2. the component Price (of Price_Type).
 3. The returned value of Price_Result type is one of the following:
 1. If the index specified in a call to Get contains a valid (initialized) price, then
 - Ok is set to **True**, and
 - the Price component contains the price for that index.
 2. Otherwise:
 - Ok is set to **False**, and
 - the Price component is not available.
6. Procedure Display shows all prices from the list.
 1. The header (first line) must be PRICE LIST.
 2. The remaining lines contain one price per line.
 3. For example:
 - For the following code:

```
procedure Test is
  L : Price_List (10);
begin
  Reset (L);
  Add (L, 1.45);
  Add (L, 2.37);
  Display (L);
end Test;
```

- The output is:

```
PRICE LIST
1.45
2.37
```

Remarks:

1. To implement the package, you'll use the following features of the Ada language:
 1. decimal fixed-point types;
 2. records with discriminants;
 3. dynamically-sized record types;

4. variant records.
2. For record type `Price_List`, you may use an unconstrained array as a component of the record and use the discriminant in the component declaration.

Listing 88: `price_lists.ads`

```

1 package Price_Lists is
2
3   -- Replace by actual type declaration
4   type Price_Type is new Float;
5
6   -- Replace by actual type declaration
7   type Price_List is null record;
8
9   -- Replace by actual type declaration
10  type Price_Result is null record;
11
12  procedure Reset (Prices : in out Price_List);
13
14  procedure Add (Prices : in out Price_List;
15               Item   : Price_Type);
16
17  function Get (Prices : Price_List;
18              Idx     : Positive) return Price_Result;
19
20  procedure Display (Prices : Price_List);
21
22 end Price_Lists;

```

Listing 89: `price_lists.adb`

```

1 package body Price_Lists is
2
3   procedure Reset (Prices : in out Price_List) is
4   begin
5     null;
6   end Reset;
7
8   procedure Add (Prices : in out Price_List;
9                Item   : Price_Type) is
10  begin
11    null;
12  end Add;
13
14  function Get (Prices : Price_List;
15              Idx     : Positive) return Price_Result is
16  begin
17    null;
18  end Get;
19
20  procedure Display (Prices : Price_List) is
21  begin
22    null;
23  end Display;
24
25 end Price_Lists;

```

Listing 90: `main.adb`

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;

```

(continues on next page)

(continued from previous page)

```

3
4 with Price_Lists;          use Price_Lists;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Price_Type_Chk,
9      Price_List_Chk,
10     Price_List_Get_Chk);
11
12   procedure Check (TC : Test_Case_Index) is
13     L : Price_List (10);
14
15     procedure Local_Init_List is
16     begin
17       Reset (L);
18       Add (L, 1.45);
19       Add (L, 2.37);
20       Add (L, 3.21);
21       Add (L, 4.14);
22       Add (L, 5.22);
23       Add (L, 6.69);
24       Add (L, 7.77);
25       Add (L, 8.14);
26       Add (L, 9.99);
27       Add (L, 10.01);
28     end Local_Init_List;
29
30     procedure Get_Display (Idx : Positive) is
31       R : constant Price_Result := Get (L, Idx);
32     begin
33       Put_Line ("Attempt Get # " & Positive'Image (Idx));
34       if R.Ok then
35         Put_Line ("Element # " & Positive'Image (Idx)
36           & " => " & Price_Type'Image (R.Price));
37       else
38         declare
39         begin
40           Put_Line ("Element # " & Positive'Image (Idx)
41             & " => " & Price_Type'Image (R.Price));
42         exception
43           when others =>
44             Put_Line ("Element not available (as expected)");
45         end;
46       end if;
47     end Get_Display;
48
49   begin
50     case TC is
51     when Price_Type_Chk =>
52       Put_Line ("The delta value of Price_Type is "
53         & Price_Type'Image (Price_Type'Delta) & ";");
54       Put_Line ("The minimum value of Price_Type is "
55         & Price_Type'Image (Price_Type'First) & ";");
56       Put_Line ("The maximum value of Price_Type is "
57         & Price_Type'Image (Price_Type'Last) & ";");
58     when Price_List_Chk =>
59       Local_Init_List;
60       Display (L);
61     when Price_List_Get_Chk =>
62       Local_Init_List;
63

```

(continues on next page)

(continued from previous page)

```
64         Get_Display (5);
65         Get_Display (40);
66     end case;
67 end Check;
68
69 begin
70     if Argument_Count < 1 then
71         Put_Line ("ERROR: missing arguments! Exiting...");
72         return;
73     elsif Argument_Count > 1 then
74         Put_Line ("Ignoring additional arguments...");
75     end if;
76
77     Check (Test_Case_Index'Value (Argument (1)));
78 end Main;
```


PRIVACY

8.1 Directions

Goal: create a package that handles directions and geometric angles using a previous implementation.

Steps:

1. Fix the implementation of the `Test_Directions` procedure.

Requirements:

1. The implementation of the `Test_Directions` procedure must compile correctly.

Remarks:

1. This exercise is based on the *Directions* exercise from the [Records](#) (page 31) labs.
 1. In this version, however, `Ext_Angle` is a private type.
2. In the implementation of the `Test_Directions` procedure below, the Ada developer tried to initialize `All_Directions` — an array of `Ext_Angle` type — with aggregates.
 1. Since we now have a private type, the compiler complains about this initialization.
3. To fix the implementation of the `Test_Directions` procedure, you should use the appropriate function from the `Directions` package.
4. The initialization of `All_Directions` in the code below contains a consistency error where the angle doesn't match the assessed direction.
 1. See if you can spot this error!
 2. This kind of errors can happen when record components that have correlated information are initialized individually without consistency checks — using private types helps to avoid the problem by requiring initialization routines that can enforce consistency.

Listing 91: `directions.ads`

```
1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northwest,
8      West,
9      Southwest,
10     South,
11     Southeast,
12     East);
13
```

(continues on next page)

(continued from previous page)

```

14  function To_Direction (N : Angle_Mod) return Direction;
15
16  type Ext_Angle is private;
17
18  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
19
20  procedure Display (N : Ext_Angle);
21
22 private
23
24  type Ext_Angle is record
25      Angle_Elem      : Angle_Mod;
26      Direction_Elem  : Direction;
27  end record;
28
29 end Directions;

```

Listing 92: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          Put_Line ("Angle: "
8                  & Angle_Mod'Image (N.Angle_Elem)
9                  & " => "
10                 & Direction'Image (N.Direction_Elem)
11                 & ".");
12      end Display;
13
14      function To_Direction (N : Angle_Mod) return Direction is
15      begin
16          case N is
17              when 0      => return East;
18              when 1 .. 89 => return Northwest;
19              when 90     => return North;
20              when 91 .. 179 => return Northwest;
21              when 180    => return West;
22              when 181 .. 269 => return Southwest;
23              when 270    => return South;
24              when 271 .. 359 => return Southeast;
25          end case;
26      end To_Direction;
27
28      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29      begin
30          return (Angle_Elem      => N,
31                 Direction_Elem => To_Direction (N));
32      end To_Ext_Angle;
33
34 end Directions;

```

Listing 93: test_directions.adb

```

1  with Directions; use Directions;
2
3  procedure Test_Directions is
4      type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;

```

(continues on next page)

(continued from previous page)

```

5
6  All_Directions : constant Ext_Angle_Array (1 .. 6)
7      := ((0,   East),
8          (45,  Northwest),
9          (90,  North),
10         (91,  North),
11         (180, West),
12         (270, South));
13
14 begin
15     for I in All_Directions'Range loop
16         Display (All_Directions (I));
17     end loop;
18
19 end Test_Directions;

```

Listing 94: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when Direction_Chk =>
14                 Test_Directions;
15         end case;
16     end Check;
17
18     begin
19         if Argument_Count < 1 then
20             Put_Line ("ERROR: missing arguments! Exiting...");
21             return;
22         elsif Argument_Count > 1 then
23             Put_Line ("Ignoring additional arguments...");
24         end if;
25
26         Check (Test_Case_Index'Value (Argument (1)));
27     end Main;

```

8.2 Limited Strings

Goal: work with **limited private** types.

Steps:

1. Implement the Limited_Strings package.
 1. Implement the Copy function.
 2. Implement the = operator.

Requirements:

1. For both Copy and =, the two parameters may refer to strings with different lengths.

We'll limit the implementation to just take the minimum length:

1. In case of copying the string "Hello World" to a string with 5 characters, the copied string is "Hello":

```
S1 : constant Lim_String := Init ("Hello World");
S2 :          Lim_String := Init (5);
begin
  Copy (From => S1, To => S2);
  Put_Line (S2);      -- This displays "Hello".
```

2. When comparing "Hello World" to "Hello", the = operator indicates that these strings are equivalent:

```
S1 : constant Lim_String := Init ("Hello World");
S2 : constant Lim_String := Init ("Hello");
begin
  if S1 = S2 then
    -- True => This branch gets selected.
```

2. When copying from a short string to a longer string, the remaining characters of the longer string must be initialized with underscores (_). For example:

```
S1 : constant Lim_String := Init ("Hello");
S2 :          Lim_String := Init (10);
begin
  Copy (From => S1, To => S2);
  Put_Line (S2);      -- This displays "Hello_____".
```

Remarks:

1. As we've discussed in the course:
 1. Variables of limited types have the following limitations:
 - they cannot be assigned to;
 - they don't have an equality operator (=).
 2. We can, however, define our own, custom subprograms to circumvent these limitations:
 - In order to copy instances of a limited type, we can define a custom Copy procedure.
 - In order to compare instances of a limited type, we can define an = operator.
2. You can use the `Min_Last` constant — which is already declared in the implementation of these subprograms — in the code you write.
3. Some details about the `Limited_Strings` package:
 1. The `Lim_String` type acts as a container for strings.
 1. In the the private part, `Lim_String` is declared as an access type to a **String**.
 2. There are two versions of the `Init` function that initializes an object of `Lim_String` type:
 1. The first one takes another string.
 2. The second one receives the number of characters for a string *container*.
 3. Procedure `Put_Line` displays object of `Lim_String` type.
 4. The design and implementation of the `Limited_Strings` package is very simplistic.
 1. A good design would have better handling of access types, for example.

Listing 95: limited_strings.ads

```

1 package Limited_Strings is
2
3     type Lim_String is limited private;
4
5     function Init (S : String) return Lim_String;
6
7     function Init (Max : Positive) return Lim_String;
8
9     procedure Put_Line (LS : Lim_String);
10
11    procedure Copy (From : Lim_String;
12                  To   : in out Lim_String);
13
14    function "=" (Ref, Dut : Lim_String) return Boolean;
15
16 private
17
18     type Lim_String is access String;
19
20 end Limited_Strings;
```

Listing 96: limited_strings.adb

```

1 with Ada.Text_IO;
2
3 package body Limited_Strings
4 is
5
6     function Init (S : String) return Lim_String is
7         LS : constant Lim_String := new String'(S);
8     begin
9         return Ls;
10    end Init;
11
12    function Init (Max : Positive) return Lim_String is
13        LS : constant Lim_String := new String (1 .. Max);
14    begin
15        LS.all := (others => '_');
16        return LS;
17    end Init;
18
19    procedure Put_Line (LS : Lim_String) is
20    begin
21        Ada.Text_IO.Put_Line (LS.all);
22    end Put_Line;
23
24    function Get_Min_Last (A, B : Lim_String) return Positive is
25    begin
26        return Positive'Min (A'Last, B'Last);
27    end Get_Min_Last;
28
29    procedure Copy (From : Lim_String;
30                  To   : in out Lim_String) is
31        Min_Last : constant Positive := Get_Min_Last (From, To);
32    begin
33        -- Complete the implementation!
34        null;
35    end;
```

(continues on next page)

(continued from previous page)

```
37 function "=" (Ref, Dut : Lim_String) return Boolean is
38   Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
39 begin
40   -- Complete the implementation!
41   return True;
42 end;
43
44 end Limited_Strings;
```

Listing 97: check_lim_string.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Limited_Strings; use Limited_Strings;
4
5 procedure Check_Lim_String is
6   S : constant String := "-----";
7   S1 : constant Lim_String := Init ("Hello World");
8   S2 : constant Lim_String := Init (30);
9   S3 : Lim_String := Init (5);
10  S4 : Lim_String := Init (S & S & S);
11 begin
12   Put ("S1 => ");
13   Put_Line (S1);
14   Put ("S2 => ");
15   Put_Line (S2);
16
17   if S1 = S2 then
18     Put_Line ("S1 is equal to S2.");
19   else
20     Put_Line ("S1 isn't equal to S2.");
21   end if;
22
23   Copy (From => S1, To => S3);
24   Put ("S3 => ");
25   Put_Line (S3);
26
27   if S1 = S3 then
28     Put_Line ("S1 is equal to S3.");
29   else
30     Put_Line ("S1 isn't equal to S3.");
31   end if;
32
33   Copy (From => S1, To => S4);
34   Put ("S4 => ");
35   Put_Line (S4);
36
37   if S1 = S4 then
38     Put_Line ("S1 is equal to S4.");
39   else
40     Put_Line ("S1 isn't equal to S4.");
41   end if;
42 end Check_Lim_String;
```

Listing 98: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Check_Lim_String;
```

(continues on next page)

(continued from previous page)

```

5
6 procedure Main is
7   type Test_Case_Index is
8     (Lim_String_Chk);
9
10  procedure Check (TC : Test_Case_Index) is
11  begin
12    case TC is
13    when Lim_String_Chk =>
14      Check_Lim_String;
15    end case;
16  end Check;
17
18 begin
19   if Argument_Count < 1 then
20     Put_Line ("ERROR: missing arguments! Exiting...");
21     return;
22   elsif Argument_Count > 1 then
23     Put_Line ("Ignoring additional arguments...");
24   end if;
25
26   Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

8.3 Bonus exercise

In previous labs, we had many source-code snippets containing records that could be declared private. The source-code for the exercise above (*Directions*) is an example: we've modified the type declaration of `Ext_Angle`, so that the record is now private. Encapsulating the record components — by declaring record components in the private part — makes the code safer. Also, because many of the code snippets weren't making use of record components directly (but handling record types via the API instead), they continue to work fine after these modifications.

This exercise doesn't contain any source-code. In fact, the **goal** here is to modify previous labs, so that the record declarations are made private. You can look into those labs, modify the type declarations, and recompile the code. The corresponding test-cases must still pass.

If no other changes are needed apart from changes in the declaration, then that indicates we have used good programming techniques in the original code. On the other hand, if further changes are needed, then you should investigate why this is the case.

Also note that, in some cases, you can move support types into the private part of the specification without affecting its compilation. This is the case, for example, for the `People_Array` type of the *List of Names* lab mentioned below. You should, in fact, keep only relevant types and subprograms in the public part and move all support declarations to the private part of the specification whenever possible.

Below, you find the selected labs that you can work on, including changes that you should make. In case you don't have a working version of the source-code of previous labs, you can look into the corresponding solutions.

8.3.1 Colors

Chapter: *Records* (page 31)

Steps:

1. Change declaration of RGB type to **private**.

Requirements:

1. Implementation must compile correctly and test cases must pass.

8.3.2 List of Names

Chapter: *Arrays* (page 41)

Steps:

1. Change declaration of Person and People types to **limited private**.
2. Move type declaration of People_Array to private part.

Requirements:

1. Implementation must compile correctly and test cases must pass.

8.3.3 Price List

Chapter: *More About Types* (page 59)

Steps:

1. Change declaration of Price_List type to **limited private**.

Requirements:

1. Implementation must compile correctly and test cases must pass.

GENERICCS

9.1 Display Array

Goal: create a generic procedure that displays the elements of an array.

Steps:

1. Implement the generic procedure `Display_Array`.

Requirements:

1. Generic procedure `Display_Array` displays the elements of an array.
 1. It uses the following scheme:
 - First, it displays a header.
 - Then, it displays the elements of the array.
 2. When displaying the elements, it must:
 - use one line per element, and
 - include the corresponding index of the array.
 3. This is the expected format:

```
<HEADER>
<index #1>: <element #1>
<index #2>: <element #2>
...
```

4. For example:

- For the following code:

```
procedure Test is
  A : Int_Array (1 .. 2) := (1, 5);
begin
  Display_Int_Array ("Elements of A", A);
end Test;
```

- The output is:

```
Elements of A
1: 1
2: 5
```

2. These are the formal parameters of the procedure:
 1. a range type `T_Range` for the the array;
 2. a formal type `T_Element` for the elements of the array;

- This type must be declared in such a way that it can be mapped to any type in the instantiation — including record types.
3. an array type `T_Array` using the `T_Range` and `T_Element` types;
 4. a function `Image` that converts a variable of `T_Element` type to a **String**.

Listing 99: `display_array.ads`

```
1 generic
2 procedure Display_Array (Header : String;
3                        A       : T_Array);
```

Listing 100: `display_array.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Array (Header : String;
4                        A       : T_Array) is
5 begin
6     null;
7 end Display_Array;
```

Listing 101: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Array;
5
6 procedure Main is
7     type Test_Case_Index is (Int_Array_Chk,
8                             Point_Array_Chk);
9
10    procedure Test_Int_Array is
11        type Int_Array is array (Positive range <>) of Integer;
12
13        procedure Display_Int_Array is new
14            Display_Array (T_Range => Positive,
15                          T_Element => Integer,
16                          T_Array  => Int_Array,
17                          Image    => Integer'Image);
18
19        A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
20    begin
21        Display_Int_Array ("Integers", A);
22    end Test_Int_Array;
23
24    procedure Test_Point_Array is
25        type Point is record
26            X : Float;
27            Y : Float;
28        end record;
29
30        type Point_Array is array (Natural range <>) of Point;
31
32        function Image (P : Point) return String is
33        begin
34            return "(" & Float'Image (P.X)
35                & ", " & Float'Image (P.Y) & ")";
36        end Image;
37
```

(continues on next page)

(continued from previous page)

```

38     procedure Display_Point_Array is new
39         Display_Array (T_Range  => Natural,
40                       T_Element => Point,
41                       T_Array   => Point_Array,
42                       Image     => Image);
43
44     A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
45                                           (5.0, 2.0), (-0.5, 2.0));
46
47     begin
48         Display_Point_Array ("Points", A);
49     end Test_Point_Array;
50
51     procedure Check (TC : Test_Case_Index) is
52     begin
53         case TC is
54             when Int_Array_Chk =>
55                 Test_Int_Array;
56             when Point_Array_Chk =>
57                 Test_Point_Array;
58         end case;
59     end Check;
60
61     begin
62         if Argument_Count < 1 then
63             Put_Line ("ERROR: missing arguments! Exiting...");
64             return;
65         elsif Argument_Count > 1 then
66             Put_Line ("Ignoring additional arguments...");
67         end if;
68
69         Check (Test_Case_Index'Value (Argument (1)));
70     end Main;

```

9.2 Average of Array of Float

Goal: create a generic function that calculates the average of an array of floating-point elements.

Steps:

1. Declare and implement the generic function Average.

Requirements:

1. Generic function Average calculates the average of an array containing floating-point values of arbitrary precision.
2. Generic function Average must contain the following formal parameters:
 1. a range type T_Range for the array;
 2. a formal type T_Element that can be mapped to floating-point types of arbitrary precision;
 3. an array type T_Array using T_Range and T_Element;

Remarks:

1. You should use the **Float** type for the accumulator.

Listing 102: average.ads

```
1 generic
2 function Average (A : T_Array) return T_Element;
```

Listing 103: average.adb

```
1 function Average (A : T_Array) return T_Element is
2 begin
3     return 0.0;
4 end Average;
```

Listing 104: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Average;
5
6 procedure Main is
7     type Test_Case_Index is (Float_Array_Chk,
8                               Digits_7_Float_Array_Chk);
9
10    procedure Test_Float_Array is
11        type Float_Array is array (Positive range <>) of Float;
12
13        function Average_Float is new
14            Average (T_Range => Positive,
15                    T_Element => Float,
16                    T_Array => Float_Array);
17
18        A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
19    begin
20        Put_Line ("Average: " & Float'Image (Average_Float (A)));
21    end Test_Float_Array;
22
23    procedure Test_Digits_7_Float_Array is
24        type Custom_Float is digits 7 range 0.0 .. 1.0;
25
26        type Float_Array is
27            array (Integer range <>) of Custom_Float;
28
29        function Average_Float is new
30            Average (T_Range => Integer,
31                    T_Element => Custom_Float,
32                    T_Array => Float_Array);
33
34        A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
35    begin
36        Put_Line ("Average: "
37                  & Custom_Float'Image (Average_Float (A)));
38    end Test_Digits_7_Float_Array;
39
40    procedure Check (TC : Test_Case_Index) is
41    begin
42        case TC is
43            when Float_Array_Chk =>
44                Test_Float_Array;
45            when Digits_7_Float_Array_Chk =>
46                Test_Digits_7_Float_Array;
47        end case;
```

(continues on next page)

(continued from previous page)

```

48     end Check;
49
50 begin
51     if Argument_Count < 1 then
52         Put_Line ("ERROR: missing arguments! Exiting...");
53         return;
54     elsif Argument_Count > 1 then
55         Put_Line ("Ignoring additional arguments...");
56     end if;
57
58     Check (Test_Case_Index'Value (Argument (1)));
59 end Main;

```

9.3 Average of Array of Any Type

Goal: create a generic function that calculates the average of an array of elements of any arbitrary type.

Steps:

1. Declare and implement the generic function Average.
2. Implement the test procedure Test_Item.
 1. Declare the F_IO package.
 2. Implement the Get_Total function for the Item type.
 3. Implement the Get_Price function for the Item type.
 4. Declare the Average_Total function.
 5. Declare the Average_Price function.

Requirements:

1. Generic function Average calculates the average of an array containing elements of any arbitrary type.
2. Generic function Average has the same formal parameters as in the previous exercise, except for:
 1. T_Element, which is now a formal type that can be mapped to any arbitrary type.
 2. To_Float, which is an *additional* formal parameter.
 - To_Float is a function that converts the arbitrary element of T_Element type to the **Float** type.
3. Procedure Test_Item is used to test the generic Average procedure for a record type (Item).
 1. Record type Item contains the Quantity and Price components.
4. The following functions have to be implemented to be used for the formal To_Float function parameter:
 1. For the Decimal type, the function is pretty straightforward: it simply returns the floating-point value converted from the decimal type.
 2. For the Item type, two functions must be created to convert to floating-point type:
 1. Get_Total, which returns the multiplication of the quantity and the price components of the Item type;
 2. Get_Price, which returns just the price.

5. The generic function `Average` must be instantiated as follows:

1. For the `Item` type, you must:

1. declare the `Average_Total` function (as an instance of `Average`) using the `Get_Total` for the `To_Float` parameter;
2. declare the `Average_Price` function (as an instance of `Average`) using the `Get_Price` for the `To_Float` parameter.

6. You must use the `Put` procedure from `Ada.Text_IO.Float_IO`.

1. The generic standard package `Ada.Text_IO.Float_IO` must be instantiated as `F_IO` in the test procedures.
2. This is the specification of the `Put` procedure, as described in the appendix A.10.9 of the Ada Reference Manual:

```
procedure Put(Item : in Num;  
             Fore : in Field := Default_Fore;  
             Aft  : in Field := Default_Aft;  
             Exp  : in Field := Default_Exp);
```

3. This is the expected format when calling `Put` from `Float_IO`:

Function	Fore	Aft	Exp
Test_Item	3	2	0

Remarks:

1. In this exercise, you'll abstract the `Average` function from the previous exercises a step further.
 1. In this case, the function shall be able to calculate the average of any arbitrary type — including arrays containing elements of record types.
 2. Since record types can be composed by many components of different types, we need to provide a way to indicate which component (or components) of the record will be used when calculating the average of the array.
 3. This problem is solved by specifying a `To_Float` function as a formal parameter, which converts the arbitrary element of `T_Element` type to the **Float** type.
 4. In the implementation of the `Average` function, we use the `To_Float` function and calculate the average using a floating-point variable.

Listing 105: `average.ads`

```
1 generic  
2 function Average (A : T_Array) return Float;
```

Listing 106: `average.adb`

```
1 function Average (A : T_Array) return Float is  
2 begin  
3   null;  
4 end Average;
```

Listing 107: `test_item.ads`

```
1 procedure Test_Item;
```

Listing 108: test_item.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Average;
4
5  procedure Test_Item is
6      type Amount is delta 0.01 digits 12;
7
8      type Item is record
9          Quantity : Natural;
10         Price    : Amount;
11     end record;
12
13     type Item_Array is
14         array (Positive range <>) of Item;
15
16     A : constant Item_Array (1 .. 4)
17         := ((Quantity => 5,   Price => 10.00),
18            (Quantity => 80,  Price => 2.50),
19            (Quantity => 40,  Price => 5.00),
20            (Quantity => 20,  Price => 12.50));
21
22     begin
23         Put ("Average per item & quantity: ");
24         F_IO.Put (Average_Total (A));
25         New_Line;
26
27         Put ("Average price:                ");
28         F_IO.Put (Average_Price (A));
29         New_Line;
30     end Test_Item;

```

Listing 109: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Item;
5
6  procedure Main is
7      type Test_Case_Index is (Item_Array_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10     begin
11         case TC is
12             when Item_Array_Chk =>
13                 Test_Item;
14         end case;
15     end Check;
16
17     begin
18         if Argument_Count < 1 then
19             Put_Line ("ERROR: missing arguments! Exiting...");
20             return;
21         elsif Argument_Count > 1 then
22             Put_Line ("Ignoring additional arguments...");
23         end if;
24
25         Check (Test_Case_Index'Value (Argument (1)));
26     end Main;

```

9.4 Generic list

Goal: create a system based on a generic list to add and displays elements.

Steps:

1. Declare and implement the generic package `Gen_List`.
 1. Implement the `Init` procedure.
 2. Implement the `Add` procedure.
 3. Implement the `Display` procedure.

Requirements:

1. Generic package `Gen_List` must have the following subprograms:
 1. Procedure `Init` initializes the list.
 2. Procedure `Add` adds an item to the list.
 1. This procedure must contain a `Status` output parameter that is set to **False** when the list was full — i.e. if the procedure failed while trying to add the item;
 3. Procedure `Display` displays the complete list.
 1. This includes the *name* of the list and its elements — using one line per element.
 2. This is the expected format:

```
<NAME>  
<element #1>  
<element #2>  
...
```

2. Generic package `Gen_List` has these formal parameters:
 1. an arbitrary formal type `Item`;
 2. an unconstrained array type `Items` of `Item` element with positive range;
 3. the `Name` parameter containing the name of the list;
 - This must be a formal input object of **String** type.
 - It must be used in the `Display` procedure.
 4. an actual array `List_Array` to store the list;
 - This must be a formal **in out** object of `Items` type.
 5. the variable `Last` to store the index of the last element;
 - This must be a formal **in out** object of **Natural** type.
 6. a procedure `Put` for the `Item` type.
 - This procedure is used in the `Display` procedure to display individual elements of the list.
3. The test procedure `Test_Int` is used to test a list of elements of **Integer** type.
4. For both test procedures, you must:
 1. add missing type declarations;
 2. declare and implement a `Put` procedure for individual elements of the list;
 3. declare instances of the `Gen_List` package.
 - For the `Test_Int` procedure, declare the `Int_List` package.

Remarks:

1. In previous labs, you've been implementing lists for a variety of types.
 - The *List of Names* exercise from the [Arrays](#) (page 41) labs is an example.
 - In this exercise, you have to abstract those implementations to create the generic `Gen_List` package.

Listing 110: `gen_list.ads`

```

1 generic
2 package Gen_List is
3
4     procedure Init;
5
6     procedure Add (I      : Item;
7                   Status : out Boolean);
8
9     procedure Display;
10
11 end Gen_List;
```

Listing 111: `gen_list.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Gen_List is
4
5     procedure Init is
6     begin
7         null;
8     end Init;
9
10    procedure Add (I      : Item;
11                  Status : out Boolean) is
12    begin
13        null;
14    end Add;
15
16    procedure Display is
17    begin
18        null;
19    end Display;
20
21 end Gen_List;
```

Listing 112: `test_int.ads`

```

1 procedure Test_Int;
```

Listing 113: `test_int.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_List;
4
5 procedure Test_Int is
6
7     type Integer_Array is array (Positive range <>) of Integer;
8
9     A : Integer_Array (1 .. 3);
```

(continues on next page)

(continued from previous page)

```

10  L : Natural;
11
12  Success : Boolean;
13
14  procedure Display_Add_Success (Success : Boolean) is
15  begin
16      if Success then
17          Put_Line ("Added item successfully!");
18      else
19          Put_Line ("Couldn't add item!");
20      end if;
21
22  end Display_Add_Success;
23
24  begin
25      Int_List.Init;
26
27      Int_List.Add (2, Success);
28      Display_Add_Success (Success);
29
30      Int_List.Add (5, Success);
31      Display_Add_Success (Success);
32
33      Int_List.Add (7, Success);
34      Display_Add_Success (Success);
35
36      Int_List.Add (8, Success);
37      Display_Add_Success (Success);
38
39      Int_List.Display;
40  end Test_Int;

```

Listing 114: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Test_Int;
5
6  procedure Main is
7      type Test_Case_Index is (Int_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10     begin
11         case TC is
12             when Int_Chk =>
13                 Test_Int;
14         end case;
15     end Check;
16
17     begin
18         if Argument_Count < 1 then
19             Put_Line ("ERROR: missing arguments! Exiting...");
20             return;
21         elsif Argument_Count > 1 then
22             Put_Line ("Ignoring additional arguments...");
23         end if;
24
25         Check (Test_Case_Index'Value (Argument (1)));
26     end Main;

```

EXCEPTIONS

10.1 Uninitialized Value

Goal: implement an enumeration to avoid the use of uninitialized values.

Steps:

1. Implement the Options package.
 1. Declare the Option enumeration type.
 2. Declare the Uninitialized_Value exception.
 3. Implement the Image function.

Requirements:

1. Enumeration Option contains:
 1. the Uninitialized value, and
 2. the actual options:
 - Option_1,
 - Option_2,
 - Option_3.
2. Function Image returns a string for the Option type.
 1. In case the argument to Image is Uninitialized, the function must raise the Uninitialized_Value exception.

Remarks:

1. In this exercise, we employ exceptions as a mechanism to avoid the use of uninitialized values for a certain type.

Listing 115: options.ads

```
1 package Options is
2
3   -- Declare the Option enumeration type!
4   type Option is null record;
5
6   function Image (O : Option) return String;
7
8 end Options;
```


Listing 116: options.adb

```
1 package body Options is
2
3     function Image (O : Option) return String is
4     begin
5         return "";
6     end Image;
7
8 end Options;
```

Listing 117: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with Ada.Exceptions;    use Ada.Exceptions;
4
5 with Options;           use Options;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Options_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Check (O : Option) is
14         begin
15             Put_Line (Image (O));
16         exception
17             when E : Uninitialized_Value =>
18                 Put_Line (Exception_Message (E));
19         end Check;
20
21     begin
22         case TC is
23             when Options_Chk =>
24                 for O in Option loop
25                     Check (O);
26                 end loop;
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;
```

10.2 Numerical Exception

Goal: handle numerical exceptions in a test procedure.

Steps:

1. Add exception handling to the Check_Exception procedure.

Requirements:

1. The test procedure Num_Exception_Test from the Tests package below must be used in the implementation of Check_Exception.
2. The Check_Exception procedure must be extended to handle exceptions as follows:
 1. If the exception raised by Num_Exception_Test is Constraint_Error, the procedure must display the message "Constraint_Error detected!" to the user.
 2. Otherwise, it must display the message associated with the exception.

Remarks:

1. You can use the Exception_Message function to retrieve the message associated with an exception.

Listing 118: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;
```

Listing 119: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10        A := Integer'Last;
11        B := Integer'Last;
12        C := A + B;
13       when Test_2 =>
14        raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;
```

Listing 120: check_exception.adb

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID) is
4 begin
5   Num_Exception_Test (ID);
6 end Check_Exception;
```

Listing 121: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```

3  with Ada.Exceptions;    use Ada.Exceptions;
4
5  with Tests;             use Tests;
6  with Check_Exception;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14
15         procedure Check_Handle_Exception (ID : Test_ID) is
16             begin
17                 Check_Exception (ID);
18             exception
19                 when Constraint_Error =>
20                     Put_Line ("Constraint_Error"
21                               & " (raised by Check_Exception) detected!");
22                 when E : others =>
23                     Put_Line (Exception_Name (E)
24                               & " (raised by Check_Exception) detected!");
25             end Check_Handle_Exception;
26
27         begin
28             case TC is
29                 when Exception_1_Chk =>
30                     Check_Handle_Exception (Test_1);
31                 when Exception_2_Chk =>
32                     Check_Handle_Exception (Test_2);
33             end case;
34         end Check;
35
36     begin
37         if Argument_Count < 1 then
38             Put_Line ("ERROR: missing arguments! Exiting...");
39             return;
40         elsif Argument_Count > 1 then
41             Put_Line ("Ignoring additional arguments...");
42         end if;
43
44         Check (Test_Case_Index'Value (Argument (1)));
45     end Main;

```

10.3 Re-raising Exceptions

Goal: make use of exception re-raising in a test procedure.

Steps:

1. Declare new exception: Another_Exception.
2. Add exception re-raise to the Check_Exception procedure.

Requirements:

1. Exception Another_Exception must be declared in the Tests package.
2. Procedure Check_Exception must be extended to *re-raise* any exception. When an exception is detected, the procedure must:
 1. display a user message (as implemented in the previous exercise), and then

2. Raise or *re-raise* exception depending on the exception that is being handled:
 1. In case of `Constraint_Error` exception, *re-raise* the exception.
 2. In all other cases, raise `Another_Exception`.

Remarks:

1. In this exercise, you should extend the implementation of the `Check_Exception` procedure from the previous exercise.
1. Naturally, you can use the code for the `Check_Exception` procedure from the previous exercise as a starting point.

Listing 122: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;
```

Listing 123: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10        A := Integer'Last;
11        B := Integer'Last;
12        C := A + B;
13       when Test_2 =>
14        raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;
```

Listing 124: check_exception.ads

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID);
```

Listing 125: check_exception.adb

```

1 procedure Check_Exception (ID : Test_ID) is
2 begin
3   Num_Exception_Test (ID);
4 end Check_Exception;
```

Listing 126: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with Ada.Exceptions;    use Ada.Exceptions;
4
5 with Tests;             use Tests;
6 with Check_Exception;
7
8 procedure Main is
9     type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14
15         procedure Check_Handle_Exception (ID : Test_ID) is
16             begin
17                 Check_Exception (ID);
18             exception
19                 when Constraint_Error =>
20                     Put_Line ("Constraint_Error"
21                               & " (raised by Check_Exception) detected!");
22                 when E : others =>
23                     Put_Line (Exception_Name (E)
24                               & " (raised by Check_Exception) detected!");
25             end Check_Handle_Exception;
26
27             begin
28                 case TC is
29                     when Exception_1_Chk =>
30                         Check_Handle_Exception (Test_1);
31                     when Exception_2_Chk =>
32                         Check_Handle_Exception (Test_2);
33                     end case;
34             end Check;
35
36 begin
37     if Argument_Count < 1 then
38         Put_Line ("ERROR: missing arguments! Exiting...");
39         return;
40     elsif Argument_Count > 1 then
41         Put_Line ("Ignoring additional arguments...");
42     end if;
43
44     Check (Test_Case_Index'Value (Argument (1)));
45 end Main;
```

TASKING

11.1 Display Service

Goal: create a simple service that displays messages to the user.

Steps:

1. Implement the `Display_Services` package.
 1. Declare the task type `Display_Service`.
 2. Implement the `Display` entry for strings.
 3. Implement the `Display` entry for integers.

Requirements:

1. Task type `Display_Service` uses the `Display` entry to display messages to the user.
2. There are two versions of the `Display` entry:
 1. One that receives messages as a string parameter.
 2. One that receives messages as an **Integer** parameter.
3. When a message is received via a `Display` entry, it must be displayed immediately to the user.

Listing 127: `display_services.ads`

```
1 package Display_Services is
2
3 end Display_Services;
```

Listing 128: `display_services.adb`

```
1 package body Display_Services is
2
3 end Display_Services;
```

Listing 129: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Services; use Display_Services;
5
6 procedure Main is
7     type Test_Case_Index is (Display_Service_Chk);
8
9     procedure Check (TC : Test_Case_Index) is
10         Display : Display_Service;
```

(continues on next page)

(continued from previous page)

```

11  begin
12      case TC is
13          when Display_Service_Chk =>
14              Display.Display ("Hello");
15              delay 0.5;
16              Display.Display ("Hello again");
17              delay 0.5;
18              Display.Display (55);
19              delay 0.5;
20          end case;
21  end Check;
22
23  begin
24      if Argument_Count < 1 then
25          Put_Line ("ERROR: missing arguments! Exiting...");
26          return;
27      elsif Argument_Count > 1 then
28          Put_Line ("Ignoring additional arguments...");
29      end if;
30
31      Check (Test_Case_Index'Value (Argument (1)));
32  end Main;

```

11.2 Event Manager

Goal: implement a simple event manager.

Steps:

1. Implement the Event_Managers package.
 1. Declare the task type Event_Manager.
 2. Implement the Start entry.
 3. Implement the Event entry.

Requirements:

1. The event manager has a similar behavior as an alarm
 1. The sole purpose of this event manager is to display the event ID at the correct time.
 2. After the event ID is displayed, the task must finish.
2. The event manager (Event_Manager type) must have two entries:
 1. Start, which starts the event manager with an event ID;
 2. Event, which delays the task until a certain time and then displays the event ID as a user message.
3. The format of the user message displayed by the event manager is Event #<event_id>.
 1. You should use Natural'Image to display the ID (as indicated in the body of the Event_Managers package below).

Remarks:

1. In the Start entry, you can use the **Natural** type for the ID.
2. In the Event entry, you should use the Time type from the Ada.Real_Time package for the time parameter.

3. Note that the test application below creates an array of event managers with different delays.

Listing 130: event_managers.ads

```

1 package Event_Managers is
2
3 end Event_Managers;
```

Listing 131: event_managers.adb

```

1 package body Event_Managers is
2
3     -- Don't forget to display the event ID:
4     --
5     -- Put_Line ("Event #" & Natural'Image (Event_ID));
6
7 end Event_Managers;
```

Listing 132: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Event_Managers;   use Event_Managers;
5 with Ada.Real_Time;   use Ada.Real_Time;
6
7 procedure Main is
8     type Test_Case_Index is (Event_Manager_Chk);
9
10    procedure Check (TC : Test_Case_Index) is
11        Ev_Mng : array (1 .. 5) of Event_Manager;
12    begin
13        case TC is
14            when Event_Manager_Chk =>
15                for I in Ev_Mng'Range loop
16                    Ev_Mng (I).Start (I);
17                end loop;
18                Ev_Mng (1).Event (Clock + Seconds (5));
19                Ev_Mng (2).Event (Clock + Seconds (3));
20                Ev_Mng (3).Event (Clock + Seconds (1));
21                Ev_Mng (4).Event (Clock + Seconds (2));
22                Ev_Mng (5).Event (Clock + Seconds (4));
23            end case;
24        end Check;
25
26    begin
27        if Argument_Count < 1 then
28            Put_Line ("ERROR: missing arguments! Exiting...");
29            return;
30        elsif Argument_Count > 1 then
31            Put_Line ("Ignoring additional arguments...");
32        end if;
33
34        Check (Test_Case_Index'Value (Argument (1)));
35    end Main;
```


11.3 Generic Protected Queue

Goal: create a queue container using a protected type.

Steps:

1. Implement the generic package `Gen_Queues`.
 1. Declare the protected type `Queue`.
 2. Implement the `Empty` function.
 3. Implement the `Full` function.
 4. Implement the `Push` entry.
 5. Implement the `Pop` entry.

Requirements:

1. These are the formal parameters for the generic package `Gen_Queues`:
 1. a formal modular type;
 - This modular type should be used by the `Queue` to declare an array that stores the elements of the queue.
 - The modulus of the modular type must correspond to the maximum number of elements of the queue.
 2. the data type of the elements of the queue.
 - Select a formal parameter that allows you to store elements of any data type in the queue.
2. These are the operations of the `Queue` type:
 1. Function `Empty` indicates whether the queue is empty.
 2. Function `Full` indicates whether the queue is full.
 3. Entry `Push` stores an element in the queue.
 4. Entry `Pop` removes an element from the queue and returns the element via output parameter.

Remarks:

1. In this exercise, we create a queue container by declaring and implementing a protected type (`Queue`) as part of a generic package (`Gen_Queues`).
2. As a bonus exercise, you can analyze the body of the `Queue_Tests` package and understand how the `Queue` type is used there.
 1. In particular, the procedure `Concurrent_Test` implements two tasks: `T_Producer` and `T_Consumer`. They make use of the queue concurrently.

Listing 133: `gen_queues.ads`

```
1 package Gen_Queues is
2
3 end Gen_Queues;
```

Listing 134: `gen_queues.adb`

```
1 package body Gen_Queues is
2
3 end Gen_Queues;
```

Listing 135: queue_tests.ads

```

1 package Queue_Tests is
2
3     procedure Simple_Test;
4
5     procedure Concurrent_Test;
6
7 end Queue_Tests;
```

Listing 136: queue_tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_Queues;
4
5 package body Queue_Tests is
6
7     Max : constant := 10;
8     type Queue_Mod is mod Max;
9
10    procedure Simple_Test is
11        package Queues_Float is new Gen_Queues (Queue_Mod, Float);
12
13        Q_F : Queues_Float.Queue;
14        V : Float;
15    begin
16        V := 10.0;
17        while not Q_F.Full loop
18            Q_F.Push (V);
19            V := V + 1.5;
20        end loop;
21
22        while not Q_F.Empty loop
23            Q_F.Pop (V);
24            Put_Line ("Value from queue: " & Float'Image (V));
25        end loop;
26    end Simple_Test;
27
28    procedure Concurrent_Test is
29        package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
30
31        Q_I : Queues_Integer.Queue;
32
33        task T_Producer;
34        task T_Consumer;
35
36        task body T_Producer is
37            V : Integer := 100;
38        begin
39            for I in 1 .. 2 * Max loop
40                Q_I.Push (V);
41                V := V + 1;
42            end loop;
43        end T_Producer;
44
45        task body T_Consumer is
46            V : Integer;
47        begin
48            delay 1.5;
```

(continues on next page)

(continued from previous page)

```

50     while not Q_I.Empty loop
51         Q_I.Pop (V);
52         Put_Line ("Value from queue: " & Integer'Image (V));
53         delay 0.2;
54     end loop;
55 end T_Consumer;
56 begin
57     null;
58 end Concurrent_Test;
59
60 end Queue_Tests;

```

Listing 137: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Queue_Tests;      use Queue_Tests;
5
6  procedure Main is
7      type Test_Case_Index is (Simple_Queue_Chk,
8                               Concurrent_Queue_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11
12     begin
13         case TC is
14             when Simple_Queue_Chk =>
15                 Simple_Test;
16             when Concurrent_Queue_Chk =>
17                 Concurrent_Test;
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 1 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 1 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)));
30 end Main;

```

DESIGN BY CONTRACTS

12.1 Price Range

Goal: use predicates to indicate the correct range of prices.

Steps:

1. Complete the Prices package.
 1. Rewrite the type declaration of Price.

Requirements:

1. Type Price must use a predicate instead of a range.

Remarks:

1. As discussed in the course, ranges are a form of contract.
 1. For example, the subtype Price below indicates that a value of this subtype must always be positive:
2. Interestingly, you can replace ranges by predicates, which is the goal of this exercise.

```
subtype Price is Amount range 0.0 .. Amount'Last;
```

Listing 138: prices.ads

```
1 package Prices is
2
3     type Amount is delta 10.0 ** (-2) digits 12;
4
5     subtype Price is Amount range 0.0 .. Amount'Last;
6
7 end Prices;
```

Listing 139: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Prices;             use Prices;
6
7 procedure Main is
8
9     type Test_Case_Index is
10         (Price_Range_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
```

(continues on next page)

(continued from previous page)

```

13
14     procedure Check_Range (A : Amount) is
15         P : constant Price := A;
16     begin
17         Put_Line ("Price: " & Price'Image (P));
18     end Check_Range;
19
20     begin
21         case TC is
22             when Price_Range_Chk =>
23                 Check_Range (-2.0);
24         end case;
25     exception
26         when Constraint_Error =>
27             Put_Line ("Constraint_Error detected (NOT as expected).");
28         when Assert_Failure =>
29             Put_Line ("Assert_Failure detected (as expected).");
30     end Check;
31
32     begin
33         if Argument_Count < 1 then
34             Put_Line ("ERROR: missing arguments! Exiting...");
35             return;
36         elsif Argument_Count > 1 then
37             Put_Line ("Ignoring additional arguments...");
38         end if;
39
40         Check (Test_Case_Index'Value (Argument (1)));
41     end Main;

```

12.2 Pythagorean Theorem: Predicate

Goal: use the Pythagorean theorem as a predicate.

Steps:

1. Complete the Triangles package.
 1. Add a predicate to the Right_Triangle type.

Requirements:

1. The Right_Triangle type must use the Pythagorean theorem as a predicate to ensure that its components are consistent.

Remarks:

1. As you probably remember, the [Pythagoras' theorem](https://en.wikipedia.org/wiki/Pythagorean_theorem)² states that the square of the hypotenuse of a right triangle is equal to the sum of the squares of the other two sides.

Listing 140: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is record
6         H      : Length := 0;
7         -- Hypotenuse

```

(continues on next page)

² https://en.wikipedia.org/wiki/Pythagorean_theorem

(continued from previous page)

```

8      C1, C2 : Length := 0;
9      -- Catheti / legs
10     end record;
11
12     function Init (H, C1, C2 : Length) return Right_Triangle is
13         ((H, C1, C2));
14
15 end Triangles;

```

Listing 141: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 142: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         "(" & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")";
8
9 end Triangles.IO;

```

Listing 143: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");

```

(continues on next page)

(continued from previous page)

```

30     end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;

```

12.3 Pythagorean Theorem: Precondition

Goal: use the Pythagorean theorem as a precondition.

Steps:

1. Complete the Triangles package.
 1. Add a precondition to the Init function.

Requirements:

1. The Init function must use the Pythagorean theorem as a precondition to ensure that the input values are consistent.

Remarks:

1. In this exercise, you'll work again with the Right_Triangle type.
 1. This time, your job is to use a precondition instead of a predicate.
 2. The precondition is applied to the Init function, not to the Right_Triangle type.

Listing 144: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record
6          H      : Length := 0;
7          -- Hypotenuse
8          C1, C2 : Length := 0;
9          -- Catheti / legs
10     end record;
11
12     function Init (H, C1, C2 : Length) return Right_Triangle is
13         ((H, C1, C2));

```

(continues on next page)

(continued from previous page)

```

14
15 end Triangles;

```

Listing 145: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 146: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         "(" & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")");
8
9 end Triangles.IO;

```

Listing 147: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk  => Check_Triangle (12, 8, 6);

```

(continues on next page)

(continued from previous page)

```

36         when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37         when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38         when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39         when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40     end case;
41 end Check;
42
43 begin
44     if Argument_Count < 1 then
45         Put_Line ("ERROR: missing arguments! Exiting...");
46         return;
47     elsif Argument_Count > 1 then
48         Put_Line ("Ignoring additional arguments...");
49     end if;
50
51     Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

12.4 Pythagorean Theorem: Postcondition

Goal: use the Pythagorean theorem as a postcondition.

Steps:

1. Complete the Triangles package.
 1. Add a postcondition to the Init function.

Requirements:

1. The Init function must use the Pythagorean theorem as a postcondition to ensure that the returned object is consistent.

Remarks:

1. In this exercise, you'll work again with the Triangles package.
 1. This time, your job is to apply a postcondition instead of a precondition to the Init function.

Listing 148: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is record
6         H      : Length := 0;
7         -- Hypotenuse
8         C1, C2 : Length := 0;
9         -- Catheti / legs
10    end record;
11
12    function Init (H, C1, C2 : Length) return Right_Triangle is
13        ((H, C1, C2));
14
15 end Triangles;

```

Listing 149: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;
```

Listing 150: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         (" " & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")");
8
9 end Triangles.IO;
```

Listing 151: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
```

(continues on next page)

(continued from previous page)

```

41     end Check;
42
43 begin
44     if Argument_Count < 1 then
45         Put_Line ("ERROR: missing arguments! Exiting...");
46         return;
47     elsif Argument_Count > 1 then
48         Put_Line ("Ignoring additional arguments...");
49     end if;
50
51     Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

12.5 Pythagorean Theorem: Type Invariant

Goal: use the Pythagorean theorem as a type invariant.

Steps:

1. Complete the Triangles package.
 1. Add a type invariant to the Right_Triangle type.

Requirements:

1. Right_Triangle is a private type.
 1. It must use the Pythagorean theorem as a type invariant to ensure that its encapsulated components are consistent.

Remarks:

1. In this exercise, Right_Triangle is declared as a private type.
 1. In this case, we use a type invariant for Right_Triangle to check the Pythagorean theorem.
2. As a bonus, after completing the exercise, you may analyze the effect that default values have on type invariants.
 1. For example, the declaration of Right_Triangle uses zero as the default values of the three triangle lengths.
 2. If you replace those default values with Length'Last, you'll get different results.
 3. Make sure you understand why this is happening.

Listing 152: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is private;
6
7      function Init (H, C1, C2 : Length) return Right_Triangle;
8
9  private
10
11      type Right_Triangle is record
12          H      : Length := 0;
13          -- Hypotenuse
14          C1, C2 : Length := 0;

```

(continues on next page)

(continued from previous page)

```

15     -- Catheti / legs
16 end record;
17
18 function Init (H, C1, C2 : Length) return Right_Triangle is
19     ((H, C1, C2));
20
21 end Triangles;

```

Listing 153: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 154: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         "(" & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")";
8
9 end Triangles.IO;

```

Listing 155: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;

```

(continues on next page)

(continued from previous page)

```

31
32   begin
33       case TC is
34           when Triangle_8_6_Pass_Chk    => Check_Triangle (10, 8, 6);
35           when Triangle_8_6_Fail_Chk    => Check_Triangle (12, 8, 6);
36           when Triangle_10_24_Pass_Chk  => Check_Triangle (26, 10, 24);
37           when Triangle_10_24_Fail_Chk  => Check_Triangle (12, 10, 24);
38           when Triangle_18_24_Pass_Chk  => Check_Triangle (30, 18, 24);
39           when Triangle_18_24_Fail_Chk  => Check_Triangle (32, 18, 24);
40       end case;
41   end Check;
42
43   begin
44       if Argument_Count < 1 then
45           Put_Line ("ERROR: missing arguments! Exiting...");
46           return;
47       elsif Argument_Count > 1 then
48           Put_Line ("Ignoring additional arguments...");
49       end if;
50
51       Check (Test_Case_Index'Value (Argument (1)));
52   end Main;

```

12.6 Primary Color

Goal: extend a package for HTML colors so that it can handle primary colors.

Steps:

1. Complete the Color_Types package.
 1. Declare the HTML_RGB_Color subtype.
 2. Implement the To_Int_Color function.

Requirements:

1. The HTML_Color type is an enumeration that contains a list of HTML colors.
2. The To_RGB_Lookup_Table array implements a lookup-table to convert the colors into a hexadecimal value using RGB color components (i.e. Red, Green and Blue)
3. Function To_Int_Color extracts one of the RGB components of an HTML color and returns its hexadecimal value.
 1. The function has two parameters:
 - First parameter is the HTML color (HTML_Color type).
 - Second parameter indicates which RGB component is to be extracted from the HTML color (HTML_RGB_Color subtype).
 2. For example, if we call To_Int_Color (Salmon, Red), the function returns #FA,
 - This is the hexadecimal value of the red component of the Salmon color.
 - You can find further remarks below about this color as an example.
4. The HTML_RGB_Color subtype is limited to the primary RGB colors components (i.e. Red, Green and Blue).
 1. This subtype is used to select the RGB component in calls to To_Int_Color.
 2. You must use a predicate in the type declaration.

Remarks:

1. In this exercise, we reuse the code of the Colors: Lookup-Table exercise from the [Arrays](#) (page 41) labs.
2. These are the hexadecimal values of the colors that we used in the original exercise:

Color	Value
Salmon	#FA8072
Firebrick	#B22222
Red	#FF0000
Darkred	#8B0000
Lime	#00FF00
Forestgreen	#228B22
Green	#008000
Darkgreen	#006400
Blue	#0000FF
Mediumblue	#0000CD
Darkblue	#00008B

3. You can extract the hexadecimal value of each primary color by splitting the values from the table above into three hexadecimal values with two digits each.
 - For example, the hexadecimal value of Salmon is #FA8072, where:
 - the first part of this hexadecimal value (#FA) corresponds to the red component,
 - the second part (#80) corresponds to the green component, and
 - the last part (#72) corresponds to the blue component.

Listing 156: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   subtype Int_Color is Integer range 0 .. 255;
17
18   function Image (I : Int_Color) return String;
19
20   type RGB is record
21     Red   : Int_Color;
22     Green : Int_Color;
23     Blue  : Int_Color;
24   end record;
25
26   function To_RGB (C : HTML_Color) return RGB;
27
28   function Image (C : RGB) return String;
29

```

(continues on next page)

(continued from previous page)

```

30  type HTML_Color_RGB_Array is array (HTML_Color) of RGB;
31
32  To_RGB_Lookup_Table : constant HTML_Color_RGB_Array
33  := (Salmon      => (16#FA#, 16#80#, 16#72#),
34       Firebrick  => (16#B2#, 16#22#, 16#22#),
35       Red        => (16#FF#, 16#00#, 16#00#),
36       Darkred    => (16#8B#, 16#00#, 16#00#),
37       Lime       => (16#00#, 16#FF#, 16#00#),
38       Forestgreen => (16#22#, 16#8B#, 16#22#),
39       Green      => (16#00#, 16#80#, 16#00#),
40       Darkgreen  => (16#00#, 16#64#, 16#00#),
41       Blue       => (16#00#, 16#00#, 16#FF#),
42       Mediumblue => (16#00#, 16#00#, 16#CD#),
43       Darkblue   => (16#00#, 16#00#, 16#8B#));
44
45  subtype HTML_RGB_Color is HTML_Color;
46
47  function To_Int_Color (C : HTML_Color;
48                        S : HTML_RGB_Color) return Int_Color;
49  -- Convert to hexadecimal value for the selected RGB component S
50
51  end Color_Types;

```

Listing 157: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_RGB (C : HTML_Color) return RGB is
6      begin
7          return To_RGB_Lookup_Table (C);
8      end To_RGB;
9
10     function To_Int_Color (C : HTML_Color;
11                           S : HTML_RGB_Color) return Int_Color is
12     begin
13         -- Implement function!
14         return 0;
15     end To_Int_Color;
16
17     function Image (I : Int_Color) return String is
18     subtype Str_Range is Integer range 1 .. 10;
19     S : String (Str_Range);
20     begin
21         Ada.Integer_Text_IO.Put (To    => S,
22                                Item  => I,
23                                Base  => 16);
24         return S;
25     end Image;
26
27     function Image (C : RGB) return String is
28     begin
29         return "(" & Image (C.Red)
30            & ", " & Image (C.Green)
31            & ", " & Image (C.Blue)
32            & ")";
33     end Image;
34
35  end Color_Types;

```

Listing 158: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_Red_Chk,
9           HTML_Color_Green_Chk,
10          HTML_Color_Blue_Chk);
11
12      procedure Check (TC : Test_Case_Index) is
13
14          procedure Check_HTML_Colors (S : HTML_RGB_Color) is
15              begin
16                  Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
17                  for I in HTML_Color'Range loop
18                      Put_Line (HTML_Color'Image (I) & " => "
19                              & Image (To_Int_Color (I, S)) & ".");
20                  end loop;
21              end Check_HTML_Colors;
22
23          begin
24              case TC is
25                  when HTML_Color_Red_Chk =>
26                      Check_HTML_Colors (Red);
27                  when HTML_Color_Green_Chk =>
28                      Check_HTML_Colors (Green);
29                  when HTML_Color_Blue_Chk =>
30                      Check_HTML_Colors (Blue);
31              end case;
32          end Check;
33
34      begin
35          if Argument_Count < 1 then
36              Put_Line ("ERROR: missing arguments! Exiting...");
37              return;
38          elsif Argument_Count > 1 then
39              Put_Line ("Ignoring additional arguments...");
40          end if;
41
42          Check (Test_Case_Index'Value (Argument (1)));
43      end Main;

```


OBJECT-ORIENTED PROGRAMMING

13.1 Simple type extension

Goal: work with type extensions using record types containing numeric components.

Steps:

1. Implement the `Type_Extensions` package.
 1. Declare the record type `T_Float`.
 2. Declare the record type `T_Mixed`
 3. Implement the `Init` function for the `T_Float` type with a floating-point input parameter.
 4. Implement the `Init` function for the `T_Float` type with an integer input parameter.
 5. Implement the `Image` function for the `T_Float` type.
 6. Implement the `Init` function for the `T_Mixed` type with a floating-point input parameter.
 7. Implement the `Init` function for the `T_Mixed` type with an integer input parameter.
 8. Implement the `Image` function for the `T_Mixed` type.

Requirements:

1. Record type `T_Float` contains the following component:
 1. `F`, a floating-point type.
2. Record type `T_Mixed` is derived from the `T_Float` type.
 1. `T_Mixed` extends `T_Float` with the following component:
 1. `I`, an integer component.
 2. Both components must be numerically *synchronized*:
 - For example, if the floating-point component contains the value 2.0, the value of the integer component must be 2.
 - In order to simplify the implementation, you can simply use **Integer** (`F`) to convert a floating-point variable `F` to integer.
3. Function `Init` returns an object of the corresponding type (`T_Float` or `T_Mixed`).
 1. For each type, two versions of `Init` must be declared:
 1. one with a floating-point input parameter,
 2. another with an integer input parameter.
 2. The parameter to `Init` is used to initialize the record components.
4. Function `Image` returns a string for the components of the record type.

1. In case of the Image function for the T_Float type, the string must have the format "{ F => <float value> }".
 - For example, the call Image (T_Float'(Init (8.0))) should return the string "{ F => 8.00000E+00 }".
2. In case of the Image function for the T_Mixed type, the string must have the format "{ F => <float value>, I => <integer value> }".
 - For example, the call Image (T_Mixed'(Init (8.0))) should return the string "{ F => 8.00000E+00, I => 8 }".

Listing 159: type_extensions.ads

```
1 package Type_Extensions is
2
3   -- Create declaration of T_Float type!
4   type T_Float is null record;
5
6   -- function Init ...
7
8   -- function Image ...
9
10  -- Create declaration of T_Mixed type!
11  type T_Mixed is null record;
12
13 end Type_Extensions;
```

Listing 160: type_extensions.adb

```
1 package body Type_Extensions is
2
3 end Type_Extensions;
```

Listing 161: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Type_Extensions; use Type_Extensions;
5
6 procedure Main is
7
8   type Test_Case_Index is
9     (Type_Extension_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12    F1, F2 : T_Float;
13    M1, M2 : T_Mixed;
14  begin
15    case TC is
16    when Type_Extension_Chk =>
17      F1 := Init (2.0);
18      F2 := Init (3);
19      M1 := Init (4.0);
20      M2 := Init (5);
21
22      if M2 in T_Float'Class then
23        Put_Line ("T_Mixed is in T_Float'Class as expected");
24      end if;
25
26      Put_Line ("F1: " & Image (F1));
```

(continues on next page)

(continued from previous page)

```

27         Put_Line ("F2: " & Image (F2));
28         Put_Line ("M1: " & Image (M1));
29         Put_Line ("M2: " & Image (M2));
30     end case;
31 end Check;
32
33 begin
34     if Argument_Count < 1 then
35         Put_Line ("ERROR: missing arguments! Exiting...");
36         return;
37     elsif Argument_Count > 1 then
38         Put_Line ("Ignoring additional arguments...");
39     end if;
40
41     Check (Test_Case_Index'Value (Argument (1)));
42 end Main;

```

13.2 Online Store

Goal: create an online store for the members of an association.

Steps:

1. Implement the `Online_Store` package.
 1. Declare the `Member` type.
 2. Declare the `Full_Member` type.
 3. Implement the `Get_Status` function for the `Member` type.
 4. Implement the `Get_Price` function for the `Member` type.
 5. Implement the `Get_Status` function for the `Full_Member` type.
 6. Implement the `Get_Price` function for the `Full_Member` type.
2. Implement the `Online_Store.Tests` child package.
 1. Implement the `Simple_Test` procedure.

Requirements:

1. Package `Online_Store` implements an online store application for the members of an association.
 1. In this association, members can have one of the following status:
 - associate member, or
 - full member.
2. Function `Get_Price` returns the correct price of an item.
 1. Associate members must pay the full price when they buy items from the online store.
 2. Full members can get a discount.
 1. The discount rate can be different for each full member — depending on factors that are irrelevant for this exercise.
3. Package `Online_Store` has following types:
 1. Percentage type, which represents a percentage ranging from 0.0 to 1.0.
 2. Member type for associate members containing following components:

- Start, which indicates the starting year of the membership.
 - This information is common for both associate and full members.
 - You can use the Year_Number type from the standard Ada.Calendar package for this component.
- 3. Full_Member type for full members.
 1. This type must extend the Member type above.
 2. It contains the following additional component:
 - Discount, which indicates the discount rate that the full member gets in the online store.
 - This component must be of Percentage type.
- 4. For the Member and Full_Member types, you must implement the following functions:
 1. Get_Status, which returns a string with the membership status.
 - The string must be "Associate Member" or "Full Member", respectively.
 2. Get_Price, which returns the *adapted price* of an item — indicating the actual due amount.
 - For example, for a full member with a 10% discount rate, the actual due amount of an item with a price of 100.00 is 90.00.
 - Associated members don't get a discount, so they always pay the full price.
- 5. Procedure Simple_Test (from the Online_Store.Tests package) is used for testing.
 1. Based on a list of members that bought on the online store and the corresponding full price of the item, Simple_Test must display information about each member and the actual due amount after discounts.
 2. Information about the members must be displayed in the following format:

```
Member # <number>
Status: <status>
Since: <year>
Due Amount: <value>
-----
```

3. For this exercise, Simple_Test must use the following list:

#	Membership status	Start (year)	Discount	Full Price
1	Associate	2010	N/A	250.00
2	Full	1998	10.0 %	160.00
3	Full	1987	20.0 %	400.00
4	Associate	2013	N/A	110.00

4. In order to pass the tests, the information displayed by a call to Simple_Test must conform to the format described above.
 - You can find another example in the remarks below.

Remarks:

1. In previous labs, we could have implemented a simplified version of the system described above by simply using an enumeration type to specify the membership status. For example:

```
type Member_Status is (Associate_Member, Full_Member);
```

1. In this case, the `Get_Price` function would then evaluate the membership status and adapt the item price — assuming a fixed discount rate for all full members. This could be the corresponding function declaration:

```
type Amount is delta 10.0**(-2) digits 10;

function Get_Price (M : Member_Status;
                   P : Amount) return Amount;
```

2. In this exercise, however, we'll use type extension to represent the membership status in our application.
2. For the procedure `Simple_Test`, let's consider the following list of members as an example:

#	Membership status	Start (year)	Discount	Full Price
1	Associate	2002	N/A	100.00
2	Full	2005	10.0 %	100.00

- For this list, the test procedure displays the following information (in this exact format):

```
Member # 1
Status: Associate Member
Since: 2002
Due Amount: 100.00
-----
Member # 2
Status: Full Member
Since: 2005
Due Amount: 90.00
-----
```

- Here, although both members had the same full price (as indicated by the last column), member #2 gets a reduced due amount of 90.00 because of the full membership status.

Listing 162: online_store.ads

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 package Online_Store is
4
5     type Amount is delta 10.0**(-2) digits 10;
6
7     subtype Percentage is Amount range 0.0 .. 1.0;
8
9     -- Create declaration of Member type!
10    --
11    -- You can use Year_Number from Ada.Calendar for the membership
12    -- starting year.
13    --
14    type Member is null record;
15
16    function Get_Status (M : Member) return String;
17
18    function Get_Price (M : Member;
19                      P : Amount) return Amount;
20
21    -- Create declaration of Full_Member type!
22    --
```

(continues on next page)

(continued from previous page)

```
23  -- Use the Percentage type for storing the membership discount.
24  --
25  type Full_Member is null record;
26
27  function Get_Status (M : Full_Member) return String;
28
29  function Get_Price (M : Full_Member;
30                    P : Amount) return Amount;
31
32  end Online_Store;
```

Listing 163: online_store.adb

```
1  package body Online_Store is
2
3      function Get_Status (M : Member) return String is
4          ("");
5
6      function Get_Status (M : Full_Member) return String is
7          ("");
8
9      function Get_Price (M : Member;
10                       P : Amount) return Amount is (0.0);
11
12     function Get_Price (M : Full_Member;
13                       P : Amount) return Amount is
14         (0.0);
15
16  end Online_Store;
```

Listing 164: online_store-tests.ads

```
1  package Online_Store.Tests is
2
3      procedure Simple_Test;
4
5  end Online_Store.Tests;
```

Listing 165: online_store-tests.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Online_Store.Tests is
4
5      procedure Simple_Test is
6      begin
7          null;
8      end Simple_Test;
9
10  end Online_Store.Tests;
```

Listing 166: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  with Online_Store; use Online_Store;
5  with Online_Store.Tests; use Online_Store.Tests;
6
7  procedure Main is
```

(continues on next page)

(continued from previous page)

```

8
9  type Test_Case_Index is
10     (Type_Chk,
11      Unit_Test_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14
15      function Result_Image (Result : Boolean) return String is
16          (if Result then "OK" else "not OK");
17
18  begin
19      case TC is
20      when Type_Chk =>
21          declare
22              AM : constant Member      := (Start      => 2002);
23              FM : constant Full_Member := (Start      => 1990,
24                                             Discount => 0.2);
25
26              begin
27                  Put_Line ("Testing Status of Associate Member Type => "
28                           & Result_Image (AM.Get_Status = "Associate Member"));
29                  Put_Line ("Testing Status of Full Member Type => "
30                           & Result_Image (FM.Get_Status = "Full Member"));
31                  Put_Line ("Testing Discount of Associate Member Type => "
32                           & Result_Image (AM.Get_Price (100.0) = 100.0));
33                  Put_Line ("Testing Discount of Full Member Type => "
34                           & Result_Image (FM.Get_Price (100.0) = 80.0));
35
36              end;
37              when Unit_Test_Chk =>
38                  Simple_Test;
39
40          end case;
41      end Check;
42
43  begin
44      if Argument_Count < 1 then
45          Put_Line ("ERROR: missing arguments! Exiting...");
46          return;
47      elsif Argument_Count > 1 then
48          Put_Line ("Ignoring additional arguments...");
49      end if;
50
51      Check (Test_Case_Index'Value (Argument (1)));
52  end Main;

```


STANDARD LIBRARY: CONTAINERS

14.1 Simple todo list

Goal: implement a simple to-do list system using vectors.

Steps:

1. Implement the `Todo_Lists` package.
 1. Declare the `Todo_Item` type.
 2. Declare the `Todo_List` type.
 3. Implement the `Add` procedure.
 4. Implement the `Display` procedure.
2. `Todo_Item` type is used to store to-do items.
 1. It should be implemented as an access type to strings.
3. `Todo_List` type is the container for all to-do items.
 1. It should be implemented as a **vector**.
4. Procedure `Add` adds items (of `Todo_Item` type) to the list (of `Todo_List` type).
 1. This requires allocating a string for the access type.
5. Procedure `Display` is used to display all to-do items.
 1. It must display one item per line.

Remarks:

1. This exercise is based on the *Simple todo list* exercise from the [More About Types](#) (page 59).
 1. Your goal is to rewrite that exercise using vectors instead of arrays.
 2. You may reuse the code you've already implemented as a starting point.

Listing 167: `todo_lists.ads`

```
1 package Todo_Lists is
2
3     type Todo_Item is access String;
4
5     type Todo_List is null record;
6
7     procedure Add (Todos : in out Todo_List;
8                   Item   : String);
9
10    procedure Display (Todos : Todo_List);
```

(continues on next page)

(continued from previous page)

```

11
12 end Todo_Lists;

```

Listing 168: todo_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Todo_Lists is
4
5     procedure Add (Todos : in out Todo_List;
6                   Item : String) is
7     begin
8         null;
9     end Add;
10
11    procedure Display (Todos : Todo_List) is
12    begin
13        Put_Line ("TO-DO LIST");
14    end Display;
15
16 end Todo_Lists;

```

Listing 169: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Todo_Lists;       use Todo_Lists;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Todo_List_Chk);
9
10    procedure Check (TC : Test_Case_Index) is
11        T : Todo_List;
12    begin
13        case TC is
14            when Todo_List_Chk =>
15                Add (T, "Buy milk");
16                Add (T, "Buy tea");
17                Add (T, "Buy present");
18                Add (T, "Buy tickets");
19                Add (T, "Pay electricity bill");
20                Add (T, "Schedule dentist appointment");
21                Add (T, "Call sister");
22                Add (T, "Revise spreadsheet");
23                Add (T, "Edit entry page");
24                Add (T, "Select new design");
25                Add (T, "Create upgrade plan");
26                Display (T);
27        end case;
28    end Check;
29
30 begin
31     if Argument_Count < 1 then
32         Put_Line ("ERROR: missing arguments! Exiting...");
33         return;
34     elsif Argument_Count > 1 then
35         Put_Line ("Ignoring additional arguments...");
36     end if;

```

(continues on next page)

(continued from previous page)

```

37
38     Check (Test_Case_Index'Value (Argument (1)));
39 end Main;
```

14.2 List of unique integers

Goal: create function that removes duplicates from and orders a collection of elements.

Steps:

1. Implement package Ops.
 1. Declare the Int_Array type.
 2. Declare the Integer_Sets type.
 3. Implement the Get_Unique function that returns a set.
 4. Implement the Get_Unique function that returns an array of integer values.

Requirements:

1. The Int_Array type is an unconstrained array of positive range.
2. The Integer_Sets package is an instantiation of the Ordered_Sets package for the **Integer** type.
3. The Get_Unique function must remove duplicates from an input array of integer values and order the elements.
 1. For example:
 - if the input array contains (7, 7, 1)
 - the function must return (1, 7).
 2. You must implement this function by using sets from the Ordered_Sets package.
 3. Get_Unique must be implemented in two versions:
 - one version that returns a set — Set type from the Ordered_Sets package.
 - one version that returns an array of integer values — Int_Array type.

Remarks:

1. Sets — as the one found in the generic Ordered_Sets package — are useful for quickly and easily creating an algorithm that removes duplicates from a list of elements.

Listing 170: ops.ads

```

1 with Ada.Containers.Ordered_Sets;
2
3 package Ops is
4
5     -- type Int_Array is ...
6
7     -- package Integer_Sets is ...
8
9     subtype Int_Set is Integer_Sets.Set;
10
11     function Get_Unique (A : Int_Array) return Int_Set;
12
13     function Get_Unique (A : Int_Array) return Int_Array;
14
15 end Ops;
```

Listing 171: ops.adb

```
1 package body Ops is
2
3     function Get_Unique (A : Int_Array) return Int_Set is
4     begin
5         null;
6     end Get_Unique;
7
8     function Get_Unique (A : Int_Array) return Int_Array is
9     begin
10        null;
11    end Get_Unique;
12
13 end Ops;
```

Listing 172: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Ops;                    use Ops;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Get_Unique_Set_Chk,
9          Get_Unique_Array_Chk);
10
11     procedure Check (TC : Test_Case_Index;
12                     A : Int_Array) is
13
14         procedure Display_Unique_Set (A : Int_Array) is
15             S : constant Int_Set := Get_Unique (A);
16         begin
17             for E of S loop
18                 Put_Line (Integer'Image (E));
19             end loop;
20         end Display_Unique_Set;
21
22         procedure Display_Unique_Array (A : Int_Array) is
23             AU : constant Int_Array := Get_Unique (A);
24         begin
25             for E of AU loop
26                 Put_Line (Integer'Image (E));
27             end loop;
28         end Display_Unique_Array;
29
30     begin
31         case TC is
32             when Get_Unique_Set_Chk => Display_Unique_Set (A);
33             when Get_Unique_Array_Chk => Display_Unique_Array (A);
34         end case;
35     end Check;
36
37 begin
38     if Argument_Count < 3 then
39         Put_Line ("ERROR: missing arguments! Exiting...");
40         return;
41     else
42         declare
43             A : Int_Array (1 .. Argument_Count - 1);
```

(continues on next page)

(continued from previous page)

```
44     begin
45         for I in A'Range loop
46             A (I) := Integer'Value (Argument (1 + I));
47         end loop;
48         Check (Test_Case_Index'Value (Argument (1)), A);
49     end;
50 end if;
51 end Main;
```


STANDARD LIBRARY: DATES & TIMES

15.1 Holocene calendar

Goal: create a function that returns the year in the Holocene calendar.

Steps:

1. Implement the `To_Holocene_Year` function.

Requirements:

1. The `To_Holocene_Year` extracts the year from a time object (Time type) and returns the corresponding year for the [Holocene calendar](https://en.wikipedia.org/wiki/Holocene_calendar)³.
 1. For positive (AD) years, the Holocene year is calculated by adding 10,000 to the year number.

Remarks:

1. In this exercise, we don't deal with BC years.
2. Note that the year component of the Time type from the `Ada.Calendar` package is limited to years starting with 1901.

Listing 173: `to_holocene_year.adb`

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 function To_Holocene_Year (T : Time) return Integer is
4 begin
5     return 0;
6 end To_Holocene_Year;
```

Listing 174: `main.adb`

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Calendar;          use Ada.Calendar;
4
5 with To_Holocene_Year;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Holocene_Chk);
10
11     procedure Display_Holocene_Year (Y : Year_Number) is
12         HY : Integer;
13     begin
14         HY := To_Holocene_Year (Time_Of (Y, 1, 1));
```

(continues on next page)

³ https://en.wikipedia.org/wiki/Holocene_calendar

(continued from previous page)

```
15     Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
16     Put_Line ("Year (Holocene): " & Integer'Image (HY));
17 end Display_Holocene_Year;
18
19 procedure Check (TC : Test_Case_Index) is
20 begin
21     case TC is
22     when Holocene_Chk =>
23         Display_Holocene_Year (2012);
24         Display_Holocene_Year (2020);
25     end case;
26 end Check;
27
28 begin
29     if Argument_Count < 1 then
30         Put_Line ("ERROR: missing arguments! Exiting...");
31         return;
32     elsif Argument_Count > 1 then
33         Put_Line ("Ignoring additional arguments...");
34     end if;
35
36     Check (Test_Case_Index'Value (Argument (1)));
37 end Main;
```

15.2 List of events

Goal: create a system to manage a list of events.

Steps:

1. Implement the Events package.
 1. Declare the Event_Item type.
 2. Declare the Event_Items type.
2. Implement the Events.Lists package.
 1. Declare the Event_List type.
 2. Implement the Add procedure.
 3. Implement the Display procedure.

Requirements:

1. The Event_Item type (from the Events package) contains the *description of an event*.
 1. This description shall be stored in an access-to-string type.
2. The Event_Items type stores a list of events.
 1. This will be used later to represent multiple events for a specific date.
 2. You shall use a vector for this type.
3. The Events.Lists package contains the subprograms that are used in the test application.
4. The Event_List type (from the Events.Lists package) maps a list of events to a specific date.
 1. You must use the Event_Items type for the list of events.
 2. You shall use the Time type from the Ada.Calendar package for the dates.

3. Since we expect the events to be ordered by the date, you shall use ordered maps for the `Event_List` type.
5. Procedure `Add` adds an event into the list of events for a specific date.
6. Procedure `Display` must display all events for each date (ordered by date) using the following format:

```
<event_date #1>
  <description of item #1a>
  <description of item #1b>
<event_date #2>
  <description of item #2a>
  <description of item #2b>
```

1. You should use the auxiliary `Date_Image` function — available in the body of the `Events.Lists` package — to display the date in the `YYYY-MM-DD` format.

Remarks:

1. Let's briefly illustrate the expected output of this system.
 1. Consider the following example:

```
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

with Events.Lists;          use Events.Lists;

procedure Test is
  EL : Event_List;
begin
  EL.Add (Time_Of (2019, 4, 16),
    "Item #2");
  EL.Add (Time_Of (2019, 4, 15),
    "Item #1");
  EL.Add (Time_Of (2019, 4, 16),
    "Item #3");
  EL.Display;
end Test;
```

2. The expected output of the `Test` procedure must be:

```
EVENTS LIST
- 2019-04-15
  - Item #1
- 2019-04-16
  - Item #2
  - Item #3
```

Listing 175: `events.ads`

```
1 package Events is
2
3   type Event_Item is null record;
4
5   type Event_Items is null record;
6
7 end Events;
```

Listing 176: events-lists.ads

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 package Events.Lists is
4
5     type Event_List is tagged private;
6
7     procedure Add (Events      : in out Event_List;
8                   Event_Time  : Time;
9                   Event       : String);
10
11     procedure Display (Events : Event_List);
12
13 private
14
15     type Event_List is tagged null record;
16
17 end Events.Lists;
```

Listing 177: events-lists.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4 package body Events.Lists is
5
6     procedure Add (Events      : in out Event_List;
7                   Event_Time  : Time;
8                   Event       : String) is
9
10     begin
11         null;
12     end Add;
13
14     function Date_Image (T : Time) return String is
15         Date_Img : constant String := Image (T);
16     begin
17         return Date_Img (1 .. 10);
18     end;
19
20     procedure Display (Events : Event_List) is
21         T : Time;
22     begin
23         Put_Line ("EVENTS LIST");
24         -- You should use Date_Image (T) here!
25     end Display;
26
27 end Events.Lists;
```

Listing 178: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Calendar;          use Ada.Calendar;
4 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5
6 with Events.Lists;          use Events.Lists;
7
8 procedure Main is
9     type Test_Case_Index is
10         (Event_List_Chk);
```

(continues on next page)

(continued from previous page)

```
11
12 procedure Check (TC : Test_Case_Index) is
13   EL : Event_List;
14 begin
15   case TC is
16     when Event_List_Chk =>
17       EL.Add (Time_Of (2018, 2, 16),
18             "Final check");
19       EL.Add (Time_Of (2018, 2, 16),
20             "Release");
21       EL.Add (Time_Of (2018, 12, 3),
22             "Brother's birthday");
23       EL.Add (Time_Of (2018, 1, 1),
24             "New Year's Day");
25       EL.Display;
26   end case;
27 end Check;
28
29 begin
30   if Argument_Count < 1 then
31     Put_Line ("ERROR: missing arguments! Exiting...");
32     return;
33   elsif Argument_Count > 1 then
34     Put_Line ("Ignoring additional arguments...");
35   end if;
36
37   Check (Test_Case_Index'Value (Argument (1)));
38 end Main;
```


STANDARD LIBRARY: STRINGS

16.1 Concatenation

Goal: implement functions to concatenate an array of unbounded strings.

Steps:

1. Implement the `Str_Concat` package.
 1. Implement the `Concat` function for `Unbounded_String`.
 2. Implement the `Concat` function for **`String`**.

Requirements:

1. The first `Concat` function receives an unconstrained array of unbounded strings and returns the concatenation of those strings as an unbounded string.
 1. The second `Concat` function has the same parameters, but returns a standard string (**`String`** type).
2. Both `Concat` functions have the following parameters:
 1. An unconstrained array of `Unbounded_String` strings (`Unbounded_Strings` type).
 2. `Trim_Str`, a Boolean parameter indicating whether each unbounded string must be trimmed.
 3. `Add_Whitespace`, a Boolean parameter indicating whether a whitespace shall be added between each unbounded string and the next one.
 1. No whitespace shall be added after the last string of the array.

Remarks:

1. You can use the `Trim` function from the `Ada.Strings.Unbounded` package.

Listing 179: `str_concat.ads`

```
1 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2
3 package Str_Concat is
4
5     type Unbounded_Strings is array (Positive range <>) of Unbounded_String;
6
7     function Concat (USA           : Unbounded_Strings;
8                     Trim_Str      : Boolean;
9                     Add_Whitespace : Boolean) return Unbounded_String;
10
11     function Concat (USA           : Unbounded_Strings;
12                     Trim_Str      : Boolean;
13                     Add_Whitespace : Boolean) return String;
14
15 end Str_Concat;
```

Listing 180: str_concat.adb

```
1 with Ada.Strings; use Ada.Strings;
2
3 package body Str_Concat is
4
5     function Concat (USA           : Unbounded_Strings;
6                     Trim_Str      : Boolean;
7                     Add_Whitespace : Boolean) return Unbounded_String is
8
9     begin
10         return "";
11     end Concat;
12
13     function Concat (USA           : Unbounded_Strings;
14                     Trim_Str      : Boolean;
15                     Add_Whitespace : Boolean) return String is
16
17     begin
18         return "";
19     end Concat;
20
21 end Str_Concat;
```

Listing 181: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
4
5 with Str_Concat; use Str_Concat;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Unbounded_Concat_No_Trim_No_WS_Chk,
10          Unbounded_Concat_Trim_No_WS_Chk,
11          String_Concat_Trim_WS_Chk,
12          Concat_Single_Element);
13
14     procedure Check (TC : Test_Case_Index) is
15     begin
16         case TC is
17             when Unbounded_Concat_No_Trim_No_WS_Chk =>
18                 declare
19                     S : constant Unbounded_Strings := (
20                         To_Unbounded_String ("Hello"),
21                         To_Unbounded_String (" World"),
22                         To_Unbounded_String ("!"));
23                 begin
24                     Put_Line (To_String (Concat (S, False, False)));
25                 end;
26             when Unbounded_Concat_Trim_No_WS_Chk =>
27                 declare
28                     S : constant Unbounded_Strings := (
29                         To_Unbounded_String (" This "),
30                         To_Unbounded_String (" _is_ "),
31                         To_Unbounded_String (" a "),
32                         To_Unbounded_String (" _check "));
33                 begin
34                     Put_Line (To_String (Concat (S, True, False)));
35                 end;
36             when String_Concat_Trim_WS_Chk =>
37                 declare
```

(continues on next page)

(continued from previous page)

```

38         S : constant Unbounded_Strings := (
39             To_Unbounded_String (" This "),
40             To_Unbounded_String (" is a "),
41             To_Unbounded_String (" test. "));
42     begin
43         Put_Line (Concat (S, True, True));
44     end;
45 when Concat_Single_Element =>
46     declare
47         S : constant Unbounded_Strings := (
48             1 => To_Unbounded_String (" Hi "));
49     begin
50         Put_Line (Concat (S, True, True));
51     end;
52 end case;
53 end Check;
54
55 begin
56     if Argument_Count < 1 then
57         Put_Line ("ERROR: missing arguments! Exiting...");
58         return;
59     elsif Argument_Count > 1 then
60         Put_Line ("Ignoring additional arguments...");
61     end if;
62
63     Check (Test_Case_Index'Value (Argument (1)));
64 end Main;

```

16.2 List of events

Goal: create a system to manage a list of events.

Steps:

1. Implement the Events package.
 1. Declare the Event_Item subtype.
2. Implement the Events.Lists package.
 1. Adapt the Add procedure.
 2. Adapt the Display procedure.

Requirements:

1. The Event_Item type (from the Events package) contains the *description of an event*.
 1. This description is declared as a subtype of unbounded string.
2. Procedure Add adds an event into the list of events for a specific date.
 1. The declaration of E needs to be adapted to use unbounded strings.
3. Procedure Display must display all events for each date (ordered by date) using the following format:
 1. The arguments to Put_Line need to be adapted to use unbounded strings.

Remarks:

1. We use the lab on the list of events from the previous chapter (*Standard library: Dates & Times* (page 129)) as a starting point.

Listing 182: events.ads

```
1 with Ada.Containers.Vectors;
2
3 package Events is
4
5     -- subtype Event_Item is
6
7     package Event_Item_Containers is new
8         Ada.Containers.Vectors
9         (Index_Type => Positive,
10          Element_Type => Event_Item);
11
12     subtype Event_Items is Event_Item_Containers.Vector;
13
14 end Events;
```

Listing 183: events-lists.ads

```
1 with Ada.Calendar;           use Ada.Calendar;
2 with Ada.Containers.Ordered_Maps;
3
4 package Events.Lists is
5
6     type Event_List is tagged private;
7
8     procedure Add (Events      : in out Event_List;
9                   Event_Time  : Time;
10                   Event       : String);
11
12     procedure Display (Events : Event_List);
13
14 private
15
16     package Event_Time_Item_Containers is new
17         Ada.Containers.Ordered_Maps
18         (Key_Type => Time,
19          Element_Type => Event_Items,
20          "=" => Event_Item_Containers "=");
21
22     type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;
```

Listing 184: events-lists.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4 package body Events.Lists is
5
6     procedure Add (Events      : in out Event_List;
7                   Event_Time  : Time;
8                   Event       : String) is
9         use Event_Item_Containers;
10        E : constant Event_Item := new String'(Event);
11    begin
12        if not Events.Contains (Event_Time) then
13            Events.Include (Event_Time, Empty_Vector);
14        end if;
15        Events (Event_Time).Append (E);
```

(continues on next page)

(continued from previous page)

```

16  end Add;
17
18  function Date_Image (T : Time) return String is
19      Date_Img : constant String := Image (T);
20  begin
21      return Date_Img (1 .. 10);
22  end;
23
24  procedure Display (Events : Event_List) is
25      use Event_Time_Item_Containers;
26      T : Time;
27  begin
28      Put_Line ("EVENTS LIST");
29      for C in Events.Iterate loop
30          T := Key (C);
31          Put_Line ("- " & Date_Image (T));
32          for I of Events (C) loop
33              Put_Line ("    - " & I.all);
34          end loop;
35      end loop;
36  end Display;
37
38  end Events.Lists;

```

Listing 185: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;          use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
6
7  with Events;
8  with Events.Lists;          use Events.Lists;
9
10 procedure Main is
11     type Test_Case_Index is
12         (Unbounded_String_Chk,
13          Event_List_Chk);
14
15     procedure Check (TC : Test_Case_Index) is
16         EL : Event_List;
17     begin
18         case TC is
19             when Unbounded_String_Chk =>
20                 declare
21                     S : constant Events.Event_Item := To_Unbounded_String ("Checked");
22                 begin
23                     Put_Line (To_String (S));
24                 end;
25             when Event_List_Chk =>
26                 EL.Add (Time_Of (2018, 2, 16),
27                     "Final check");
28                 EL.Add (Time_Of (2018, 2, 16),
29                     "Release");
30                 EL.Add (Time_Of (2018, 12, 3),
31                     "Brother's birthday");
32                 EL.Add (Time_Of (2018, 1, 1),
33                     "New Year's Day");
34                 EL.Display;
35             end case;

```

(continues on next page)

(continued from previous page)

```
36     end Check;
37
38 begin
39     if Argument_Count < 1 then
40         Put_Line ("ERROR: missing arguments! Exiting...");
41         return;
42     elsif Argument_Count > 1 then
43         Put_Line ("Ignoring additional arguments...");
44     end if;
45
46     Check (Test_Case_Index'Value (Argument (1)));
47 end Main;
```

STANDARD LIBRARY: NUMERICS

17.1 Decibel Factor

Goal: implement functions to convert from Decibel values to factors and vice-versa.

Steps:

1. Implement the Decibels package.
 1. Implement the To_Decibel function.
 2. Implement the To_Factor function.

Requirements:

1. The subtypes Decibel and Factor are based on a floating-point type.
2. Function To_Decibel converts a multiplication factor (or ratio) to decibels.
 - For the implementation, use $20 * \log_{10}(F)$, where F is the factor/ratio.
3. Function To_Factor converts a value in decibels to a multiplication factor (or ratio).
 - For the implementation, use $10^{D/20}$, where D is the value in Decibel.

Remarks:

1. The [Decibel](https://en.wikipedia.org/wiki/Decibel)⁴ is used to express the ratio of two values on a logarithmic scale.
 1. For example, an increase of 6 dB corresponds roughly to a multiplication by two (or an increase by 100 % of the original value).
2. You can find the functions that you'll need for the calculation in the Ada.Numerics.Elementary_Functions package.

Listing 186: decibels.ads

```
1 package Decibels is
2
3     subtype Decibel is Float;
4     subtype Factor   is Float;
5
6     function To_Decibel (F : Factor) return Decibel;
7
8     function To_Factor (D : Decibel) return Factor;
9
10 end Decibels;
```

⁴ <https://en.wikipedia.org/wiki/Decibel>

Listing 187: decibels.adb

```
1 package body Decibels is
2
3     function To_Decibel (F : Factor) return Decibel is
4     begin
5         return 0.0;
6     end To_Decibel;
7
8     function To_Factor (D : Decibel) return Factor is
9     begin
10        return 0.0;
11    end To_Factor;
12
13 end Decibels;
```

Listing 188: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Decibels;          use Decibels;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Db_Chk,
9          Factor_Chk);
10
11     procedure Check (TC : Test_Case_Index; V : Float) is
12
13         package F_IO is new Ada.Text_IO.Float_IO (Factor);
14         package D_IO is new Ada.Text_IO.Float_IO (Decibel);
15
16         procedure Put_Decibel_Cnvt (D : Decibel) is
17             F : constant Factor := To_Factor (D);
18         begin
19             D_IO.Put (D, 0, 2, 0);
20             Put (" dB => Factor of ");
21             F_IO.Put (F, 0, 2, 0);
22             New_Line;
23         end;
24
25         procedure Put_Factor_Cnvt (F : Factor) is
26             D : constant Decibel := To_Decibel (F);
27         begin
28             Put ("Factor of ");
29             F_IO.Put (F, 0, 2, 0);
30             Put (" => ");
31             D_IO.Put (D, 0, 2, 0);
32             Put_Line (" dB");
33         end;
34     begin
35         case TC is
36             when Db_Chk =>
37                 Put_Decibel_Cnvt (Decibel (V));
38             when Factor_Chk =>
39                 Put_Factor_Cnvt (Factor (V));
40         end case;
41     end Check;
42
43 begin
44     if Argument_Count < 2 then
```

(continues on next page)

(continued from previous page)

```

45     Put_Line ("ERROR: missing arguments! Exiting...");
46     return;
47   elsif Argument_Count > 2 then
48     Put_Line ("Ignoring additional arguments...");
49   end if;
50
51   Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
52 end Main;

```

17.2 Root-Mean-Square

Goal: implement a function to calculate the root-mean-square of a sequence of values.

Steps:

1. Implement the Signals package.
 1. Implement the Rms function.

Requirements:

1. Subtype Sig_Value is based on a floating-point type.
2. Type Signal is an unconstrained array of Sig_Value elements.
3. Function Rms calculates the RMS of a sequence of values stored in an array of type Signal.
 1. See the remarks below for a description of the RMS calculation.

Remarks:

1. The [root-mean-square](https://en.wikipedia.org/wiki/Root_mean_square)⁵ (RMS) value is an important information associated with sequences of values.
 1. It's used, for example, as a measurement for signal processing.
 2. It is calculated by:
 1. Creating a sequence S with the square of each value of an input sequence S_{in} .
 2. Calculating the mean value M of the sequence S .
 3. Calculating the square-root R of M .
 3. You can optimize the algorithm above by combining steps #1 and #2 into a single step.

Listing 189: signals.ads

```

1 package Signals is
2
3   subtype Sig_Value is Float;
4
5   type Signal is array (Natural range <>) of Sig_Value;
6
7   function Rms (S : Signal) return Sig_Value;
8
9 end Signals;

```

⁵ https://en.wikipedia.org/wiki/Root_mean_square

Listing 190: signals.adb

```
1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Signals is
4
5     function Rms (S : Signal) return Sig_Value is
6     begin
7         return 0.0;
8     end;
9
10 end Signals;
```

Listing 191: signals-std.ads

```
1 package Signals.Std is
2
3     Sample_Rate : Float := 8000.0;
4
5     function Generate_Sine (N : Positive; Freq : Float) return Signal;
6
7     function Generate_Square (N : Positive) return Signal;
8
9     function Generate_Triangular (N : Positive) return Signal;
10
11 end Signals.Std;
```

Listing 192: signals-std.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
3
4 package body Signals.Std is
5
6     function Generate_Sine (N : Positive; Freq : Float) return Signal is
7     S : Signal (0 .. N - 1);
8     begin
9         for I in S'First .. S'Last loop
10             S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
11         end loop;
12
13         return S;
14     end;
15
16     function Generate_Square (N : Positive) return Signal is
17     S : constant Signal (0 .. N - 1) := (others => 1.0);
18     begin
19         return S;
20     end;
21
22     function Generate_Triangular (N : Positive) return Signal is
23     S : Signal (0 .. N - 1);
24     S_Half : constant Natural := S'Last / 2;
25     begin
26         for I in S'First .. S_Half loop
27             S (I) := 1.0 * (Float (I) / Float (S_Half));
28         end loop;
29         for I in S_Half .. S'Last loop
30             S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
31         end loop;
32
33     end;
```

(continues on next page)

(continued from previous page)

```

33     return S;
34 end;
35
36 end Signals.Std;

```

Listing 193: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Signals;               use Signals;
5  with Signals.Std;           use Signals.Std;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Sine_Signal_Chk,
10           Square_Signal_Chk,
11           Triangular_Signal_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14          package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
15
16          N : constant Positive := 1024;
17          S_Si : constant Signal := Generate_Sine (N, 440.0);
18          S_Sq : constant Signal := Generate_Square (N);
19          S_Tr : constant Signal := Generate_Triangular (N + 1);
20      begin
21          case TC is
22              when Sine_Signal_Chk =>
23                  Put ("RMS of Sine Signal: ");
24                  Sig_IO.Put (Rms (S_Si), 0, 2, 0);
25                  New_Line;
26              when Square_Signal_Chk =>
27                  Put ("RMS of Square Signal: ");
28                  Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
29                  New_Line;
30              when Triangular_Signal_Chk =>
31                  Put ("RMS of Triangular Signal: ");
32                  Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
33                  New_Line;
34          end case;
35      end Check;
36
37  begin
38      if Argument_Count < 1 then
39          Put_Line ("ERROR: missing arguments! Exiting...");
40          return;
41      elsif Argument_Count > 1 then
42          Put_Line ("Ignoring additional arguments...");
43      end if;
44
45      Check (Test_Case_Index'Value (Argument (1)));
46  end Main;

```

17.3 Rotation

Goal: use complex numbers to calculate the positions of an object in a circle after rotation.

Steps:

1. Implement the Rotation package.
 1. Implement the Rotation function.

Requirements:

1. Type `Complex_Points` is an unconstrained array of complex values.
2. Function `Rotation` returns a list of positions (represented by the `Complex_Points` type) when dividing a circle in N equal slices.
 1. See the remarks below for a more detailed explanation.
 2. You must use functions from `Ada.Numerics.Complex_Types` to implement `Rotation`.
3. Subtype `Angle` is based on a floating-point type.
4. Type `Angles` is an unconstrained array of angles.
5. Function `To_Angles` returns a list of angles based on an input list of positions.

Remarks:

1. Complex numbers are particularly useful in computer graphics to simplify the calculation of rotations.
 1. For example, let's assume you've drawn an object on your screen on position (1.0, 0.0).
 2. Now, you want to move this object in a circular path — i.e. make it rotate around position (0.0, 0.0) on your screen.
 - You could use *sine* and *cosine* functions to calculate each position of the path.
 - However, you could also calculate the positions using complex numbers.
2. In this exercise, you'll use complex numbers to calculate the positions of an object that starts on zero degrees — on position (1.0, 0.0) — and rotates around (0.0, 0.0) for N slices of a circle.
 1. For example, if we divide the circle in four slices, the object's path will consist of following points / positions:

```
Point #1: ( 1.0,  0.0)
Point #2: ( 0.0,  1.0)
Point #3: (-1.0,  0.0)
Point #4: ( 0.0, -1.0)
Point #5: ( 1.0,  0.0)
```

1. As expected, point #5 is equal to the starting point (point #1), since the object rotates around (0.0, 0.0) and returns to the starting point.
2. We can also describe this path in terms of angles. The following list presents the angles for the path on a four-sliced circle:

```
Point #1:    0.00 degrees
Point #2:   90.00 degrees
Point #3:  180.00 degrees
Point #4: -90.00 degrees (= 270 degrees)
Point #5:    0.00 degrees
```

1. To rotate a complex number simply multiply it by a unit vector whose arg is the radian angle to be rotated: $Z = e^{\frac{2\pi}{N}}$

Listing 194: rotation.ads

```

1 with Ada.Numerics.Complex_Types;
2 use Ada.Numerics.Complex_Types;
3
4 package Rotation is
5
6     type Complex_Points is array (Positive range <>) of Complex;
7
8     function Rotation (N : Positive) return Complex_Points;
9
10 end Rotation;
```

Listing 195: rotation.adb

```

1 with Ada.Numerics; use Ada.Numerics;
2
3 package body Rotation is
4
5     function Rotation (N : Positive) return Complex_Points is
6         C : Complex_Points (1 .. 1) := (others => (0.0, 0.0));
7     begin
8         return C;
9     end;
10
11 end Rotation;
```

Listing 196: angles.ads

```

1 with Rotation; use Rotation;
2
3 package Angles is
4
5     subtype Angle is Float;
6
7     type Angles is array (Positive range <>) of Angle;
8
9     function To_Angles (C : Complex_Points) return Angles;
10
11 end Angles;
```

Listing 197: angles.adb

```

1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Complex_Types; use Ada.Numerics.Complex_Types;
3
4 package body Angles is
5
6     function To_Angles (C : Complex_Points) return Angles is
7     begin
8         return A : Angles (C'Range) do
9             for I in A'Range loop
10                 A (I) := Argument (C (I)) / Pi * 180.0;
11             end loop;
12         end return;
13     end To_Angles;
14
15 end Angles;
```

Listing 198: rotation-tests.ads

```
1 package Rotation.Tests is
2
3   procedure Test_Rotation (N : Positive);
4
5   procedure Test_Angles (N : Positive);
6
7 end Rotation.Tests;
```

Listing 199: rotation-tests.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO;
3 with Ada.Numerics;          use Ada.Numerics;
4
5 with Angles;                 use Angles;
6
7 package body Rotation.Tests is
8
9   package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
10  package F_IO is new Ada.Text_IO.Float_IO (Float);
11
12  --
13  -- Adapt value due to floating-point inaccuracies
14  --
15
16  function Adapt (C : Complex) return Complex is
17    function Check_Zero (F : Float) return Float is
18      (if F <= 0.0 and F >= -0.01 then 0.0 else F);
19  begin
20    return C_Out : Complex := C do
21      C_Out.Re := Check_Zero (C_Out.Re);
22      C_Out.Im := Check_Zero (C_Out.Im);
23    end return;
24  end Adapt;
25
26  function Adapt (A : Angle) return Angle is
27    (if A <= -179.99 and A >= -180.01 then 180.0 else A);
28
29  procedure Test_Rotation (N : Positive) is
30    C : constant Complex_Points := Rotation (N);
31  begin
32    Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
33    for V of C loop
34      Put ("Point: ");
35      C_IO.Put (Adapt (V), 0, 1, 0);
36      New_Line;
37    end loop;
38  end Test_Rotation;
39
40  procedure Test_Angles (N : Positive) is
41    C : constant Complex_Points := Rotation (N);
42    A : constant Angles.Angles := To_Angles (C);
43  begin
44    Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
45    for V of A loop
46      Put ("Angle: ");
47      F_IO.Put (Adapt (V), 0, 2, 0);
48      Put_Line (" degrees");
49    end loop;
```

(continues on next page)

(continued from previous page)

```

50     end Test_Angles;
51
52 end Rotation.Tests;

```

Listing 200: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;          use Ada.Text_IO;
3
4  with Rotation.Tests;        use Rotation.Tests;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Rotation_Chk,
9           Angles_Chk);
10
11     procedure Check (TC : Test_Case_Index; N : Positive) is
12     begin
13         case TC is
14             when Rotation_Chk =>
15                 Test_Rotation (N);
16             when Angles_Chk =>
17                 Test_Angles (N);
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 2 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 2 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
30 end Main;

```


SOLUTIONS

18.1 Imperative Language

18.1.1 Hello World

Listing 201: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     Put_Line ("Hello World!");
6 end Main;
```

18.1.2 Greetings

Listing 202: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Greet (Name : String) is
7     begin
8         Put_Line ("Hello " & Name & "!");
9     end Greet;
10
11 begin
12     if Argument_Count < 1 then
13         Put_Line ("ERROR: missing arguments! Exiting...");
14         return;
15     elsif Argument_Count > 1 then
16         Put_Line ("Ignoring additional arguments...");
17     end if;
18
19     Greet (Argument (1));
20 end Main;
```

18.1.3 Positive Or Negative

Listing 203: classify_number.ads

```
1 procedure Classify_Number (X : Integer);
```

Listing 204: classify_number.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5     if X > 0 then
6         Put_Line ("Positive");
7     elsif X < 0 then
8         Put_Line ("Negative");
9     else
10        Put_Line ("Zero");
11    end if;
12 end Classify_Number;
```

Listing 205: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Classify_Number;
5
6 procedure Main is
7     A : Integer;
8 begin
9     if Argument_Count < 1 then
10        Put_Line ("ERROR: missing arguments! Exiting...");
11        return;
12    elsif Argument_Count > 1 then
13        Put_Line ("Ignoring additional arguments...");
14    end if;
15
16    A := Integer'Value (Argument (1));
17
18    Classify_Number (A);
19 end Main;
```

18.1.4 Numbers

Listing 206: display_numbers.ads

```
1 procedure Display_Numbers (A, B : Integer);
```

Listing 207: display_numbers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Numbers (A, B : Integer) is
4     X, Y : Integer;
5 begin
6     if A <= B then
7         X := A;
8         Y := B;
9     else
10        X := B;
11        Y := A;
12    end if;
13
14    for I in X .. Y loop
```

(continues on next page)

(continued from previous page)

```

15     Put_Line (Integer'Image (I));
16 end loop;
17 end Display_Numbers;

```

Listing 208: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Display_Numbers;
5
6  procedure Main is
7      A, B : Integer;
8  begin
9      if Argument_Count < 2 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 2 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     A := Integer'Value (Argument (1));
17     B := Integer'Value (Argument (2));
18
19     Display_Numbers (A, B);
20 end Main;

```

18.2 Subprograms

18.2.1 Subtract Procedure

Listing 209: subtract.ads

```

1  procedure Subtract (A, B : Integer;
2                      Result : out Integer);

```

Listing 210: subtract.adb

```

1  procedure Subtract (A, B : Integer;
2                      Result : out Integer) is
3  begin
4      Result := A - B;
5  end Subtract;

```

Listing 211: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Subtract;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Sub_10_1_Chk,
9           Sub_10_100_Chk,
10          Sub_0_5_Chk,
11          Sub_0_Minus_5_Chk);

```

(continues on next page)

(continued from previous page)

```

12
13 procedure Check (TC : Test_Case_Index) is
14     Result : Integer;
15 begin
16     case TC is
17     when Sub_10_1_Chk =>
18         Subtract (10, 1, Result);
19         Put_Line ("Result: " & Integer'Image (Result));
20     when Sub_10_100_Chk =>
21         Subtract (10, 100, Result);
22         Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Subtract (0, 5, Result);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Subtract (0, -5, Result);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

18.2.2 Subtract Function

Listing 212: subtract.ads

```

1 function Subtract (A, B : Integer) return Integer;

```

Listing 213: subtract.adb

```

1 function Subtract (A, B : Integer) return Integer is
2 begin
3     return A - B;
4 end Subtract;

```

Listing 214: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Subtract;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Sub_10_1_Chk,
9          Sub_10_100_Chk,
10         Sub_0_5_Chk,
11         Sub_0_Minus_5_Chk);
12
13     procedure Check (TC : Test_Case_Index) is

```

(continues on next page)

(continued from previous page)

```

14     Result : Integer;
15 begin
16     case TC is
17     when Sub_10_1_Chk =>
18         Result := Subtract (10, 1);
19         Put_Line ("Result: " & Integer'Image (Result));
20     when Sub_10_100_Chk =>
21         Result := Subtract (10, 100);
22         Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Result := Subtract (0, 5);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Result := Subtract (0, -5);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

18.2.3 Equality function

Listing 215: is_equal.ads

```

1 function Is_Equal (A, B : Integer) return Boolean;

```

Listing 216: is_equal.adb

```

1 function Is_Equal (A, B : Integer) return Boolean is
2 begin
3     return A = B;
4 end Is_Equal;

```

Listing 217: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Is_Equal;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Equal_Chk,
9          Inequal_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Display_Equal (A, B : Integer;
14                                 Equal : Boolean) is
15         begin

```

(continues on next page)

(continued from previous page)

```

16     Put (Integer'Image (A));
17     if Equal then
18         Put (" is equal to ");
19     else
20         Put (" isn't equal to ");
21     end if;
22     Put_Line (Integer'Image (B) & ".");
23 end Display_Equal;
24
25 Result : Boolean;
26 begin
27     case TC is
28     when Equal_Chk =>
29         for I in 0 .. 10 loop
30             Result := Is_Equal (I, I);
31             Display_Equal (I, I, Result);
32         end loop;
33     when Inequal_Chk =>
34         for I in 0 .. 10 loop
35             Result := Is_Equal (I, I - 1);
36             Display_Equal (I, I - 1, Result);
37         end loop;
38     end case;
39 end Check;
40
41 begin
42     if Argument_Count < 1 then
43         Put_Line ("ERROR: missing arguments! Exiting...");
44         return;
45     elsif Argument_Count > 1 then
46         Put_Line ("Ignoring additional arguments...");
47     end if;
48
49     Check (Test_Case_Index'Value (Argument (1)));
50 end Main;

```

18.2.4 States

Listing 218: display_state.ads

```

1 procedure Display_State (State : Integer);

```

Listing 219: display_state.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_State (State : Integer) is
4 begin
5     case State is
6     when 0 =>
7         Put_Line ("Off");
8     when 1 =>
9         Put_Line ("On: Simple Processing");
10    when 2 =>
11        Put_Line ("On: Advanced Processing");
12    when others =>
13        null;
14    end case;
15 end Display_State;

```

Listing 220: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_State;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Display_State (State);
19 end Main;

```

18.2.5 States #2

Listing 221: get_state.ads

```

1 function Get_State (State : Integer) return String;

```

Listing 222: get_state.adb

```

1 function Get_State (State : Integer) return String is
2 begin
3     return (case State is
4         when 0 => "Off",
5         when 1 => "On: Simple Processing",
6         when 2 => "On: Advanced Processing",
7         when others => "");
8 end Get_State;

```

Listing 223: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Get_State;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Put_Line (Get_State (State));
19 end Main;

```

18.2.6 States #3

Listing 224: is_on.ads

```
1 function Is_On (State : Integer) return Boolean;
```

Listing 225: is_on.adb

```
1 function Is_On (State : Integer) return Boolean is
2 begin
3     return not (State = 0);
4 end Is_On;
```

Listing 226: display_on_off.ads

```
1 procedure Display_On_Off (State : Integer);
```

Listing 227: display_on_off.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Is_On;
3
4 procedure Display_On_Off (State : Integer) is
5 begin
6     Put_Line (if Is_On (State) then "On" else "Off");
7 end Display_On_Off;
```

Listing 228: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_On_Off;
5 with Is_On;
6
7 procedure Main is
8     State : Integer;
9 begin
10     if Argument_Count < 1 then
11         Put_Line ("ERROR: missing arguments! Exiting...");
12         return;
13     elsif Argument_Count > 1 then
14         Put_Line ("Ignoring additional arguments...");
15     end if;
16
17     State := Integer'Value (Argument (1));
18
19     Display_On_Off (State);
20     Put_Line (Boolean'Image (Is_On (State)));
21 end Main;
```

18.2.7 States #4

Listing 229: set_next.ads

```
1 procedure Set_Next (State : in out Integer);
```

Listing 230: set_next.adb

```

1 procedure Set_Next (State : in out Integer) is
2 begin
3   State := (if State < 2 then State + 1 else 0);
4 end Set_Next;

```

Listing 231: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Set_Next;
5
6 procedure Main is
7   State : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");
14   end if;
15
16   State := Integer'Value (Argument (1));
17
18   Set_Next (State);
19   Put_Line (Integer'Image (State));
20 end Main;

```

18.3 Modular Programming

18.3.1 Months

Listing 232: months.ads

```

1 package Months is
2
3   Jan : constant String := "January";
4   Feb : constant String := "February";
5   Mar : constant String := "March";
6   Apr : constant String := "April";
7   May : constant String := "May";
8   Jun : constant String := "June";
9   Jul : constant String := "July";
10  Aug : constant String := "August";
11  Sep : constant String := "September";
12  Oct : constant String := "October";
13  Nov : constant String := "November";
14  Dec : constant String := "December";
15
16  procedure Display_Months;
17
18 end Months;

```

Listing 233: months.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Months is
4
5     procedure Display_Months is
6     begin
7         Put_Line ("Months:");
8         Put_Line ("- " & Jan);
9         Put_Line ("- " & Feb);
10        Put_Line ("- " & Mar);
11        Put_Line ("- " & Apr);
12        Put_Line ("- " & May);
13        Put_Line ("- " & Jun);
14        Put_Line ("- " & Jul);
15        Put_Line ("- " & Aug);
16        Put_Line ("- " & Sep);
17        Put_Line ("- " & Oct);
18        Put_Line ("- " & Nov);
19        Put_Line ("- " & Dec);
20    end Display_Months;
21
22 end Months;
```

Listing 234: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Months;            use Months;
5
6 procedure Main is
7
8     type Test_Case_Index is
9         (Months_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Months_Chk =>
15                 Display_Months;
16         end case;
17     end Check;
18
19 begin
20     if Argument_Count < 1 then
21         Put_Line ("ERROR: missing arguments! Exiting...");
22         return;
23     elsif Argument_Count > 1 then
24         Put_Line ("Ignoring additional arguments...");
25     end if;
26
27     Check (Test_Case_Index'Value (Argument (1)));
28 end Main;
```

18.3.2 Operations

Listing 235: operations.ads

```

1 package Operations is
2
3     function Add (A, B : Integer) return Integer;
4
5     function Subtract (A, B : Integer) return Integer;
6
7     function Multiply (A, B : Integer) return Integer;
8
9     function Divide (A, B : Integer) return Integer;
10
11 end Operations;
```

Listing 236: operations.adb

```

1 package body Operations is
2
3     function Add (A, B : Integer) return Integer is
4     begin
5         return A + B;
6     end Add;
7
8     function Subtract (A, B : Integer) return Integer is
9     begin
10        return A - B;
11    end Subtract;
12
13    function Multiply (A, B : Integer) return Integer is
14    begin
15        return A * B;
16    end Multiply;
17
18    function Divide (A, B : Integer) return Integer is
19    begin
20        return A / B;
21    end Divide;
22
23 end Operations;
```

Listing 237: operations-test.ads

```

1 package Operations.Test is
2
3     procedure Display (A, B : Integer);
4
5 end Operations.Test;
```

Listing 238: operations-test.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Operations.Test is
4
5     procedure Display (A, B : Integer) is
6         A_Str : constant String := Integer'Image (A);
7         B_Str : constant String := Integer'Image (B);
8     begin
9         Put_Line ("Operations:");
```

(continues on next page)

(continued from previous page)

```

10     Put_Line (A_Str & " + " & B_Str & " = "
11               & Integer'Image (Add (A, B))
12               & ",");
13     Put_Line (A_Str & " - " & B_Str & " = "
14               & Integer'Image (Subtract (A, B))
15               & ",");
16     Put_Line (A_Str & " * " & B_Str & " = "
17               & Integer'Image (Multiply (A, B))
18               & ",");
19     Put_Line (A_Str & " / " & B_Str & " = "
20               & Integer'Image (Divide (A, B))
21               & ",");
22 end Display;
23
24 end Operations.Test;

```

Listing 239: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Operations;
5  with Operations.Test; use Operations.Test;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Operations_Chk,
11          Operations_Display_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when Operations_Chk =>
17                 Put_Line ("Add (100, 2) = "
18                           & Integer'Image (Operations.Add (100, 2)));
19                 Put_Line ("Subtract (100, 2) = "
20                           & Integer'Image (Operations.Subtract (100, 2)));
21                 Put_Line ("Multiply (100, 2) = "
22                           & Integer'Image (Operations.Multiply (100, 2)));
23                 Put_Line ("Divide (100, 2) = "
24                           & Integer'Image (Operations.Divide (100, 2)));
25             when Operations_Display_Chk =>
26                 Display (10, 5);
27                 Display (1, 2);
28         end case;
29     end Check;
30
31 begin
32     if Argument_Count < 1 then
33         Put_Line ("ERROR: missing arguments! Exiting...");
34         return;
35     elsif Argument_Count > 1 then
36         Put_Line ("Ignoring additional arguments...");
37     end if;
38
39     Check (Test_Case_Index'Value (Argument (1)));
40 end Main;

```

18.4 Strongly typed language

18.4.1 Colors

Listing 240: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   function To_Integer (C : HTML_Color) return Integer;
17
18   type Basic_HTML_Color is
19     (Red,
20     Green,
21     Blue);
22
23   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25 end Color_Types;
```

Listing 241: color_types.adb

```

1 package body Color_Types is
2
3   function To_Integer (C : HTML_Color) return Integer is
4   begin
5     case C is
6       when Salmon    => return 16#FA8072#;
7       when Firebrick => return 16#B22222#;
8       when Red       => return 16#FF0000#;
9       when Darkred   => return 16#8B0000#;
10      when Lime       => return 16#00FF00#;
11      when Forestgreen => return 16#228B22#;
12      when Green      => return 16#008000#;
13      when Darkgreen  => return 16#006400#;
14      when Blue       => return 16#0000FF#;
15      when Mediumblue => return 16#0000CD#;
16      when Darkblue   => return 16#00008B#;
17    end case;
18
19   end To_Integer;
20
21   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
22   begin
23     case C is
24       when Red    => return Red;
25       when Green  => return Green;
26       when Blue   => return Blue;
27    end case;
28
```

(continues on next page)

(continued from previous page)

```

28     end To_HTML_Color;
29
30 end Color_Types;

```

Listing 242: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Integer_Text_IO;
4
5  with Color_Types;      use Color_Types;
6
7  procedure Main is
8      type Test_Case_Index is
9          (HTML_Color_Range,
10           HTML_Color_To_Integer,
11           Basic_HTML_Color_To_HTML_Color);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when HTML_Color_Range =>
17                 for I in HTML_Color'Range loop
18                     Put_Line (HTML_Color'Image (I));
19                 end loop;
20             when HTML_Color_To_Integer =>
21                 for I in HTML_Color'Range loop
22                     Ada.Integer_Text_IO.Put (Item => To_Integer (I),
23                                             Width => 1,
24                                             Base  => 16);
25                     New_Line;
26                 end loop;
27             when Basic_HTML_Color_To_HTML_Color =>
28                 for I in Basic_HTML_Color'Range loop
29                     Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30                 end loop;
31             end case;
32     end Check;
33
34 begin
35     if Argument_Count < 1 then
36         Put_Line ("ERROR: missing arguments! Exiting...");
37         return;
38     elsif Argument_Count > 1 then
39         Put_Line ("Ignoring additional arguments...");
40     end if;
41
42     Check (Test_Case_Index'Value (Argument (1)));
43 end Main;

```

18.4.2 Integers

Listing 243: int_types.ads

```

1  package Int_Types is
2
3      type I_100 is range 0 .. 100;
4
5      type U_100 is mod 101;

```

(continues on next page)

(continued from previous page)

```

6
7  function To_I_100 (V : U_100) return I_100;
8
9  function To_U_100 (V : I_100) return U_100;
10
11 type D_50 is new I_100 range 10 .. 50;
12
13 subtype S_50 is I_100 range 10 .. 50;
14
15 function To_D_50 (V : I_100) return D_50;
16
17 function To_S_50 (V : I_100) return S_50;
18
19 function To_I_100 (V : D_50) return I_100;
20
21 end Int_Types;

```

Listing 244: int_types.adb

```

1  package body Int_Types is
2
3      function To_I_100 (V : U_100) return I_100 is
4      begin
5          return I_100 (V);
6      end To_I_100;
7
8      function To_U_100 (V : I_100) return U_100 is
9      begin
10         return U_100 (V);
11     end To_U_100;
12
13     function To_D_50 (V : I_100) return D_50 is
14     Min : constant I_100 := I_100 (D_50'First);
15     Max : constant I_100 := I_100 (D_50'Last);
16     begin
17         if V > Max then
18             return D_50'Last;
19         elsif V < Min then
20             return D_50'First;
21         else
22             return D_50 (V);
23         end if;
24     end To_D_50;
25
26     function To_S_50 (V : I_100) return S_50 is
27     begin
28         if V > S_50'Last then
29             return S_50'Last;
30         elsif V < S_50'First then
31             return S_50'First;
32         else
33             return V;
34         end if;
35     end To_S_50;
36
37     function To_I_100 (V : D_50) return I_100 is
38     begin
39         return I_100 (V);
40     end To_I_100;
41
42 end Int_Types;

```

Listing 245: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Int_Types;        use Int_Types;
5
6  procedure Main is
7      package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
8      package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
9      package D_50_IO  is new Ada.Text_IO.Integer_IO (D_50);
10
11     use I_100_IO;
12     use U_100_IO;
13     use D_50_IO;
14
15     type Test_Case_Index is
16         (I_100_Range,
17          U_100_Range,
18          U_100_Wraparound,
19          U_100_To_I_100,
20          I_100_To_U_100,
21          D_50_Range,
22          S_50_Range,
23          I_100_To_D_50,
24          I_100_To_S_50,
25          D_50_To_I_100,
26          S_50_To_I_100);
27
28     procedure Check (TC : Test_Case_Index) is
29     begin
30         I_100_IO.Default_Width := 1;
31         U_100_IO.Default_Width := 1;
32         D_50_IO.Default_Width  := 1;
33
34         case TC is
35             when I_100_Range =>
36                 Put (I_100'First);
37                 New_Line;
38                 Put (I_100'Last);
39                 New_Line;
40             when U_100_Range =>
41                 Put (U_100'First);
42                 New_Line;
43                 Put (U_100'Last);
44                 New_Line;
45             when U_100_Wraparound =>
46                 Put (U_100'First - 1);
47                 New_Line;
48                 Put (U_100'Last + 1);
49                 New_Line;
50             when U_100_To_I_100 =>
51                 for I in U_100'Range loop
52                     I_100_IO.Put (To_I_100 (I));
53                     New_Line;
54                 end loop;
55             when I_100_To_U_100 =>
56                 for I in I_100'Range loop
57                     Put (To_U_100 (I));
58                     New_Line;
59                 end loop;

```

(continues on next page)

(continued from previous page)

```

60     when D_50_Range =>
61         Put (D_50'First);
62         New_Line;
63         Put (D_50'Last);
64         New_Line;
65     when S_50_Range =>
66         Put (S_50'First);
67         New_Line;
68         Put (S_50'Last);
69         New_Line;
70     when I_100_To_D_50 =>
71         for I in I_100'Range loop
72             Put (To_D_50 (I));
73             New_Line;
74         end loop;
75     when I_100_To_S_50 =>
76         for I in I_100'Range loop
77             Put (To_S_50 (I));
78             New_Line;
79         end loop;
80     when D_50_To_I_100 =>
81         for I in D_50'Range loop
82             Put (To_I_100 (I));
83             New_Line;
84         end loop;
85     when S_50_To_I_100 =>
86         for I in S_50'Range loop
87             Put (I);
88             New_Line;
89         end loop;
90     end case;
91 end Check;
92
93 begin
94     if Argument_Count < 1 then
95         Put_Line ("ERROR: missing arguments! Exiting...");
96         return;
97     elsif Argument_Count > 1 then
98         Put_Line ("Ignoring additional arguments...");
99     end if;
100
101     Check (Test_Case_Index'Value (Argument (1)));
102 end Main;

```

18.4.3 Temperatures

Listing 246: temperature_types.ads

```

1 package Temperature_Types is
2
3     type Celsius is digits 6 range -273.15 .. 5504.85;
4
5     type Int_Celsius is range -273 .. 5505;
6
7     function To_Celsius (T : Int_Celsius) return Celsius;
8
9     function To_Int_Celsius (T : Celsius) return Int_Celsius;
10
11     type Kelvin is digits 6 range 0.0 .. 5778.00;
12

```

(continues on next page)

(continued from previous page)

```

13  function To_Celsius (T : Kelvin) return Celsius;
14
15  function To_Kelvin (T : Celsius) return Kelvin;
16
17  end Temperature_Types;

```

Listing 247: temperature_types.adb

```

1  package body Temperature_Types is
2
3      function To_Celsius (T : Int_Celsius) return Celsius is
4          Min : constant Float := Float (Celsius'First);
5          Max : constant Float := Float (Celsius'Last);
6
7          F : constant Float := Float (T);
8      begin
9          if F > Max then
10             return Celsius (Max);
11          elsif F < Min then
12             return Celsius (Min);
13          else
14             return Celsius (F);
15          end if;
16      end To_Celsius;
17
18      function To_Int_Celsius (T : Celsius) return Int_Celsius is
19      begin
20          return Int_Celsius (T);
21      end To_Int_Celsius;
22
23      function To_Celsius (T : Kelvin) return Celsius is
24          F : constant Float := Float (T);
25      begin
26          return Celsius (F - 273.15);
27      end To_Celsius;
28
29      function To_Kelvin (T : Celsius) return Kelvin is
30          F : constant Float := Float (T);
31      begin
32          return Kelvin (F + 273.15);
33      end To_Kelvin;
34
35  end Temperature_Types;

```

Listing 248: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Temperature_Types; use Temperature_Types;
5
6  procedure Main is
7      package Celsius_IO is new Ada.Text_IO.Float_IO (Celsius);
8      package Kelvin_IO is new Ada.Text_IO.Float_IO (Kelvin);
9      package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);
10
11      use Celsius_IO;
12      use Kelvin_IO;
13      use Int_Celsius_IO;
14

```

(continues on next page)

(continued from previous page)

```

15  type Test_Case_Index is
16      (Celsius_Range,
17       Celsius_To_Int_Celsius,
18       Int_Celsius_To_Celsius,
19       Kelvin_To_Celsius,
20       Celsius_To_Kelvin);
21
22  procedure Check (TC : Test_Case_Index) is
23  begin
24      Celsius_IO.Default_Fore := 1;
25      Kelvin_IO.Default_Fore  := 1;
26      Int_Celsius_IO.Default_Width := 1;
27
28      case TC is
29          when Celsius_Range =>
30              Put (Celsius'First);
31              New_Line;
32              Put (Celsius'Last);
33              New_Line;
34          when Celsius_To_Int_Celsius =>
35              Put (To_Int_Celsius (Celsius'First));
36              New_Line;
37              Put (To_Int_Celsius (0.0));
38              New_Line;
39              Put (To_Int_Celsius (Celsius'Last));
40              New_Line;
41          when Int_Celsius_To_Celsius =>
42              Put (To_Celsius (Int_Celsius'First));
43              New_Line;
44              Put (To_Celsius (0));
45              New_Line;
46              Put (To_Celsius (Int_Celsius'Last));
47              New_Line;
48          when Kelvin_To_Celsius =>
49              Put (To_Celsius (Kelvin'First));
50              New_Line;
51              Put (To_Celsius (0));
52              New_Line;
53              Put (To_Celsius (Kelvin'Last));
54              New_Line;
55          when Celsius_To_Kelvin =>
56              Put (To_Kelvin (Celsius'First));
57              New_Line;
58              Put (To_Kelvin (Celsius'Last));
59              New_Line;
60      end case;
61  end Check;
62
63  begin
64      if Argument_Count < 1 then
65          Put_Line ("ERROR: missing arguments! Exiting...");
66          return;
67      elsif Argument_Count > 1 then
68          Put_Line ("Ignoring additional arguments...");
69      end if;
70
71      Check (Test_Case_Index'Value (Argument (1)));
72  end Main;

```


18.5 Records

18.5.1 Directions

Listing 249: directions.ads

```
1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northeast,
8      East,
9      Southeast,
10     South,
11     Southwest,
12     West,
13     Northwest);
14
15   function To_Direction (N: Angle_Mod) return Direction;
16
17   type Ext_Angle is record
18     Angle_Elem      : Angle_Mod;
19     Direction_Elem  : Direction;
20   end record;
21
22   function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
23
24   procedure Display (N : Ext_Angle);
25
26 end Directions;
```

Listing 250: directions.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Directions is
4
5   procedure Display (N : Ext_Angle) is
6   begin
7     Put_Line ("Angle: "
8              & Angle_Mod'Image (N.Angle_Elem)
9              & " => "
10             & Direction'Image (N.Direction_Elem)
11             & ".");
12   end Display;
13
14   function To_Direction (N : Angle_Mod) return Direction is
15   begin
16     case N is
17       when 0      => return North;
18       when 1 .. 89 => return Northeast;
19       when 90     => return East;
20       when 91 .. 179 => return Southeast;
21       when 180    => return South;
22       when 181 .. 269 => return Southwest;
23       when 270    => return West;
24       when 271 .. 359 => return Northwest;
25     end case;
26   end To_Direction;
```

(continues on next page)

(continued from previous page)

```

27
28   function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29   begin
30       return (Angle_Elem    => N,
31              Direction_Elem => To_Direction (N));
32   end To_Ext_Angle;
33
34 end Directions;

```

Listing 251: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Directions;       use Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13         when Direction_Chk =>
14             Display (To_Ext_Angle (0));
15             Display (To_Ext_Angle (30));
16             Display (To_Ext_Angle (45));
17             Display (To_Ext_Angle (90));
18             Display (To_Ext_Angle (91));
19             Display (To_Ext_Angle (120));
20             Display (To_Ext_Angle (180));
21             Display (To_Ext_Angle (250));
22             Display (To_Ext_Angle (270));
23         end case;
24     end Check;
25
26     begin
27         if Argument_Count < 1 then
28             Put_Line ("ERROR: missing arguments! Exiting...");
29             return;
30         elsif Argument_Count > 1 then
31             Put_Line ("Ignoring additional arguments...");
32         end if;
33
34         Check (Test_Case_Index'Value (Argument (1)));
35     end Main;

```

18.5.2 Colors

Listing 252: color_types.ads

```

1  package Color_Types is
2
3      type HTML_Color is
4          (Salmon,
5           Firebrick,
6           Red,
7           Darkred,
8           Lime,

```

(continues on next page)

(continued from previous page)

```

9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16     function To_Integer (C : HTML_Color) return Integer;
17
18     type Basic_HTML_Color is
19       (Red,
20        Green,
21        Blue);
22
23     function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25     subtype Int_Color is Integer range 0 .. 255;
26
27     type RGB is record
28       Red   : Int_Color;
29       Green : Int_Color;
30       Blue  : Int_Color;
31     end record;
32
33     function To_RGB (C : HTML_Color) return RGB;
34
35     function Image (C : RGB) return String;
36
37 end Color_Types;

```

Listing 253: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_Integer (C : HTML_Color) return Integer is
6      begin
7          case C is
8              when Salmon    => return 16#FA8072#;
9              when Firebrick => return 16#B22222#;
10             when Red        => return 16#FF0000#;
11             when Darkred    => return 16#8B0000#;
12             when Lime       => return 16#00FF00#;
13             when Forestgreen => return 16#228B22#;
14             when Green      => return 16#008000#;
15             when Darkgreen  => return 16#006400#;
16             when Blue       => return 16#0000FF#;
17             when Mediumblue => return 16#0000CD#;
18             when Darkblue   => return 16#00008B#;
19          end case;
20
21      end To_Integer;
22
23      function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
24      begin
25          case C is
26              when Red    => return Red;
27              when Green  => return Green;
28              when Blue   => return Blue;
29          end case;

```

(continues on next page)

(continued from previous page)

```

30 end To_HTML_Color;
31
32 function To_RGB (C : HTML_Color) return RGB is
33 begin
34     case C is
35         when Salmon      => return (16#FA#, 16#80#, 16#72#);
36         when Firebrick   => return (16#B2#, 16#22#, 16#22#);
37         when Red         => return (16#FF#, 16#00#, 16#00#);
38         when Darkred     => return (16#8B#, 16#00#, 16#00#);
39         when Lime        => return (16#00#, 16#FF#, 16#00#);
40         when Forestgreen => return (16#22#, 16#8B#, 16#22#);
41         when Green       => return (16#00#, 16#80#, 16#00#);
42         when Darkgreen   => return (16#00#, 16#64#, 16#00#);
43         when Blue        => return (16#00#, 16#00#, 16#FF#);
44         when Mediumblue  => return (16#00#, 16#00#, 16#CD#);
45         when Darkblue    => return (16#00#, 16#00#, 16#8B#);
46     end case;
47
48 end To_RGB;
49
50 function Image (C : RGB) return String is
51     subtype Str_Range is Integer range 1 .. 10;
52     SR : String (Str_Range);
53     SG : String (Str_Range);
54     SB : String (Str_Range);
55 begin
56     Ada.Integer_Text_IO.Put (To  => SR,
57                             Item => C.Red,
58                             Base => 16);
59     Ada.Integer_Text_IO.Put (To  => SG,
60                             Item => C.Green,
61                             Base => 16);
62     Ada.Integer_Text_IO.Put (To  => SB,
63                             Item => C.Blue,
64                             Base => 16);
65     return ("(Red => " & SR
66           & ", Green => " & SG
67           & ", Blue => " & SB
68           & ")");
69 end Image;
70
71 end Color_Types;

```

Listing 254: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Color_Types;      use Color_Types;
5
6 procedure Main is
7     type Test_Case_Index is
8         (HTML_Color_To_RGB);
9
10    procedure Check (TC : Test_Case_Index) is
11    begin
12        case TC is
13            when HTML_Color_To_RGB =>
14                for I in HTML_Color'Range loop
15                    Put_Line (HTML_Color'Image (I) & " => "
16                            & Image (To_RGB (I)) & ".");

```

(continues on next page)

(continued from previous page)

```

17         end loop;
18     end case;
19 end Check;
20
21 begin
22     if Argument_Count < 1 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 1 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)));
30 end Main;

```

18.5.3 Inventory

Listing 255: inventory_pkg.ads

```

1 package Inventory_Pkg is
2
3     type Item_Name is
4         (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);
5
6     function To_String (I : Item_Name) return String;
7
8     type Item is record
9         Name      : Item_Name;
10        Quantity : Natural;
11        Price     : Float;
12    end record;
13
14    function Init (Name      : Item_Name;
15                  Quantity  : Natural;
16                  Price     : Float) return Item;
17
18    procedure Add (Assets : in out Float;
19                  I       : Item);
20
21 end Inventory_Pkg;

```

Listing 256: inventory_pkg.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Inventory_Pkg is
4
5     function To_String (I : Item_Name) return String is
6     begin
7         case I is
8             when Ballpoint_Pen      => return "Ballpoint Pen";
9             when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
10            when Feather_Quill_Pen   => return "Feather Quill Pen";
11        end case;
12    end To_String;
13
14    function Init (Name      : Item_Name;
15                  Quantity  : Natural;
16                  Price     : Float) return Item is

```

(continues on next page)

(continued from previous page)

```

17  begin
18      Put_Line ("Item: " & To_String (Name) & ".");
19
20      return (Name      => Name,
21              Quantity => Quantity,
22              Price     => Price);
23  end Init;
24
25  procedure Add (Assets : in out Float;
26                I       : Item) is
27  begin
28      Assets := Assets + Float (I.Quantity) * I.Price;
29  end Add;
30
31  end Inventory_Pkg;

```

Listing 257: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Inventory_Pkg;     use Inventory_Pkg;
5
6  procedure Main is
7      -- Remark: the following line is not relevant.
8      F : array (1 .. 10) of Float := (others => 42.42);
9
10     type Test_Case_Index is
11         (Inventory_Chk);
12
13     procedure Display (Assets : Float) is
14         package F_IO is new Ada.Text_IO.Float_IO (Float);
15
16         use F_IO;
17     begin
18         Put ("Assets: $");
19         Put (Assets, 1, 2, 0);
20         Put (".");
21         New_Line;
22     end Display;
23
24     procedure Check (TC : Test_Case_Index) is
25         I : Item;
26         Assets : Float := 0.0;
27
28         -- Please ignore the following three lines!
29         pragma Warnings (Off, "default initialization");
30         for Assets'Address use F'Address;
31         pragma Warnings (On, "default initialization");
32     begin
33         case TC is
34         when Inventory_Chk =>
35             I := Init (Ballpoint_Pen, 185, 0.15);
36             Add (Assets, I);
37             Display (Assets);
38
39             I := Init (Oil_Based_Pen_Marker, 100, 9.0);
40             Add (Assets, I);
41             Display (Assets);
42
43             I := Init (Feather_Quill_Pen, 2, 40.0);

```

(continues on next page)

(continued from previous page)

```

44     Add (Assets, I);
45     Display (Assets);
46   end case;
47 end Check;
48
49 begin
50   if Argument_Count < 1 then
51     Put_Line ("ERROR: missing arguments! Exiting...");
52     return;
53   elsif Argument_Count > 1 then
54     Put_Line ("Ignoring additional arguments...");
55   end if;
56
57   Check (Test_Case_Index'Value (Argument (1)));
58 end Main;

```

18.6 Arrays

18.6.1 Constrained Array

Listing 258: constrained_arrays.ads

```

1 package Constrained_Arrays is
2
3   type My_Index is range 1 .. 10;
4
5   type My_Array is array (My_Index) of Integer;
6
7   function Init return My_Array;
8
9   procedure Double (A : in out My_Array);
10
11  function First_Elem (A : My_Array) return Integer;
12
13  function Last_Elem (A : My_Array) return Integer;
14
15  function Length (A : My_Array) return Integer;
16
17  A : My_Array := (1, 2, others => 42);
18
19 end Constrained_Arrays;

```

Listing 259: constrained_arrays.adb

```

1 package body Constrained_Arrays is
2
3   function Init return My_Array is
4     A : My_Array;
5   begin
6     for I in My_Array'Range loop
7       A (I) := Integer (I);
8     end loop;
9
10    return A;
11  end Init;
12
13  procedure Double (A : in out My_Array) is
14  begin

```

(continues on next page)

(continued from previous page)

```

15     for I in A'Range loop
16         A (I) := A (I) * 2;
17     end loop;
18 end Double;
19
20 function First_Elem (A : My_Array) return Integer is
21 begin
22     return A (A'First);
23 end First_Elem;
24
25 function Last_Elem (A : My_Array) return Integer is
26 begin
27     return A (A'Last);
28 end Last_Elem;
29
30 function Length (A : My_Array) return Integer is
31 begin
32     return A'Length;
33 end Length;
34
35 end Constrained_Arrays;

```

Listing 260: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Constrained_Arrays; use Constrained_Arrays;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Range_Chk,
9           Array_Range_Chk,
10          A_Obj_Chk,
11          Init_Chk,
12          Double_Chk,
13          First_Elem_Chk,
14          Last_Elem_Chk,
15          Length_Chk);
16
17  procedure Check (TC : Test_Case_Index) is
18      AA : My_Array;
19
20      procedure Display (A : My_Array) is
21      begin
22          for I in A'Range loop
23              Put_Line (Integer'Image (A (I)));
24          end loop;
25      end Display;
26
27      procedure Local_Init (A : in out My_Array) is
28      begin
29          A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
30      end Local_Init;
31  begin
32      case TC is
33      when Range_Chk =>
34          for I in My_Index loop
35              Put_Line (My_Index'Image (I));
36          end loop;
37      when Array_Range_Chk =>

```

(continues on next page)

(continued from previous page)

```

38     for I in My_Array'Range loop
39         Put_Line (My_Index'Image (I));
40     end loop;
41     when A_Obj_Chk =>
42         Display (A);
43     when Init_Chk =>
44         AA := Init;
45         Display (AA);
46     when Double_Chk =>
47         Local_Init (AA);
48         Double (AA);
49         Display (AA);
50     when First_Elem_Chk =>
51         Local_Init (AA);
52         Put_Line (Integer'Image (First_Elem (AA)));
53     when Last_Elem_Chk =>
54         Local_Init (AA);
55         Put_Line (Integer'Image (Last_Elem (AA)));
56     when Length_Chk =>
57         Put_Line (Integer'Image (Length (AA)));
58     end case;
59 end Check;
60
61 begin
62     if Argument_Count < 1 then
63         Put_Line ("ERROR: missing arguments! Exiting...");
64         return;
65     elsif Argument_Count > 1 then
66         Put_Line ("Ignoring additional arguments...");
67     end if;
68
69     Check (Test_Case_Index'Value (Argument (1)));
70 end Main;

```

18.6.2 Colors: Lookup-Table

Listing 261: color_types.ads

```

1 package Color_Types is
2
3     type HTML_Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     subtype Int_Color is Integer range 0 .. 255;
17
18     type RGB is record
19         Red   : Int_Color;
20         Green : Int_Color;
21         Blue  : Int_Color;
22     end record;

```

(continues on next page)

(continued from previous page)

```

23
24 function To_RGB (C : HTML_Color) return RGB;
25
26 function Image (C : RGB) return String;
27
28 type HTML_Color_RGB is array (HTML_Color) of RGB;
29
30 To_RGB_Lookup_Table : constant HTML_Color_RGB
31 := (Salmon      => (16#FA#, 16#80#, 16#72#),
32      Firebrick  => (16#B2#, 16#22#, 16#22#),
33      Red        => (16#FF#, 16#00#, 16#00#),
34      Darkred    => (16#8B#, 16#00#, 16#00#),
35      Lime       => (16#00#, 16#FF#, 16#00#),
36      Forestgreen => (16#22#, 16#8B#, 16#22#),
37      Green      => (16#00#, 16#80#, 16#00#),
38      Darkgreen  => (16#00#, 16#64#, 16#00#),
39      Blue       => (16#00#, 16#00#, 16#FF#),
40      Mediumblue => (16#00#, 16#00#, 16#CD#),
41      Darkblue   => (16#00#, 16#00#, 16#8B#));
42
43 end Color_Types;

```

Listing 262: color_types.adb

```

1 with Ada.Integer_Text_IO;
2 package body Color_Types is
3
4     function To_RGB (C : HTML_Color) return RGB is
5     begin
6         return To_RGB_Lookup_Table (C);
7     end To_RGB;
8
9     function Image (C : RGB) return String is
10     subtype Str_Range is Integer range 1 .. 10;
11     SR : String (Str_Range);
12     SG : String (Str_Range);
13     SB : String (Str_Range);
14     begin
15         Ada.Integer_Text_IO.Put (To   => SR,
16                                Item  => C.Red,
17                                Base  => 16);
18         Ada.Integer_Text_IO.Put (To   => SG,
19                                Item  => C.Green,
20                                Base  => 16);
21         Ada.Integer_Text_IO.Put (To   => SB,
22                                Item  => C.Blue,
23                                Base  => 16);
24
25         return ("(Red => " & SR
26                & ", Green => " & SG
27                & ", Blue => " & SB
28                & ")");
29     end Image;
30 end Color_Types;

```

Listing 263: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3

```

(continues on next page)

(continued from previous page)

```

4  with Color_Types;           use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Color_Table_Chk,
9           HTML_Color_To_Integer_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Color_Table_Chk =>
15                 Put_Line ("Size of HTML_Color_RGB: "
16                     & Integer'Image (HTML_Color_RGB'Length));
17                 Put_Line ("Firebrick: "
18                     & Image (To_RGB_Lookup_Table (Firebrick)));
19             when HTML_Color_To_Integer_Chk =>
20                 for I in HTML_Color'Range loop
21                     Put_Line (HTML_Color'Image (I) & " => "
22                         & Image (To_RGB (I)) & ".");
23                 end loop;
24             end case;
25     end Check;
26
27 begin
28     if Argument_Count < 1 then
29         Put_Line ("ERROR: missing arguments! Exiting...");
30         return;
31     elsif Argument_Count > 1 then
32         Put_Line ("Ignoring additional arguments...");
33     end if;
34
35     Check (Test_Case_Index'Value (Argument (1)));
36 end Main;

```

18.6.3 Unconstrained Array

Listing 264: unconstrained_arrays.ads

```

1  package Unconstrained_Arrays is
2
3      type My_Array is array (Positive range <>) of Integer;
4
5      procedure Init (A : in out My_Array);
6
7      function Init (I, L : Positive) return My_Array;
8
9      procedure Double (A : in out My_Array);
10
11     function Diff_Prev_Elem (A : My_Array) return My_Array;
12
13 end Unconstrained_Arrays;

```

Listing 265: unconstrained_arrays.adb

```

1  package body Unconstrained_Arrays is
2
3      procedure Init (A : in out My_Array) is
4          Y : Natural := A'Last;
5      begin

```

(continues on next page)

(continued from previous page)

```

6     for I in A'Range loop
7         A (I) := Y;
8         Y := Y - 1;
9     end loop;
10    end Init;
11
12    function Init (I, L : Positive) return My_Array is
13        A : My_Array (I .. I + L - 1);
14    begin
15        Init (A);
16        return A;
17    end Init;
18
19    procedure Double (A : in out My_Array) is
20    begin
21        for I in A'Range loop
22            A (I) := A (I) * 2;
23        end loop;
24    end Double;
25
26    function Diff_Prev_Elem (A : My_Array) return My_Array is
27        A_Out : My_Array (A'Range);
28    begin
29        A_Out (A'First) := 0;
30        for I in A'First + 1 .. A'Last loop
31            A_Out (I) := A (I) - A (I - 1);
32        end loop;
33
34        return A_Out;
35    end Diff_Prev_Elem;
36
37    end Unconstrained_Arrays;

```

Listing 266: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Unconstrained_Arrays; use Unconstrained_Arrays;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Init_Chk,
9           Init_Proc_Chk,
10          Double_Chk,
11          Diff_Prev_Chk,
12          Diff_Prev_Single_Chk);
13
14      procedure Check (TC : Test_Case_Index) is
15          AA : My_Array (1 .. 5);
16          AB : My_Array (5 .. 9);
17
18          procedure Display (A : My_Array) is
19          begin
20              for I in A'Range loop
21                  Put_Line (Integer'Image (A (I)));
22              end loop;
23          end Display;
24
25          procedure Local_Init (A : in out My_Array) is
26          begin

```

(continues on next page)

(continued from previous page)

```

27     A := (1, 2, 5, 10, -10);
28     end Local_Init;
29
30     begin
31         case TC is
32             when Init_Chk =>
33                 AA := Init (AA'First, AA'Length);
34                 AB := Init (AB'First, AB'Length);
35                 Display (AA);
36                 Display (AB);
37             when Init_Proc_Chk =>
38                 Init (AA);
39                 Init (AB);
40                 Display (AA);
41                 Display (AB);
42             when Double_Chk =>
43                 Local_Init (AB);
44                 Double (AB);
45                 Display (AB);
46             when Diff_Prev_Chk =>
47                 Local_Init (AB);
48                 AB := Diff_Prev_Elem (AB);
49                 Display (AB);
50             when Diff_Prev_Single_Chk =>
51                 declare
52                     A1 : My_Array (1 .. 1) := (1 => 42);
53                 begin
54                     A1 := Diff_Prev_Elem (A1);
55                     Display (A1);
56                 end;
57             end case;
58         end Check;
59
60     begin
61         if Argument_Count < 1 then
62             Put_Line ("ERROR: missing arguments! Exiting...");
63             return;
64         elsif Argument_Count > 1 then
65             Put_Line ("Ignoring additional arguments...");
66         end if;
67
68         Check (Test_Case_Index'Value (Argument (1)));
69     end Main;

```

18.6.4 Product info

Listing 267: product_info_pkg.ads

```

1  package Product_Info_Pkg is
2
3      subtype Quantity is Natural;
4
5      subtype Currency is Float;
6
7      type Product_Info is record
8          Units : Quantity;
9          Price : Currency;
10     end record;
11
12     type Product_INFos is array (Positive range <>) of Product_Info;

```

(continues on next page)

(continued from previous page)

```

13
14  type Currency_Array is array (Positive range <>) of Currency;
15
16  procedure Total (P : Product_Infos;
17                  Tot : out Currency_Array);
18
19  function Total (P : Product_Infos) return Currency_Array;
20
21  function Total (P : Product_Infos) return Currency;
22
23  end Product_Info_Pkg;

```

Listing 268: product_info_pkg.adb

```

1  package body Product_Info_Pkg is
2
3      -- Get total for single product
4      function Total (P : Product_Info) return Currency is
5          (Currency (P.Units) * P.Price);
6
7      procedure Total (P : Product_Infos;
8                      Tot : out Currency_Array) is
9      begin
10         for I in P'Range loop
11             Tot (I) := Total (P (I));
12         end loop;
13     end Total;
14
15     function Total (P : Product_Infos) return Currency_Array
16     is
17         Tot : Currency_Array (P'Range);
18     begin
19         Total (P, Tot);
20         return Tot;
21     end Total;
22
23     function Total (P : Product_Infos) return Currency
24     is
25         Tot : Currency := 0.0;
26     begin
27         for I in P'Range loop
28             Tot := Tot + Total (P (I));
29         end loop;
30         return Tot;
31     end Total;
32
33  end Product_Info_Pkg;

```

Listing 269: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Product_Info_Pkg;   use Product_Info_Pkg;
5
6  procedure Main is
7      package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
8
9      type Test_Case_Index is
10         (Total_Func_Chk,

```

(continues on next page)

(continued from previous page)

```

11     Total_Proc_Chk,
12     Total_Value_Chk);
13
14 procedure Check (TC : Test_Case_Index) is
15     subtype Test_Range is Positive range 1 .. 5;
16
17     P    : Product_Infos (Test_Range);
18     Tots : Currency_Array (Test_Range);
19     Tot  : Currency;
20
21     procedure Display (Tots : Currency_Array) is
22     begin
23         for I in Tots'Range loop
24             Currency_IO.Put (Tots (I));
25             New_Line;
26         end loop;
27     end Display;
28
29     procedure Local_Init (P : in out Product_Infos) is
30     begin
31         P := ((1, 0.5),
32              (2, 10.0),
33              (5, 40.0),
34              (10, 10.0),
35              (10, 20.0));
36     end Local_Init;
37
38 begin
39     Currency_IO.Default_Fore := 1;
40     Currency_IO.Default_Aft  := 2;
41     Currency_IO.Default_Exp  := 0;
42
43     case TC is
44     when Total_Func_Chk =>
45         Local_Init (P);
46         Tots := Total (P);
47         Display (Tots);
48     when Total_Proc_Chk =>
49         Local_Init (P);
50         Total (P, Tots);
51         Display (Tots);
52     when Total_Value_Chk =>
53         Local_Init (P);
54         Tot := Total (P);
55         Currency_IO.Put (Tot);
56         New_Line;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

18.6.5 String_10

Listing 270: strings_10.ads

```

1 package Strings_10 is
2
3     subtype String_10 is String (1 .. 10);
4
5     -- Using "type String_10 is..." is possible, too.
6
7     function To_String_10 (S : String) return String_10;
8
9 end Strings_10;
```

Listing 271: strings_10.adb

```

1 package body Strings_10 is
2
3     function To_String_10 (S : String) return String_10 is
4         S_Out : String_10;
5     begin
6         for I in String_10'First .. Integer'Min (String_10'Last, S'Last) loop
7             S_Out (I) := S (I);
8         end loop;
9
10        for I in Integer'Min (String_10'Last + 1, S'Last + 1) .. String_10'Last loop
11            S_Out (I) := ' ';
12        end loop;
13
14        return S_Out;
15    end To_String_10;
16
17 end Strings_10;
```

Listing 272: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3
4 with Strings_10;          use Strings_10;
5
6 procedure Main is
7     type Test_Case_Index is
8         (String_10_Long_Chk,
9          String_10_Short_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         SL : constant String := "And this is a long string just for testing...";
13         SS : constant String := "Hey!";
14         S_10 : String_10;
15
16     begin
17         case TC is
18             when String_10_Long_Chk =>
19                 S_10 := To_String_10 (SL);
20                 Put_Line (String (S_10));
21             when String_10_Short_Chk =>
22                 S_10 := (others => ' ');
23                 S_10 := To_String_10 (SS);
24                 Put_Line (String (S_10));
25             end case;
26     end Check;
```

(continues on next page)

(continued from previous page)

```
27
28 begin
29   if Argument_Count < 1 then
30     Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
31     return;
32   elsif Argument_Count > 1 then
33     Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
34   end if;
35
36   Check (Test_Case_Index'Value (Argument (1)));
37 end Main;
```

18.6.6 List of Names

Listing 273: names_ages.ads

```
1 package Names_Ages is
2
3   Max_People : constant Positive := 10;
4
5   subtype Name_Type is String (1 .. 50);
6
7   type Age_Type is new Natural;
8
9   type Person is record
10     Name : Name_Type;
11     Age  : Age_Type;
12   end record;
13
14   type People_Array is array (Positive range <>) of Person;
15
16   type People is record
17     People_A : People_Array (1 .. Max_People);
18     Last_Valid : Natural;
19   end record;
20
21   procedure Reset (P : in out People);
22
23   procedure Add (P : in out People;
24                 Name : String);
25
26   function Get (P : People;
27                Name : String) return Age_Type;
28
29   procedure Update (P : in out People;
30                    Name : String;
31                    Age : Age_Type);
32
33   procedure Display (P : People);
34
35 end Names_Ages;
```

Listing 274: names_ages.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Ada.Strings;      use Ada.Strings;
3 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
4
5 package body Names_Ages is
```

(continues on next page)

(continued from previous page)

```

6
7  function To_Name_Type (S : String) return Name_Type is
8      S_Out : Name_Type := (others => ' ');
9  begin
10     for I in 1 .. Integer'Min (S'Last, Name_Type'Last) loop
11         S_Out (I) := S (I);
12     end loop;
13
14     return S_Out;
15 end To_Name_Type;
16
17 procedure Init (P      : in out Person;
18                Name :      String) is
19 begin
20     P.Name := To_Name_Type (Name);
21     P.Age := 0;
22 end Init;
23
24 function Match (P      : Person;
25                Name : String) return Boolean is
26 begin
27     return P.Name = To_Name_Type (Name);
28 end Match;
29
30 function Get (P : Person) return Age_Type is
31 begin
32     return P.Age;
33 end Get;
34
35 procedure Update (P      : in out Person;
36                  Age :      Age_Type) is
37 begin
38     P.Age := Age;
39 end Update;
40
41 procedure Display (P : Person) is
42 begin
43     Put_Line ("NAME: " & Trim (P.Name, Right));
44     Put_Line ("AGE: " & Age_Type'Image (P.Age));
45 end Display;
46
47 procedure Reset (P : in out People) is
48 begin
49     P.Last_Valid := 0;
50 end Reset;
51
52 procedure Add (P      : in out People;
53               Name :      String) is
54 begin
55     P.Last_Valid := P.Last_Valid + 1;
56     Init (P.People_A (P.Last_Valid), Name);
57 end Add;
58
59 function Get (P      : People;
60               Name : String) return Age_Type is
61 begin
62     for I in P.People_A'First .. P.Last_Valid loop
63         if Match (P.People_A (I), Name) then
64             return Get (P.People_A (I));
65         end if;
66     end loop;

```

(continues on next page)

(continued from previous page)

```

67
68     return 0;
69 end Get;
70
71 procedure Update (P      : in out People;
72                  Name   :      String;
73                  Age    :      Age_Type) is
74 begin
75     for I in P.People_A'First .. P.Last_Valid loop
76         if Match (P.People_A (I), Name) then
77             Update (P.People_A (I), Age);
78         end if;
79     end loop;
80 end Update;
81
82 procedure Display (P : People) is
83 begin
84     Put_Line ("LIST OF NAMES:");
85     for I in P.People_A'First .. P.Last_Valid loop
86         Display (P.People_A (I));
87     end loop;
88 end Display;
89
90 end Names_Ages;

```

Listing 275: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Names_Ages;         use Names_Ages;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Names_Ages_Chk,
9           Get_Age_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         P : People;
13     begin
14         case TC is
15             when Names_Ages_Chk =>
16                 Reset (P);
17                 Add (P, "John");
18                 Add (P, "Patricia");
19                 Add (P, "Josh");
20                 Display (P);
21                 Update (P, "John", 18);
22                 Update (P, "Patricia", 35);
23                 Update (P, "Josh", 53);
24                 Display (P);
25             when Get_Age_Chk =>
26                 Reset (P);
27                 Add (P, "Peter");
28                 Update (P, "Peter", 45);
29                 Put_Line ("Peter is "
30                           & Age_Type'Image (Get (P, "Peter"))
31                           & " years old.");
32         end case;
33     end Check;
34

```

(continues on next page)

(continued from previous page)

```

35 begin
36   if Argument_Count < 1 then
37     Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
38     return;
39   elsif Argument_Count > 1 then
40     Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
41   end if;
42
43   Check (Test_Case_Index'Value (Argument (1)));
44 end Main;

```

18.7 More About Types

18.7.1 Aggregate Initialization

Listing 276: aggregates.ads

```

1  package Aggregates is
2
3     type Rec is record
4       W : Integer := 10;
5       X : Integer := 11;
6       Y : Integer := 12;
7       Z : Integer := 13;
8     end record;
9
10    type Int_Arr is array (1 .. 20) of Integer;
11
12    procedure Init (R : out Rec);
13
14    procedure Init_Some (A : out Int_Arr);
15
16    procedure Init (A : out Int_Arr);
17
18 end Aggregates;

```

Listing 277: aggregates.adb

```

1  package body Aggregates is
2
3     procedure Init (R : out Rec) is
4     begin
5       R := (X      => 100,
6            Y      => 200,
7            others => <>);
8     end Init;
9
10    procedure Init_Some (A : out Int_Arr) is
11    begin
12      A := (1 .. 5 => 99,
13           others => 100);
14    end Init_Some;
15
16    procedure Init (A : out Int_Arr) is
17    begin
18      A := (others => 5);
19    end Init;
20
21 end Aggregates;

```

Listing 278: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Aggregates;       use Aggregates;
5
6  procedure Main is
7      -- Remark: the following line is not relevant.
8      F : array (1 .. 10) of Float := (others => 42.42)
9      with Unreferenced;
10
11     type Test_Case_Index is
12         (Default_Rec_Chk,
13          Init_Rec_Chk,
14          Init_Some_Arr_Chk,
15          Init_Arr_Chk);
16
17     procedure Check (TC : Test_Case_Index) is
18         A : Int_Arr;
19         R : Rec;
20         DR : constant Rec := (others => <>);
21     begin
22         case TC is
23             when Default_Rec_Chk =>
24                 R := DR;
25                 Put_Line ("Record Default:");
26                 Put_Line ("W => " & Integer'Image (R.W));
27                 Put_Line ("X => " & Integer'Image (R.X));
28                 Put_Line ("Y => " & Integer'Image (R.Y));
29                 Put_Line ("Z => " & Integer'Image (R.Z));
30             when Init_Rec_Chk =>
31                 Init (R);
32                 Put_Line ("Record Init:");
33                 Put_Line ("W => " & Integer'Image (R.W));
34                 Put_Line ("X => " & Integer'Image (R.X));
35                 Put_Line ("Y => " & Integer'Image (R.Y));
36                 Put_Line ("Z => " & Integer'Image (R.Z));
37             when Init_Some_Arr_Chk =>
38                 Init_Some (A);
39                 Put_Line ("Array Init_Some:");
40                 for I in A'Range loop
41                     Put_Line (Integer'Image (I) & " "
42                               & Integer'Image (A (I)));
43                 end loop;
44             when Init_Arr_Chk =>
45                 Init (A);
46                 Put_Line ("Array Init:");
47                 for I in A'Range loop
48                     Put_Line (Integer'Image (I) & " "
49                               & Integer'Image (A (I)));
50                 end loop;
51         end case;
52     end Check;
53
54     begin
55         if Argument_Count < 1 then
56             Put_Line ("ERROR: missing arguments! Exiting...");
57             return;
58         elsif Argument_Count > 1 then
59             Put_Line ("Ignoring additional arguments...");

```

(continues on next page)

(continued from previous page)

```

60     end if;
61
62     Check (Test_Case_Index'Value (Argument (1)));
63 end Main;

```

18.7.2 Versioning

Listing 279: versioning.ads

```

1  package Versioning is
2
3      type Version is record
4          Major      : Natural;
5          Minor      : Natural;
6          Maintenance : Natural;
7      end record;
8
9      function Convert (V : Version) return String;
10
11     function Convert (V : Version) return Float;
12
13 end Versioning;

```

Listing 280: versioning.adb

```

1  with Ada.Strings; use Ada.Strings;
2  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3
4  package body Versioning is
5
6      function Image_Trim (N : Natural) return String is
7          S_N : constant String := Trim (Natural'Image (N), Left);
8      begin
9          return S_N;
10     end Image_Trim;
11
12     function Convert (V : Version) return String is
13         S_Major : constant String := Image_Trim (V.Major);
14         S_Minor : constant String := Image_Trim (V.Minor);
15         S_Maint : constant String := Image_Trim (V.Maintenance);
16     begin
17         return (S_Major & "." & S_Minor & "." & S_Maint);
18     end Convert;
19
20     function Convert (V : Version) return Float is
21     begin
22         return Float (V.Major) + (Float (V.Minor) / 10.0);
23     end Convert;
24
25 end Versioning;

```

Listing 281: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Versioning;       use Versioning;
5
6  procedure Main is

```

(continues on next page)

(continued from previous page)

```

7  type Test_Case_Index is
8      (Ver_String_Chk,
9       Ver_Float_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12      V : constant Version := (1, 3, 23);
13  begin
14      case TC is
15          when Ver_String_Chk =>
16              Put_Line (Convert (V));
17          when Ver_Float_Chk =>
18              Put_Line (Float'Image (Convert (V)));
19      end case;
20  end Check;
21
22  begin
23      if Argument_Count < 1 then
24          Put_Line ("ERROR: missing arguments! Exiting...");
25          return;
26      elsif Argument_Count > 1 then
27          Put_Line ("Ignoring additional arguments...");
28      end if;
29
30      Check (Test_Case_Index'Value (Argument (1)));
31  end Main;

```

18.7.3 Simple todo list

Listing 282: todo_lists.ads

```

1  package Todo_Lists is
2
3      type Todo_Item is access String;
4
5      type Todo_Items is array (Positive range <>) of Todo_Item;
6
7      type Todo_List (Max_Len : Natural) is record
8          Items : Todo_Items (1 .. Max_Len);
9          Last  : Natural := 0;
10     end record;
11
12     procedure Add (Todos : in out Todo_List;
13                  Item  : String);
14
15     procedure Display (Todos : Todo_List);
16
17 end Todo_Lists;

```

Listing 283: todo_lists.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Todo_Lists is
4
5      procedure Add (Todos : in out Todo_List;
6                   Item  : String) is
7      begin
8          if Todos.Last < Todos.Items'Last then
9              Todos.Last := Todos.Last + 1;

```

(continues on next page)

(continued from previous page)

```

10     Todos.Items (Todos.Last) := new String'(Item);
11 else
12     Put_Line ("ERROR: list is full!");
13 end if;
14 end Add;
15
16 procedure Display (Todos : Todo_List) is
17 begin
18     Put_Line ("TO-DO LIST");
19     for I in Todos.Items'First .. Todos.Last loop
20         Put_Line (Todos.Items (I).all);
21     end loop;
22 end Display;
23
24 end Todo_Lists;

```

Listing 284: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Todo_Lists;       use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List (10);
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreadsheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;

```


18.7.4 Price list

Listing 285: price_lists.ads

```

1 package Price_Lists is
2
3     type Price_Type is delta 0.01 digits 12;
4
5     type Price_List_Array is array (Positive range <>) of Price_Type;
6
7     type Price_List (Max : Positive) is record
8         List : Price_List_Array (1 .. Max);
9         Last : Natural := 0;
10    end record;
11
12    type Price_Result (Ok : Boolean) is record
13        case Ok is
14            when False =>
15                null;
16            when True =>
17                Price : Price_Type;
18        end case;
19    end record;
20
21    procedure Reset (Prices : in out Price_List);
22
23    procedure Add (Prices : in out Price_List;
24                  Item   : Price_Type);
25
26    function Get (Prices : Price_List;
27                 Idx     : Positive) return Price_Result;
28
29    procedure Display (Prices : Price_List);
30
31 end Price_Lists;

```

Listing 286: price_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Price_Lists is
4
5     procedure Reset (Prices : in out Price_List) is
6     begin
7         Prices.Last := 0;
8     end Reset;
9
10    procedure Add (Prices : in out Price_List;
11                  Item   : Price_Type) is
12    begin
13        if Prices.Last < Prices.List'Last then
14            Prices.Last := Prices.Last + 1;
15            Prices.List (Prices.Last) := Item;
16        else
17            Put_Line ("ERROR: list is full!");
18        end if;
19    end Add;
20
21    function Get (Prices : Price_List;
22                 Idx     : Positive) return Price_Result is
23    begin
24        if (Idx >= Prices.List'First and then

```

(continues on next page)

(continued from previous page)

```

25         Idx <= Prices.Last)           then
26         return Price_Result'(Ok      => True,
27                               Price => Prices.List (Idx));
28     else
29         return Price_Result'(Ok      => False);
30     end if;
31 end Get;
32
33 procedure Display (Prices : Price_List) is
34 begin
35     Put_Line ("PRICE LIST");
36     for I in Prices.List'First .. Prices.Last loop
37         Put_Line (Price_Type'Image (Prices.List (I)));
38     end loop;
39 end Display;
40
41 end Price_Lists;

```

Listing 287: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Price_Lists;      use Price_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Price_Type_Chk,
9           Price_List_Chk,
10          Price_List_Get_Chk);
11
12      procedure Check (TC : Test_Case_Index) is
13          L : Price_List (10);
14
15          procedure Local_Init_List is
16          begin
17              Reset (L);
18              Add (L, 1.45);
19              Add (L, 2.37);
20              Add (L, 3.21);
21              Add (L, 4.14);
22              Add (L, 5.22);
23              Add (L, 6.69);
24              Add (L, 7.77);
25              Add (L, 8.14);
26              Add (L, 9.99);
27              Add (L, 10.01);
28          end Local_Init_List;
29
30      procedure Get_Display (Idx : Positive) is
31          R : constant Price_Result := Get (L, Idx);
32      begin
33          Put_Line ("Attempt Get # " & Positive'Image (Idx));
34          if R.Ok then
35              Put_Line ("Element # " & Positive'Image (Idx)
36                      & " => " & Price_Type'Image (R.Price));
37          else
38              declare
39              begin
40                  Put_Line ("Element # " & Positive'Image (Idx)
41                          & " => " & Price_Type'Image (R.Price));

```

(continues on next page)

(continued from previous page)

```

42         exception
43         when others =>
44             Put_Line ("Element not available (as expected)");
45         end;
46     end if;
47
48     end Get_Display;
49
50     begin
51         case TC is
52             when Price_Type_Chk =>
53                 Put_Line ("The delta value of Price_Type is "
54                     & Price_Type'Image (Price_Type'Delta) & ";");
55                 Put_Line ("The minimum value of Price_Type is "
56                     & Price_Type'Image (Price_Type'First) & ";");
57                 Put_Line ("The maximum value of Price_Type is "
58                     & Price_Type'Image (Price_Type'Last) & ";");
59             when Price_List_Chk =>
60                 Local_Init_List;
61                 Display (L);
62             when Price_List_Get_Chk =>
63                 Local_Init_List;
64                 Get_Display (5);
65                 Get_Display (40);
66         end case;
67     end Check;
68
69     begin
70         if Argument_Count < 1 then
71             Put_Line ("ERROR: missing arguments! Exiting...");
72             return;
73         elsif Argument_Count > 1 then
74             Put_Line ("Ignoring additional arguments...");
75         end if;
76
77         Check (Test_Case_Index'Value (Argument (1)));
78     end Main;

```

18.8 Privacy

18.8.1 Directions

Listing 288: directions.ads

```

1 package Directions is
2
3     type Angle_Mod is mod 360;
4
5     type Direction is
6         (North,
7          Northwest,
8          West,
9          Southwest,
10         South,
11         Southeast,
12         East);
13
14     function To_Direction (N : Angle_Mod) return Direction;
15

```

(continues on next page)

(continued from previous page)

```

16  type Ext_Angle is private;
17
18  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
19
20  procedure Display (N : Ext_Angle);
21
22 private
23
24  type Ext_Angle is record
25      Angle_Elem      : Angle_Mod;
26      Direction_Elem : Direction;
27  end record;
28
29  end Directions;

```

Listing 289: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          Put_Line ("Angle: "
8                  & Angle_Mod'Image (N.Angle_Elem)
9                  & " => "
10                 & Direction'Image (N.Direction_Elem)
11                 & ".");
12      end Display;
13
14      function To_Direction (N : Angle_Mod) return Direction is
15      begin
16          case N is
17              when 0      => return East;
18              when 1 .. 89 => return Northwest;
19              when 90     => return North;
20              when 91 .. 179 => return Northwest;
21              when 180    => return West;
22              when 181 .. 269 => return Southwest;
23              when 270    => return South;
24              when 271 .. 359 => return Southeast;
25          end case;
26      end To_Direction;
27
28      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29      begin
30          return (Angle_Elem => N,
31                 Direction_Elem => To_Direction (N));
32      end To_Ext_Angle;
33
34  end Directions;

```

Listing 290: test_directions.adb

```

1  with Directions; use Directions;
2
3  procedure Test_Directions is
4      type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;
5
6      All_Directions : constant Ext_Angle_Array (1 .. 6)

```

(continues on next page)

(continued from previous page)

```

7      := (To_Ext_Angle (0),
8          To_Ext_Angle (45),
9          To_Ext_Angle (90),
10         To_Ext_Angle (91),
11         To_Ext_Angle (180),
12         To_Ext_Angle (270));
13
14 begin
15     for I in All_Directions'Range loop
16         Display (All_Directions (I));
17     end loop;
18
19 end Test_Directions;

```

Listing 291: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when Direction_Chk =>
14                 Test_Directions;
15         end case;
16     end Check;
17
18 begin
19     if Argument_Count < 1 then
20         Put_Line ("ERROR: missing arguments! Exiting...");
21         return;
22     elsif Argument_Count > 1 then
23         Put_Line ("Ignoring additional arguments...");
24     end if;
25
26     Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

18.8.2 Limited Strings

Listing 292: limited_strings.ads

```

1  package Limited_Strings is
2
3      type Lim_String is limited private;
4
5      function Init (S : String) return Lim_String;
6
7      function Init (Max : Positive) return Lim_String;
8
9      procedure Put_Line (LS : Lim_String);
10
11     procedure Copy (From : Lim_String;

```

(continues on next page)

(continued from previous page)

```

12         To : in out Lim_String));
13
14     function "=" (Ref, Dut : Lim_String) return Boolean;
15
16 private
17
18     type Lim_String is access String;
19
20 end Limited_Strings;

```

Listing 293: limited_strings.adb

```

1  with Ada.Text_IO;
2
3  package body Limited_Strings
4  is
5
6      function Init (S : String) return Lim_String is
7          LS : constant Lim_String := new String'(S);
8      begin
9          return Ls;
10     end Init;
11
12     function Init (Max : Positive) return Lim_String is
13         LS : constant Lim_String := new String (1 .. Max);
14     begin
15         LS.all := (others => '_');
16         return LS;
17     end Init;
18
19     procedure Put_Line (LS : Lim_String) is
20     begin
21         Ada.Text_IO.Put_Line (LS.all);
22     end Put_Line;
23
24     function Get_Min_Last (A, B : Lim_String) return Positive is
25     begin
26         return Positive'Min (A'Last, B'Last);
27     end Get_Min_Last;
28
29     procedure Copy (From : Lim_String;
30                    To : in out Lim_String) is
31         Min_Last : constant Positive := Get_Min_Last (From, To);
32     begin
33         To (To'First .. Min_Last) := From (To'First .. Min_Last);
34         To (Min_Last + 1 .. To'Last) := (others => '_');
35     end;
36
37     function "=" (Ref, Dut : Lim_String) return Boolean is
38         Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
39     begin
40         for I in Dut'First .. Min_Last loop
41             if Dut (I) /= Ref (I) then
42                 return False;
43             end if;
44         end loop;
45
46         return True;
47     end;
48
49 end Limited_Strings;

```

Listing 294: check_lim_string.adb

```
1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Limited_Strings; use Limited_Strings;
4
5  procedure Check_Lim_String is
6      S : constant String := "-----";
7      S1 : constant Lim_String := Init ("Hello World");
8      S2 : constant Lim_String := Init (30);
9      S3 : Lim_String := Init (5);
10     S4 : Lim_String := Init (S & S & S);
11 begin
12     Put ("S1 => ");
13     Put_Line (S1);
14     Put ("S2 => ");
15     Put_Line (S2);
16
17     if S1 = S2 then
18         Put_Line ("S1 is equal to S2.");
19     else
20         Put_Line ("S1 isn't equal to S2.");
21     end if;
22
23     Copy (From => S1, To => S3);
24     Put ("S3 => ");
25     Put_Line (S3);
26
27     if S1 = S3 then
28         Put_Line ("S1 is equal to S3.");
29     else
30         Put_Line ("S1 isn't equal to S3.");
31     end if;
32
33     Copy (From => S1, To => S4);
34     Put ("S4 => ");
35     Put_Line (S4);
36
37     if S1 = S4 then
38         Put_Line ("S1 is equal to S4.");
39     else
40         Put_Line ("S1 isn't equal to S4.");
41     end if;
42 end Check_Lim_String;
```

Listing 295: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Check_Lim_String;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Lim_String_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13         when Lim_String_Chk =>
14             Check_Lim_String;
```

(continues on next page)

(continued from previous page)

```

15     end case;
16 end Check;
17
18 begin
19     if Argument_Count < 1 then
20         Put_Line ("ERROR: missing arguments! Exiting...");
21         return;
22     elsif Argument_Count > 1 then
23         Put_Line ("Ignoring additional arguments...");
24     end if;
25
26     Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

18.9 Generics

18.9.1 Display Array

Listing 296: display_array.ads

```

1 generic
2     type T_Range is range <>;
3     type T_Element is private;
4     type T_Array is array (T_Range range <>) of T_Element;
5     with function Image (E : T_Element) return String;
6     procedure Display_Array (Header : String;
7                             A       : T_Array);

```

Listing 297: display_array.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Array (Header : String;
4                         A       : T_Array) is
5 begin
6     Put_Line (Header);
7     for I in A'Range loop
8         Put_Line (T_Range'Image (I) & ": " & Image (A (I)));
9     end loop;
10 end Display_Array;

```

Listing 298: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Array;
5
6 procedure Main is
7     type Test_Case_Index is (Int_Array_Chk,
8                             Point_Array_Chk);
9
10    procedure Test_Int_Array is
11        type Int_Array is array (Positive range <>) of Integer;
12
13        procedure Display_Int_Array is new
14            Display_Array (T_Range => Positive,
15                          T_Element => Integer,

```

(continues on next page)

(continued from previous page)

```

16         T_Array    => Int_Array,
17         Image      => Integer'Image);
18
19     A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
20 begin
21     Display_Int_Array ("Integers", A);
22 end Test_Int_Array;
23
24 procedure Test_Point_Array is
25     type Point is record
26         X : Float;
27         Y : Float;
28     end record;
29
30     type Point_Array is array (Natural range <>) of Point;
31
32     function Image (P : Point) return String is
33     begin
34         return "(" & Float'Image (P.X)
35             & ", " & Float'Image (P.Y) & ")";
36     end Image;
37
38     procedure Display_Point_Array is new
39         Display_Array (T_Range    => Natural,
40                       T_Element  => Point,
41                       T_Array    => Point_Array,
42                       Image      => Image);
43
44     A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
45                                           (5.0, 2.0), (-0.5, 2.0));
46 begin
47     Display_Point_Array ("Points", A);
48 end Test_Point_Array;
49
50 procedure Check (TC : Test_Case_Index) is
51 begin
52     case TC is
53         when Int_Array_Chk =>
54             Test_Int_Array;
55         when Point_Array_Chk =>
56             Test_Point_Array;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

18.9.2 Average of Array of Float

Listing 299: average.ads

```

1 generic
2   type T_Range is range <>;
3   type T_Element is digits <>;
4   type T_Array is array (T_Range range <>) of T_Element;
5   function Average (A : T_Array) return T_Element;

```

Listing 300: average.adb

```

1 function Average (A : T_Array) return T_Element is
2   Acc : Float := 0.0;
3 begin
4   for I in A'Range loop
5     Acc := Acc + Float (A (I));
6   end loop;
7
8   return T_Element (Acc / Float (A'Length));
9 end Average;

```

Listing 301: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Average;
5
6 procedure Main is
7   type Test_Case_Index is (Float_Array_Chk,
8                             Digits_7_Float_Array_Chk);
9
10  procedure Test_Float_Array is
11    type Float_Array is array (Positive range <>) of Float;
12
13    function Average_Float is new
14      Average (T_Range => Positive,
15               T_Element => Float,
16               T_Array => Float_Array);
17
18    A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
19  begin
20    Put_Line ("Average: " & Float'Image (Average_Float (A)));
21  end Test_Float_Array;
22
23  procedure Test_Digits_7_Float_Array is
24    type Custom_Float is digits 7 range 0.0 .. 1.0;
25
26    type Float_Array is
27      array (Integer range <>) of Custom_Float;
28
29    function Average_Float is new
30      Average (T_Range => Integer,
31               T_Element => Custom_Float,
32               T_Array => Float_Array);
33
34    A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
35  begin
36    Put_Line ("Average: "
37              & Custom_Float'Image (Average_Float (A)));
38  end Test_Digits_7_Float_Array;
39

```

(continues on next page)

(continued from previous page)

```

40  procedure Check (TC : Test_Case_Index) is
41  begin
42      case TC is
43          when Float_Array_Chk =>
44              Test_Float_Array;
45          when Digits_7_Float_Array_Chk =>
46              Test_Digits_7_Float_Array;
47      end case;
48  end Check;
49
50  begin
51      if Argument_Count < 1 then
52          Put_Line ("ERROR: missing arguments! Exiting...");
53          return;
54      elsif Argument_Count > 1 then
55          Put_Line ("Ignoring additional arguments...");
56      end if;
57
58      Check (Test_Case_Index'Value (Argument (1)));
59  end Main;

```

18.9.3 Average of Array of Any Type

Listing 302: average.ads

```

1  generic
2      type T_Range is range <>;
3      type T_Element is private;
4      type T_Array is array (T_Range range <>) of T_Element;
5      with function To_Float (E : T_Element) return Float is <>;
6  function Average (A : T_Array) return Float;

```

Listing 303: average.adb

```

1  function Average (A : T_Array) return Float is
2      Acc : Float := 0.0;
3  begin
4      for I in A'Range loop
5          Acc := Acc + To_Float (A (I));
6      end loop;
7
8      return Acc / Float (A'Length);
9  end Average;

```

Listing 304: test_item.ads

```

1  procedure Test_Item;

```

Listing 305: test_item.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Average;
4
5  procedure Test_Item is
6      package F_IO is new Ada.Text_IO.Float_IO (Float);
7
8      type Amount is delta 0.01 digits 12;
9

```

(continues on next page)

(continued from previous page)

```

10  type Item is record
11      Quantity : Natural;
12      Price    : Amount;
13  end record;
14
15  type Item_Array is
16      array (Positive range <>) of Item;
17
18  function Get_Total (I : Item) return Float is
19      (Float (I.Quantity) * Float (I.Price));
20
21  function Get_Price (I : Item) return Float is
22      (Float (I.Price));
23
24  function Average_Total is new
25      Average (T_Range => Positive,
26              T_Element => Item,
27              T_Array   => Item_Array,
28              To_Float  => Get_Total);
29
30  function Average_Price is new
31      Average (T_Range => Positive,
32              T_Element => Item,
33              T_Array   => Item_Array,
34              To_Float  => Get_Price);
35
36  A : constant Item_Array (1 .. 4)
37      := ((Quantity => 5,   Price => 10.00),
38         (Quantity => 80,  Price => 2.50),
39         (Quantity => 40,  Price => 5.00),
40         (Quantity => 20,  Price => 12.50));
41
42  begin
43      Put ("Average per item & quantity: ");
44      F_IO.Put (Average_Total (A), 3, 2, 0);
45      New_Line;
46
47      Put ("Average price: ");
48      F_IO.Put (Average_Price (A), 3, 2, 0);
49      New_Line;
50  end Test_Item;

```

Listing 306: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Item;
5
6  procedure Main is
7      type Test_Case_Index is (Item_Array_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10         begin
11             case TC is
12                 when Item_Array_Chk =>
13                     Test_Item;
14             end case;
15         end Check;
16
17  begin

```

(continues on next page)

(continued from previous page)

```
18   if Argument_Count < 1 then
19       Put_Line ("ERROR: missing arguments! Exiting...");
20       return;
21   elsif Argument_Count > 1 then
22       Put_Line ("Ignoring additional arguments...");
23   end if;
24
25   Check (Test_Case_Index'Value (Argument (1)));
26 end Main;
```

18.9.4 Generic list

Listing 307: gen_list.ads

```
1  generic
2      type Item is private;
3      type Items is array (Positive range <>) of Item;
4      Name      : String;
5      List_Array : in out Items;
6      Last      : in out Natural;
7      with procedure Put (I : Item) is <>;
8  package Gen_List is
9
10     procedure Init;
11
12     procedure Add (I      : Item;
13                  Status : out Boolean);
14
15     procedure Display;
16
17 end Gen_List;
```

Listing 308: gen_list.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Gen_List is
4
5      procedure Init is
6      begin
7          Last := List_Array'First - 1;
8      end Init;
9
10     procedure Add (I      : Item;
11                  Status : out Boolean) is
12     begin
13         Status := Last < List_Array'Last;
14
15         if Status then
16             Last := Last + 1;
17             List_Array (Last) := I;
18         end if;
19     end Add;
20
21     procedure Display is
22     begin
23         Put_Line (Name);
24         for I in List_Array'First .. Last loop
25             Put (List_Array (I));
```

(continues on next page)

(continued from previous page)

```

26     New_Line;
27     end loop;
28 end Display;
29
30 end Gen_List;

```

Listing 309: test_int.ads

```

1 procedure Test_Int;

```

Listing 310: test_int.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_List;
4
5 procedure Test_Int is
6
7     procedure Put (I : Integer) is
8     begin
9         Ada.Text_IO.Put (Integer'Image (I));
10    end Put;
11
12    type Integer_Array is array (Positive range <>) of Integer;
13
14    A : Integer_Array (1 .. 3);
15    L : Natural;
16
17    package Int_List is new
18        Gen_List (Item      => Integer,
19                  Items     => Integer_Array,
20                  Name      => "List of integers",
21                  List_Array => A,
22                  Last      => L);
23
24    Success : Boolean;
25
26    procedure Display_Add_Success (Success : Boolean) is
27    begin
28        if Success then
29            Put_Line ("Added item successfully!");
30        else
31            Put_Line ("Couldn't add item!");
32        end if;
33
34    end Display_Add_Success;
35
36 begin
37     Int_List.Init;
38
39     Int_List.Add (2, Success);
40     Display_Add_Success (Success);
41
42     Int_List.Add (5, Success);
43     Display_Add_Success (Success);
44
45     Int_List.Add (7, Success);
46     Display_Add_Success (Success);
47
48     Int_List.Add (8, Success);

```

(continues on next page)

(continued from previous page)

```
49   Display_Add_Success (Success);
50
51   Int_List.Display;
52 end Test_Int;
```

Listing 311: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Int;
5
6  procedure Main is
7      type Test_Case_Index is (Int_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10         begin
11             case TC is
12                 when Int_Chk =>
13                     Test_Int;
14             end case;
15         end Check;
16
17     begin
18         if Argument_Count < 1 then
19             Put_Line ("ERROR: missing arguments! Exiting...");
20             return;
21         elsif Argument_Count > 1 then
22             Put_Line ("Ignoring additional arguments...");
23         end if;
24
25         Check (Test_Case_Index'Value (Argument (1)));
26     end Main;
```

18.10 Exceptions

18.10.1 Uninitialized Value

Listing 312: options.ads

```
1  package Options is
2
3      type Option is (Uninitialized,
4                      Option_1,
5                      Option_2,
6                      Option_3);
7
8      Uninitialized_Value : exception;
9
10     function Image (O : Option) return String;
11
12 end Options;
```

Listing 313: options.adb

```
1  package body Options is
2
3      function Image (O : Option) return String is
```

(continues on next page)

(continued from previous page)

```

4   begin
5       case 0 is
6           when Uninitialized =>
7               raise Uninitialized_Value with "Uninitialized value detected!";
8           when others =>
9               return Option'Image (0);
10          end case;
11      end Image;
12
13  end Options;

```

Listing 314: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Exceptions;   use Ada.Exceptions;
4
5  with Options;          use Options;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Options_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Check (O : Option) is
14             begin
15                 Put_Line (Image (O));
16             exception
17                 when E : Uninitialized_Value =>
18                     Put_Line (Exception_Message (E));
19             end Check;
20
21     begin
22         case TC is
23             when Options_Chk =>
24                 for 0 in Option loop
25                     Check (0);
26                 end loop;
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;

```

18.10.2 Numerical Exception

Listing 315: tests.ads

```

1  package Tests is
2

```

(continues on next page)

(continued from previous page)

```

3  type Test_ID is (Test_1, Test_2);
4
5  Custom_Exception : exception;
6
7  procedure Num_Exception_Test (ID : Test_ID);
8
9  end Tests;

```

Listing 316: tests.adb

```

1  package body Tests is
2
3      pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5      procedure Num_Exception_Test (ID : Test_ID) is
6          A, B, C : Integer;
7      begin
8          case ID is
9              when Test_1 =>
10                 A := Integer'Last;
11                 B := Integer'Last;
12                 C := A + B;
13              when Test_2 =>
14                 raise Custom_Exception with "Custom_Exception raised!";
15          end case;
16      end Num_Exception_Test;
17
18      pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20  end Tests;

```

Listing 317: check_exception.adb

```

1  with Tests;          use Tests;
2
3  with Ada.Text_IO;    use Ada.Text_IO;
4  with Ada.Exceptions; use Ada.Exceptions;
5
6  procedure Check_Exception (ID : Test_ID) is
7  begin
8      Num_Exception_Test (ID);
9  exception
10     when Constraint_Error =>
11         Put_Line ("Constraint_Error detected!");
12     when E : others =>
13         Put_Line (Exception_Message (E));
14  end Check_Exception;

```

Listing 318: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Exceptions;   use Ada.Exceptions;
4
5  with Tests;            use Tests;
6  with Check_Exception;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Exception_1_Chk,

```

(continues on next page)

(continued from previous page)

```

11     Exception_2_Chk);
12
13 procedure Check (TC : Test_Case_Index) is
14
15     procedure Check_Handle_Exception (ID : Test_ID) is
16     begin
17         Check_Exception (ID);
18     exception
19         when Constraint_Error =>
20             Put_Line ("Constraint_Error"
21                 & " (raised by Check_Exception) detected!");
22         when E : others =>
23             Put_Line (Exception_Name (E)
24                 & " (raised by Check_Exception) detected!");
25     end Check_Handle_Exception;
26
27 begin
28     case TC is
29     when Exception_1_Chk =>
30         Check_Handle_Exception (Test_1);
31     when Exception_2_Chk =>
32         Check_Handle_Exception (Test_2);
33     end case;
34 end Check;
35
36 begin
37     if Argument_Count < 1 then
38         Put_Line ("ERROR: missing arguments! Exiting...");
39         return;
40     elsif Argument_Count > 1 then
41         Put_Line ("Ignoring additional arguments...");
42     end if;
43
44     Check (Test_Case_Index'Value (Argument (1)));
45 end Main;

```

18.10.3 Re-raising Exceptions

Listing 319: tests.ads

```

1 package Tests is
2
3     type Test_ID is (Test_1, Test_2);
4
5     Custom_Exception, Another_Exception : exception;
6
7     procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;

```

Listing 320: tests.adb

```

1 package body Tests is
2
3     pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5     procedure Num_Exception_Test (ID : Test_ID) is
6         A, B, C : Integer;
7     begin

```

(continues on next page)

(continued from previous page)

```

8      case ID is
9          when Test_1 =>
10             A := Integer'Last;
11             B := Integer'Last;
12             C := A + B;
13             when Test_2 =>
14                 raise Custom_Exception with "Custom_Exception raised!";
15             end case;
16     end Num_Exception_Test;
17
18     pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;

```

Listing 321: check_exception.ads

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID);

```

Listing 322: check_exception.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Check_Exception (ID : Test_ID) is
5 begin
6     Num_Exception_Test (ID);
7 exception
8     when Constraint_Error =>
9         Put_Line ("Constraint_Error detected!");
10        raise;
11     when E : others =>
12         Put_Line (Exception_Message (E));
13         raise Another_Exception;
14 end Check_Exception;

```

Listing 323: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with Ada.Exceptions; use Ada.Exceptions;
4
5 with Tests; use Tests;
6 with Check_Exception;
7
8 procedure Main is
9     type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14
15         procedure Check_Handle_Exception (ID : Test_ID) is
16         begin
17             Check_Exception (ID);
18         exception
19             when Constraint_Error =>
20                 Put_Line ("Constraint_Error"
21                     & " (raised by Check_Exception) detected!");

```

(continues on next page)

(continued from previous page)

```

22     when E : others =>
23         Put_Line (Exception_Name (E)
24             & " (raised by Check_Exception) detected!");
25     end Check_Handle_Exception;
26
27     begin
28         case TC is
29         when Exception_1_Chk =>
30             Check_Handle_Exception (Test_1);
31         when Exception_2_Chk =>
32             Check_Handle_Exception (Test_2);
33         end case;
34     end Check;
35
36     begin
37         if Argument_Count < 1 then
38             Put_Line ("ERROR: missing arguments! Exiting...");
39             return;
40         elsif Argument_Count > 1 then
41             Put_Line ("Ignoring additional arguments...");
42         end if;
43
44         Check (Test_Case_Index'Value (Argument (1)));
45     end Main;

```

18.11 Tasking

18.11.1 Display Service

Listing 324: display_services.ads

```

1  package Display_Services is
2
3      task type Display_Service is
4          entry Display (S : String);
5          entry Display (I : Integer);
6      end Display_Service;
7
8  end Display_Services;

```

Listing 325: display_services.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Display_Services is
4
5      task body Display_Service is
6      begin
7          loop
8              select
9                  accept Display (S : String) do
10                     Put_Line (S);
11                 end Display;
12             or
13                 accept Display (I : Integer) do
14                     Put_Line (Integer'Image (I));
15                 end Display;
16             or

```

(continues on next page)

(continued from previous page)

```

17         terminate;
18     end select;
19 end loop;
20 end Display_Service;
21
22 end Display_Services;

```

Listing 326: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Display_Services; use Display_Services;
5
6  procedure Main is
7      type Test_Case_Index is (Display_Service_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10         Display : Display_Service;
11     begin
12         case TC is
13             when Display_Service_Chk =>
14                 Display.Display ("Hello");
15                 delay 0.5;
16                 Display.Display ("Hello again");
17                 delay 0.5;
18                 Display.Display (55);
19                 delay 0.5;
20         end case;
21     end Check;
22
23     begin
24         if Argument_Count < 1 then
25             Put_Line ("ERROR: missing arguments! Exiting...");
26             return;
27         elsif Argument_Count > 1 then
28             Put_Line ("Ignoring additional arguments...");
29         end if;
30
31         Check (Test_Case_Index'Value (Argument (1)));
32     end Main;

```

18.11.2 Event Manager

Listing 327: event_managers.ads

```

1  with Ada.Real_Time; use Ada.Real_Time;
2
3  package Event_Managers is
4
5      task type Event_Manager is
6          entry Start (ID : Natural);
7          entry Event (T : Time);
8      end Event_Manager;
9
10 end Event_Managers;

```

Listing 328: event_managers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Event_Managers is
4
5      task body Event_Manager is
6          Event_ID    : Natural := 0;
7          Event_Delay : Time;
8      begin
9          accept Start (ID : Natural) do
10             Event_ID := ID;
11         end Start;
12
13         accept Event (T : Time) do
14             Event_Delay := T;
15         end Event;
16
17         delay until Event_Delay;
18
19         Put_Line ("Event #" & Natural'Image (Event_ID));
20     end Event_Manager;
21
22 end Event_Managers;

```

Listing 329: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Event_Managers;   use Event_Managers;
5  with Ada.Real_Time;    use Ada.Real_Time;
6
7  procedure Main is
8      type Test_Case_Index is (Event_Manager_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         Ev_Mng : array (1 .. 5) of Event_Manager;
12     begin
13         case TC is
14             when Event_Manager_Chk =>
15                 for I in Ev_Mng'Range loop
16                     Ev_Mng (I).Start (I);
17                 end loop;
18                 Ev_Mng (1).Event (Clock + Seconds (5));
19                 Ev_Mng (2).Event (Clock + Seconds (3));
20                 Ev_Mng (3).Event (Clock + Seconds (1));
21                 Ev_Mng (4).Event (Clock + Seconds (2));
22                 Ev_Mng (5).Event (Clock + Seconds (4));
23             end case;
24     end Check;
25
26     begin
27         if Argument_Count < 1 then
28             Put_Line ("ERROR: missing arguments! Exiting...");
29             return;
30         elsif Argument_Count > 1 then
31             Put_Line ("Ignoring additional arguments...");
32         end if;
33
34         Check (Test_Case_Index'Value (Argument (1)));

```

(continues on next page)

(continued from previous page)

```

35 end Main;

```

18.11.3 Generic Protected Queue

Listing 330: gen_queues.ads

```

1  generic
2      type Queue_Index is mod <>;
3      type T is private;
4  package Gen_Queues is
5
6      type Queue_Array is array (Queue_Index) of T;
7
8      protected type Queue is
9          function Empty return Boolean;
10         function Full return Boolean;
11         entry Push (V : T);
12         entry Pop (V : out T);
13     private
14         N : Natural := 0;
15         Idx : Queue_Index := Queue_Array'First;
16         A : Queue_Array;
17     end Queue;
18
19 end Gen_Queues;

```

Listing 331: gen_queues.adb

```

1  package body Gen_Queues is
2
3      protected body Queue is
4
5          function Empty return Boolean is
6              (N = 0);
7
8          function Full return Boolean is
9              (N = A'Length);
10
11         entry Push (V : T) when not Full is
12             begin
13                 A (Idx) := V;
14
15                 Idx := Idx + 1;
16                 N := N + 1;
17             end Push;
18
19         entry Pop (V : out T) when not Empty is
20             begin
21                 N := N - 1;
22
23                 V := A (Idx - Queue_Index (N) - 1);
24             end Pop;
25
26         end Queue;
27
28     end Gen_Queues;

```

Listing 332: queue_tests.ads

```

1 package Queue_Tests is
2
3     procedure Simple_Test;
4
5     procedure Concurrent_Test;
6
7 end Queue_Tests;
```

Listing 333: queue_tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_Queues;
4
5 package body Queue_Tests is
6
7     Max : constant := 10;
8     type Queue_Mod is mod Max;
9
10    procedure Simple_Test is
11        package Queues_Float is new Gen_Queues (Queue_Mod, Float);
12
13        Q_F : Queues_Float.Queue;
14        V : Float;
15    begin
16        V := 10.0;
17        while not Q_F.Full loop
18            Q_F.Push (V);
19            V := V + 1.5;
20        end loop;
21
22        while not Q_F.Empty loop
23            Q_F.Pop (V);
24            Put_Line ("Value from queue: " & Float'Image (V));
25        end loop;
26    end Simple_Test;
27
28    procedure Concurrent_Test is
29        package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
30
31        Q_I : Queues_Integer.Queue;
32
33        task T_Producer;
34        task T_Consumer;
35
36        task body T_Producer is
37            V : Integer := 100;
38        begin
39            for I in 1 .. 2 * Max loop
40                Q_I.Push (V);
41                V := V + 1;
42            end loop;
43        end T_Producer;
44
45        task body T_Consumer is
46            V : Integer;
47        begin
48            delay 1.5;
```

(continues on next page)

(continued from previous page)

```

50     while not Q_I.Empty loop
51         Q_I.Pop (V);
52         Put_Line ("Value from queue: " & Integer'Image (V));
53         delay 0.2;
54     end loop;
55 end T_Consumer;
56 begin
57     null;
58 end Concurrent_Test;
59
60 end Queue_Tests;

```

Listing 334: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Queue_Tests;      use Queue_Tests;
5
6  procedure Main is
7      type Test_Case_Index is (Simple_Queue_Chk,
8                               Concurrent_Queue_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when Simple_Queue_Chk =>
14                 Simple_Test;
15             when Concurrent_Queue_Chk =>
16                 Concurrent_Test;
17         end case;
18     end Check;
19
20     begin
21         if Argument_Count < 1 then
22             Put_Line ("ERROR: missing arguments! Exiting...");
23             return;
24         elsif Argument_Count > 1 then
25             Put_Line ("Ignoring additional arguments...");
26         end if;
27
28         Check (Test_Case_Index'Value (Argument (1)));
29     end Main;

```

18.12 Design by contracts

18.12.1 Price Range

Listing 335: prices.ads

```

1  package Prices is
2
3      type Amount is delta 10.0 ** (-2) digits 12;
4
5      -- subtype Price is Amount range 0.0 .. Amount'Last;
6
7      subtype Price is Amount
8          with Static_Predicate => Price >= 0.0;

```

(continues on next page)

(continued from previous page)

```

9
10 end Prices;

```

Listing 336: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Prices;           use Prices;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Price_Range_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
13
14         procedure Check_Range (A : Amount) is
15             P : constant Price := A;
16             begin
17                 Put_Line ("Price: " & Price'Image (P));
18             end Check_Range;
19
20         begin
21             case TC is
22                 when Price_Range_Chk =>
23                     Check_Range (-2.0);
24             end case;
25         exception
26             when Constraint_Error =>
27                 Put_Line ("Constraint_Error detected (NOT as expected).");
28             when Assert_Failure =>
29                 Put_Line ("Assert_Failure detected (as expected).");
30         end Check;
31
32     begin
33         if Argument_Count < 1 then
34             Put_Line ("ERROR: missing arguments! Exiting...");
35             return;
36         elsif Argument_Count > 1 then
37             Put_Line ("Ignoring additional arguments...");
38         end if;
39
40         Check (Test_Case_Index'Value (Argument (1)));
41     end Main;

```

18.12.2 Pythagorean Theorem: Predicate

Listing 337: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record
6          H      : Length := 0;
7          -- Hypotenuse
8          C1, C2 : Length := 0;

```

(continues on next page)

(continued from previous page)

```
9      -- Catheti / legs
10 end record
11 with Dynamic_Predicate => H * H = C1 * C1 + C2 * C2;
12
13 function Init (H, C1, C2 : Length) return Right_Triangle is
14     ((H, C1, C2));
15
16 end Triangles;
```

Listing 338: triangles-io.ads

```
1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;
```

Listing 339: triangles-io.adb

```
1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         "(" & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")";
8
9 end Triangles.IO;
```

Listing 340: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
```

(continues on next page)

(continued from previous page)

```

30     end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;

```

18.12.3 Pythagorean Theorem: Precondition

Listing 341: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record
6          H      : Length := 0;
7          -- Hypotenuse
8          C1, C2 : Length := 0;
9          -- Catheti / legs
10     end record;
11
12     function Init (H, C1, C2 : Length) return Right_Triangle is
13         ((H, C1, C2))
14         with Pre => H * H = C1 * C1 + C2 * C2;
15
16 end Triangles;

```

Listing 342: triangles-io.ads

```

1  package Triangles.IO is
2
3      function Image (T : Right_Triangle) return String;
4
5  end Triangles.IO;

```

Listing 343: triangles-io.adb

```

1  package body Triangles.IO is
2
3      function Image (T : Right_Triangle) return String is
4          "(" & Length'Image (T.H)

```

(continues on next page)

(continued from previous page)

```

5      & ", " & Length'Image (T.C1)
6      & ", " & Length'Image (T.C2)
7      & ")");
8
9  end Triangles.IO;
```

Listing 344: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Triangles;        use Triangles;
6  with Triangles.IO;     use Triangles.IO;
7
8  procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;
```

18.12.4 Pythagorean Theorem: Postcondition

Listing 345: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is record
6     H      : Length := 0;
7     -- Hypotenuse
8     C1, C2 : Length := 0;
9     -- Catheti / legs
10  end record;
11
12  function Init (H, C1, C2 : Length) return Right_Triangle is
13    ((H, C1, C2))
14    with Post => (Init'Result.H * Init'Result.H
15                 = Init'Result.C1 * Init'Result.C1
16                 + Init'Result.C2 * Init'Result.C2);
17
18 end Triangles;
```

Listing 346: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;
```

Listing 347: triangles-io.adb

```

1 package body Triangles.IO is
2
3   function Image (T : Right_Triangle) return String is
4     (" " & Length'Image (T.H)
5     & ", " & Length'Image (T.C1)
6     & ", " & Length'Image (T.C2)
7     & " ");
8
9 end Triangles.IO;
```

Listing 348: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10  type Test_Case_Index is
11    (Triangle_8_6_Pass_Chk,
12     Triangle_8_6_Fail_Chk,
13     Triangle_10_24_Pass_Chk,
14     Triangle_10_24_Fail_Chk,
15     Triangle_18_24_Pass_Chk,
16     Triangle_18_24_Fail_Chk);
```

(continues on next page)

(continued from previous page)

```

17
18 procedure Check (TC : Test_Case_Index) is
19
20     procedure Check_Triangle (H, C1, C2 : Length) is
21         T : Right_Triangle;
22     begin
23         T := Init (H, C1, C2);
24         Put_Line (Image (T));
25     exception
26         when Constraint_Error =>
27             Put_Line ("Constraint_Error detected (NOT as expected).");
28         when Assert_Failure =>
29             Put_Line ("Assert_Failure detected (as expected).");
30     end Check_Triangle;
31
32 begin
33     case TC is
34         when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35         when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36         when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37         when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38         when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39         when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40     end case;
41 end Check;
42
43 begin
44     if Argument_Count < 1 then
45         Put_Line ("ERROR: missing arguments! Exiting...");
46         return;
47     elsif Argument_Count > 1 then
48         Put_Line ("Ignoring additional arguments...");
49     end if;
50
51     Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

18.12.5 Pythagorean Theorem: Type Invariant

Listing 349: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is private
6         with Type_Invariant => Check (Right_Triangle);
7
8     function Check (T : Right_Triangle) return Boolean;
9
10    function Init (H, C1, C2 : Length) return Right_Triangle;
11
12 private
13
14    type Right_Triangle is record
15        H : Length := 0;
16        -- Hypotenuse
17        C1, C2 : Length := 0;
18        -- Catheti / legs
19    end record;

```

(continues on next page)

(continued from previous page)

```

20
21  function Init (H, C1, C2 : Length) return Right_Triangle is
22      ((H, C1, C2));
23
24  function Check (T : Right_Triangle) return Boolean is
25      (T.H * T.H = T.C1 * T.C1 + T.C2 * T.C2);
26
27  end Triangles;

```

Listing 350: triangles-io.ads

```

1  package Triangles.IO is
2
3      function Image (T : Right_Triangle) return String;
4
5  end Triangles.IO;

```

Listing 351: triangles-io.adb

```

1  package body Triangles.IO is
2
3      function Image (T : Right_Triangle) return String is
4          "(" & Length'Image (T.H)
5          & ", " & Length'Image (T.C1)
6          & ", " & Length'Image (T.C2)
7          & ")";
8
9  end Triangles.IO;

```

Listing 352: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO; use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Triangles; use Triangles;
6  with Triangles.IO; use Triangles.IO;
7
8  procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");

```

(continues on next page)

(continued from previous page)

```

30     end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;

```

18.12.6 Primary Colors

Listing 353: color_types.ads

```

1  package Color_Types is
2
3      type HTML_Color is
4          (Salmon,
5           Firebrick,
6           Red,
7           Darkred,
8           Lime,
9           Forestgreen,
10          Green,
11          Darkgreen,
12          Blue,
13          Mediumblue,
14          Darkblue);
15
16      subtype Int_Color is Integer range 0 .. 255;
17
18      function Image (I : Int_Color) return String;
19
20      type RGB is record
21          Red   : Int_Color;
22          Green : Int_Color;
23          Blue  : Int_Color;
24      end record;
25
26      function To_RGB (C : HTML_Color) return RGB;
27
28      function Image (C : RGB) return String;
29
30      type HTML_Color_RGB_Array is array (HTML_Color) of RGB;
31
32      To_RGB_Lookup_Table : constant HTML_Color_RGB_Array

```

(continues on next page)

(continued from previous page)

```

33     := (Salmon      => (16#FA#, 16#80#, 16#72#),
34         Firebrick  => (16#B2#, 16#22#, 16#22#),
35         Red         => (16#FF#, 16#00#, 16#00#),
36         Darkred    => (16#8B#, 16#00#, 16#00#),
37         Lime       => (16#00#, 16#FF#, 16#00#),
38         Forestgreen => (16#22#, 16#8B#, 16#22#),
39         Green      => (16#00#, 16#80#, 16#00#),
40         Darkgreen  => (16#00#, 16#64#, 16#00#),
41         Blue       => (16#00#, 16#00#, 16#FF#),
42         Mediumblue => (16#00#, 16#00#, 16#CD#),
43         Darkblue   => (16#00#, 16#00#, 16#8B#));
44
45     subtype HTML_RGB_Color is HTML_Color
46     with Static_Predicate => HTML_RGB_Color in Red | Green | Blue;
47
48     function To_Int_Color (C : HTML_Color;
49                           S : HTML_RGB_Color) return Int_Color;
50     -- Convert to hexadecimal value for the selected RGB component S
51
52 end Color_Types;

```

Listing 354: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_RGB (C : HTML_Color) return RGB is
6      begin
7          return To_RGB_Lookup_Table (C);
8      end To_RGB;
9
10     function To_Int_Color (C : HTML_Color;
11                           S : HTML_RGB_Color) return Int_Color is
12     C_RGB : constant RGB := To_RGB (C);
13     begin
14         case S is
15             when Red    => return C_RGB.Red;
16             when Green => return C_RGB.Green;
17             when Blue  => return C_RGB.Blue;
18         end case;
19     end To_Int_Color;
20
21     function Image (I : Int_Color) return String is
22     subtype Str_Range is Integer range 1 .. 10;
23     S : String (Str_Range);
24     begin
25         Ada.Integer_Text_IO.Put (To    => S,
26                                Item  => I,
27                                Base  => 16);
28
29         return S;
30     end Image;
31
32     function Image (C : RGB) return String is
33     begin
34         return ("(Red => " & Image (C.Red)
35                & ", Green => " & Image (C.Green)
36                & ", Blue => " & Image (C.Blue)
37                & ")");
38     end Image;

```

(continues on next page)

(continued from previous page)

39 `end Color_Types;`

Listing 355: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_Red_Chk,
9           HTML_Color_Green_Chk,
10          HTML_Color_Blue_Chk);
11
12      procedure Check (TC : Test_Case_Index) is
13
14          procedure Check_HTML_Colors (S : HTML_RGB_Color) is
15              begin
16                  Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
17                  for I in HTML_Color'Range loop
18                      Put_Line (HTML_Color'Image (I) & " => "
19                              & Image (To_Int_Color (I, S)) & ".");
20                  end loop;
21              end Check_HTML_Colors;
22
23          begin
24              case TC is
25                  when HTML_Color_Red_Chk =>
26                      Check_HTML_Colors (Red);
27                  when HTML_Color_Green_Chk =>
28                      Check_HTML_Colors (Green);
29                  when HTML_Color_Blue_Chk =>
30                      Check_HTML_Colors (Blue);
31              end case;
32          end Check;
33
34      begin
35          if Argument_Count < 1 then
36              Put_Line ("ERROR: missing arguments! Exiting...");
37              return;
38          elsif Argument_Count > 1 then
39              Put_Line ("Ignoring additional arguments...");
40          end if;
41
42          Check (Test_Case_Index'Value (Argument (1)));
43      end Main;

```

18.13 Object-oriented programming

18.13.1 Simple type extension

Listing 356: type_extensions.ads

```

1  package Type_Extensions is
2
3      type T_Float is tagged record
4          F : Float;

```

(continues on next page)

(continued from previous page)

```

5  end record;
6
7  function Init (F : Float) return T_Float;
8
9  function Init (I : Integer) return T_Float;
10
11 function Image (T : T_Float) return String;
12
13 type T_Mixed is new T_Float with record
14     I : Integer;
15 end record;
16
17 function Init (F : Float) return T_Mixed;
18
19 function Init (I : Integer) return T_Mixed;
20
21 function Image (T : T_Mixed) return String;
22
23 end Type_Extensions;

```

Listing 357: type_extensions.adb

```

1  package body Type_Extensions is
2
3      function Init (F : Float) return T_Float is
4      begin
5          return ((F => F));
6      end Init;
7
8      function Init (I : Integer) return T_Float is
9      begin
10         return ((F => Float (I)));
11     end Init;
12
13     function Init (F : Float) return T_Mixed is
14     begin
15         return ((F => F,
16                 I => Integer (F)));
17     end Init;
18
19     function Init (I : Integer) return T_Mixed is
20     begin
21         return ((F => Float (I),
22                 I => I));
23     end Init;
24
25     function Image (T : T_Float) return String is
26     begin
27         return "{ F => " & Float'Image (T.F) & " }";
28     end Image;
29
30     function Image (T : T_Mixed) return String is
31     begin
32         return "{ F => " & Float'Image (T.F)
33             & ", I => " & Integer'Image (T.I) & " }";
34     end Image;
35
36 end Type_Extensions;

```

Listing 358: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Type_Extensions; use Type_Extensions;
5
6 procedure Main is
7
8     type Test_Case_Index is
9         (Type_Extension_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         F1, F2 : T_Float;
13         M1, M2 : T_Mixed;
14     begin
15         case TC is
16             when Type_Extension_Chk =>
17                 F1 := Init (2.0);
18                 F2 := Init (3);
19                 M1 := Init (4.0);
20                 M2 := Init (5);
21
22                 if M2 in T_Float'Class then
23                     Put_Line ("T_Mixed is in T_Float'Class as expected");
24                 end if;
25
26                 Put_Line ("F1: " & Image (F1));
27                 Put_Line ("F2: " & Image (F2));
28                 Put_Line ("M1: " & Image (M1));
29                 Put_Line ("M2: " & Image (M2));
30             end case;
31         end Check;
32
33     begin
34         if Argument_Count < 1 then
35             Put_Line ("ERROR: missing arguments! Exiting...");
36             return;
37         elsif Argument_Count > 1 then
38             Put_Line ("Ignoring additional arguments...");
39         end if;
40
41         Check (Test_Case_Index'Value (Argument (1)));
42     end Main;
```

18.13.2 Online Store

Listing 359: online_store.ads

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 package Online_Store is
4
5     type Amount is delta 10.0**(-2) digits 10;
6
7     subtype Percentage is Amount range 0.0 .. 1.0;
8
9     type Member is tagged record
10         Start : Year_Number;
11     end record;
```

(continues on next page)

(continued from previous page)

```

12
13  type Member_Access is access Member'Class;
14
15  function Get_Status (M : Member) return String;
16
17  function Get_Price (M : Member;
18                    P : Amount) return Amount;
19
20  type Full_Member is new Member with record
21    Discount : Percentage;
22  end record;
23
24  function Get_Status (M : Full_Member) return String;
25
26  function Get_Price (M : Full_Member;
27                    P : Amount) return Amount;
28
29  end Online_Store;

```

Listing 360: online_store.adb

```

1  package body Online_Store is
2
3    function Get_Status (M : Member) return String is
4      ("Associate Member");
5
6    function Get_Status (M : Full_Member) return String is
7      ("Full Member");
8
9    function Get_Price (M : Member;
10                     P : Amount) return Amount is (P);
11
12    function Get_Price (M : Full_Member;
13                     P : Amount) return Amount is
14      (P * (1.0 - M.Discount));
15
16  end Online_Store;

```

Listing 361: online_store-tests.ads

```

1  package Online_Store.Tests is
2
3    procedure Simple_Test;
4
5  end Online_Store.Tests;

```

Listing 362: online_store-tests.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Online_Store.Tests is
4
5    procedure Simple_Test is
6
7      type Member_Due_Amount is record
8        Member      : Member_Access;
9        Due_Amount : Amount;
10     end record;
11
12     function Get_Price (MA : Member_Due_Amount) return Amount is

```

(continues on next page)

(continued from previous page)

```

13   begin
14       return MA.Member.Get_Price (MA.Due_Amount);
15   end Get_Price;
16
17   type Member_Due_Amounts is array (Positive range <>) of Member_Due_Amount;
18
19   DB : constant Member_Due_Amounts (1 .. 4)
20       := ((Member      => new Member'(Start => 2010),
21           Due_Amount => 250.0),
22          (Member      => new Full_Member'(Start  => 1998,
23                                           Discount => 0.1),
24           Due_Amount => 160.0),
25          (Member      => new Full_Member'(Start  => 1987,
26                                           Discount => 0.2),
27           Due_Amount => 400.0),
28          (Member      => new Member'(Start => 2013),
29           Due_Amount => 110.0));
30   begin
31       for I in DB'Range loop
32           Put_Line ("Member #" & Positive'Image (I));
33           Put_Line ("Status: " & DB (I).Member.Get_Status);
34           Put_Line ("Since: " & Year_Number'Image (DB (I).Member.Start));
35           Put_Line ("Due Amount: " & Amount'Image (Get_Price (DB (I))));
36           Put_Line ("-----");
37       end loop;
38   end Simple_Test;
39
40 end Online_Store.Tests;

```

Listing 363: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Online_Store;        use Online_Store;
5  with Online_Store.Tests;  use Online_Store.Tests;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Type_Chk,
11          Unit_Test_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14
15         function Result_Image (Result : Boolean) return String is
16             (if Result then "OK" else "not OK");
17
18         begin
19             case TC is
20                 when Type_Chk =>
21                     declare
22                         AM : constant Member      := (Start  => 2002);
23                         FM : constant Full_Member := (Start  => 1990,
24                                                         Discount => 0.2);
25                     begin
26                         Put_Line ("Testing Status of Associate Member Type => "
27                                 & Result_Image (AM.Get_Status = "Associate Member"));
28                         Put_Line ("Testing Status of Full Member Type => "
29                                 & Result_Image (FM.Get_Status = "Full Member"));
30                         Put_Line ("Testing Discount of Associate Member Type => "

```

(continues on next page)

(continued from previous page)

```

31         & Result_Image (AM.Get_Price (100.0) = 100.0));
32     Put_Line ("Testing Discount of Full Member Type => "
33         & Result_Image (FM.Get_Price (100.0) = 80.0));
34     end;
35     when Unit_Test_Chk =>
36         Simple_Test;
37     end case;
38 end Check;
39
40 begin
41     if Argument_Count < 1 then
42         Put_Line ("ERROR: missing arguments! Exiting...");
43         return;
44     elsif Argument_Count > 1 then
45         Put_Line ("Ignoring additional arguments...");
46     end if;
47
48     Check (Test_Case_Index'Value (Argument (1)));
49 end Main;

```

18.14 Standard library: Containers

18.14.1 Simple todo list

Listing 364: todo_lists.ads

```

1  with Ada.Containers.Vectors;
2
3  package Todo_Lists is
4
5      type Todo_Item is access String;
6
7      package Todo_List_Pkg is new Ada.Containers.Vectors
8          (Index_Type => Natural,
9           Element_Type => Todo_Item);
10
11     subtype Todo_List is Todo_List_Pkg.Vector;
12
13     procedure Add (Todos : in out Todo_List;
14                 Item : String);
15
16     procedure Display (Todos : Todo_List);
17
18 end Todo_Lists;

```

Listing 365: todo_lists.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Todo_Lists is
4
5      procedure Add (Todos : in out Todo_List;
6                  Item : String) is
7      begin
8          Todos.Append (new String'(Item));
9      end Add;
10
11     procedure Display (Todos : Todo_List) is

```

(continues on next page)

(continued from previous page)

```

12   begin
13       Put_Line ("TO-DO LIST");
14       for T of Todos loop
15           Put_Line (T.all);
16       end loop;
17   end Display;
18
19 end Todo_Lists;

```

Listing 366: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Todo_Lists;       use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List;
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreadsheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27             end case;
28     end Check;
29
30 begin
31     if Argument_Count < 1 then
32         Put_Line ("ERROR: missing arguments! Exiting...");
33         return;
34     elsif Argument_Count > 1 then
35         Put_Line ("Ignoring additional arguments...");
36     end if;
37
38     Check (Test_Case_Index'Value (Argument (1)));
39 end Main;

```

18.14.2 List of unique integers

Listing 367: ops.ads

```

1  with Ada.Containers.Ordered_Sets;
2
3  package Ops is
4

```

(continues on next page)

(continued from previous page)

```

5  type Int_Array is array (Positive range <>) of Integer;
6
7  package Integer_Sets is new Ada.Containers.Ordered_Sets
8    (Element_Type => Integer);
9
10 subtype Int_Set is Integer_Sets.Set;
11
12 function Get_Unique (A : Int_Array) return Int_Set;
13
14 function Get_Unique (A : Int_Array) return Int_Array;
15
16 end Ops;

```

Listing 368: ops.adb

```

1  package body Ops is
2
3    function Get_Unique (A : Int_Array) return Int_Set is
4      S : Int_Set;
5      begin
6        for E of A loop
7          S.Include (E);
8        end loop;
9
10       return S;
11     end Get_Unique;
12
13     function Get_Unique (A : Int_Array) return Int_Array is
14       S : constant Int_Set := Get_Unique (A);
15       AR : Int_Array (1 .. Positive (S.Length));
16       I : Positive := 1;
17       begin
18         for E of S loop
19           AR (I) := E;
20           I := I + 1;
21         end loop;
22
23         return AR;
24       end Get_Unique;
25
26 end Ops;

```

Listing 369: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;          use Ada.Text_IO;
3
4  with Ops;                  use Ops;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Get_Unique_Set_Chk,
9       Get_Unique_Array_Chk);
10
11    procedure Check (TC : Test_Case_Index;
12                   A : Int_Array) is
13
14      procedure Display_Unique_Set (A : Int_Array) is
15        S : constant Int_Set := Get_Unique (A);
16      begin

```

(continues on next page)

(continued from previous page)

```

17     for E of S loop
18         Put_Line (Integer'Image (E));
19     end loop;
20 end Display_Unique_Set;
21
22 procedure Display_Unique_Array (A : Int_Array) is
23     AU : constant Int_Array := Get_Unique (A);
24 begin
25     for E of AU loop
26         Put_Line (Integer'Image (E));
27     end loop;
28 end Display_Unique_Array;
29
30 begin
31     case TC is
32         when Get_Unique_Set_Chk => Display_Unique_Set (A);
33         when Get_Unique_Array_Chk => Display_Unique_Array (A);
34     end case;
35 end Check;
36
37 begin
38     if Argument_Count < 3 then
39         Put_Line ("ERROR: missing arguments! Exiting...");
40         return;
41     else
42         declare
43             A : Int_Array (1 .. Argument_Count - 1);
44         begin
45             for I in A'Range loop
46                 A (I) := Integer'Value (Argument (1 + I));
47             end loop;
48             Check (Test_Case_Index'Value (Argument (1)), A);
49         end;
50     end if;
51 end Main;

```

18.15 Standard library: Dates & Times

18.15.1 Holocene calendar

Listing 370: to_holocene_year.adb

```

1 with Ada.Calendar; use Ada.Calendar;
2
3 function To_Holocene_Year (T : Time) return Integer is
4 begin
5     return Year (T) + 10_000;
6 end To_Holocene_Year;

```

Listing 371: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Calendar;          use Ada.Calendar;
4
5 with To_Holocene_Year;
6
7 procedure Main is

```

(continues on next page)

(continued from previous page)

```

8  type Test_Case_Index is
9      (Holocene_Chk);
10
11  procedure Display_Holocene_Year (Y : Year_Number) is
12      HY : Integer;
13  begin
14      HY := To_Holocene_Year (Time_Of (Y, 1, 1));
15      Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
16      Put_Line ("Year (Holocene): " & Integer'Image (HY));
17  end Display_Holocene_Year;
18
19  procedure Check (TC : Test_Case_Index) is
20  begin
21      case TC is
22          when Holocene_Chk =>
23              Display_Holocene_Year (2012);
24              Display_Holocene_Year (2020);
25      end case;
26  end Check;
27
28  begin
29      if Argument_Count < 1 then
30          Put_Line ("ERROR: missing arguments! Exiting...");
31          return;
32      elsif Argument_Count > 1 then
33          Put_Line ("Ignoring additional arguments...");
34      end if;
35
36      Check (Test_Case_Index'Value (Argument (1)));
37  end Main;

```

18.15.2 List of events

Listing 372: events.ads

```

1  with Ada.Containers.Vectors;
2
3  package Events is
4
5      type Event_Item is access String;
6
7      package Event_Item_Containers is new
8          Ada.Containers.Vectors
9              (Index_Type => Positive,
10               Element_Type => Event_Item);
11
12      subtype Event_Items is Event_Item_Containers.Vector;
13
14  end Events;

```

Listing 373: events-lists.ads

```

1  with Ada.Calendar; use Ada.Calendar;
2  with Ada.Containers.Ordered_Maps;
3
4  package Events.Lists is
5
6      type Event_List is tagged private;
7

```

(continues on next page)

(continued from previous page)

```

8   procedure Add (Events      : in out Event_List;
9                  Event_Time  : Time;
10                 Event       : String);
11
12   procedure Display (Events : Event_List);
13
14 private
15
16   package Event_Time_Item_Containers is new
17     Ada.Containers.Ordered_Maps
18     (Key_Type      => Time,
19      Element_Type  => Event_Items,
20      "="          => Event_Item_Containers "=");
21
22   type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;

```

Listing 374: events-lists.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4  package body Events.Lists is
5
6      procedure Add (Events      : in out Event_List;
7                    Event_Time  : Time;
8                    Event       : String) is
9          use Event_Item_Containers;
10         E : constant Event_Item := new String'(Event);
11     begin
12         if not Events.Contains (Event_Time) then
13             Events.Include (Event_Time, Empty_Vector);
14         end if;
15         Events (Event_Time).Append (E);
16     end Add;
17
18     function Date_Image (T : Time) return String is
19         Date_Img : constant String := Image (T);
20     begin
21         return Date_Img (1 .. 10);
22     end;
23
24     procedure Display (Events : Event_List) is
25         use Event_Time_Item_Containers;
26         T : Time;
27     begin
28         Put_Line ("EVENTS LIST");
29         for C in Events.Iterate loop
30             T := Key (C);
31             Put_Line ("- " & Date_Image (T));
32             for I of Events (C) loop
33                 Put_Line ("    - " & I.all);
34             end loop;
35         end loop;
36     end Display;
37
38 end Events.Lists;

```

Listing 375: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5
6  with Events.Lists;          use Events.Lists;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Event_List_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
13         EL : Event_List;
14     begin
15         case TC is
16             when Event_List_Chk =>
17                 EL.Add (Time_Of (2018, 2, 16),
18                     "Final check");
19                 EL.Add (Time_Of (2018, 2, 16),
20                     "Release");
21                 EL.Add (Time_Of (2018, 12, 3),
22                     "Brother's birthday");
23                 EL.Add (Time_Of (2018, 1, 1),
24                     "New Year's Day");
25                 EL.Display;
26         end case;
27     end Check;
28
29 begin
30     if Argument_Count < 1 then
31         Put_Line ("ERROR: missing arguments! Exiting...");
32         return;
33     elsif Argument_Count > 1 then
34         Put_Line ("Ignoring additional arguments...");
35     end if;
36
37     Check (Test_Case_Index'Value (Argument (1)));
38 end Main;

```

18.16 Standard library: Strings

18.16.1 Concatenation

Listing 376: str_concat.ads

```

1  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2
3  package Str_Concat is
4
5      type Unbounded_Strings is array (Positive range <>) of Unbounded_String;
6
7      function Concat (USA           : Unbounded_Strings;
8                     Trim_Str       : Boolean;
9                     Add_Whitespace : Boolean) return Unbounded_String;
10
11     function Concat (USA           : Unbounded_Strings;
12                     Trim_Str       : Boolean;

```

(continues on next page)

(continued from previous page)

```

13         Add_Whitespace : Boolean) return String;
14
15 end Str_Concat;

```

Listing 377: str_concat.adb

```

1  with Ada.Strings; use Ada.Strings;
2
3  package body Str_Concat is
4
5      function Concat (USA           : Unbounded_Strings;
6                      Trim_Str      : Boolean;
7                      Add_Whitespace : Boolean) return Unbounded_String is
8
9          function Retrieve (USA           : Unbounded_Strings;
10                           Trim_Str      : Boolean;
11                           Index         : Positive) return Unbounded_String is
12              US_Internal : Unbounded_String := USA (Index);
13          begin
14              if Trim_Str then
15                  US_Internal := Trim (US_Internal, Both);
16              end if;
17              return US_Internal;
18          end Retrieve;
19
20          US : Unbounded_String := To_Unbounded_String ("");
21      begin
22          for I in USA'First .. USA'Last - 1 loop
23              US := US & Retrieve (USA, Trim_Str, I);
24              if Add_Whitespace then
25                  US := US & " ";
26              end if;
27          end loop;
28          US := US & Retrieve (USA, Trim_Str, USA'Last);
29
30          return US;
31      end Concat;
32
33      function Concat (USA           : Unbounded_Strings;
34                      Trim_Str      : Boolean;
35                      Add_Whitespace : Boolean) return String is
36      begin
37          return To_String (Concat (USA, Trim_Str, Add_Whitespace));
38      end Concat;
39
40 end Str_Concat;

```

Listing 378: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
4
5  with Str_Concat;             use Str_Concat;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Unbounded_Concat_No_Trim_No_WS_Chk,
10           Unbounded_Concat_Trim_No_WS_Chk,
11           String_Concat_Trim_WS_Chk,

```

(continues on next page)

(continued from previous page)

```

12     Concat_Single_Element);
13
14 procedure Check (TC : Test_Case_Index) is
15 begin
16     case TC is
17     when Unbounded_Concat_No_Trim_No_WS_Chk =>
18         declare
19             S : constant Unbounded_Strings := (
20                 To_Unbounded_String ("Hello"),
21                 To_Unbounded_String (" World"),
22                 To_Unbounded_String ("!"));
23         begin
24             Put_Line (To_String (Concat (S, False, False)));
25         end;
26     when Unbounded_Concat_Trim_No_WS_Chk =>
27         declare
28             S : constant Unbounded_Strings := (
29                 To_Unbounded_String (" This "),
30                 To_Unbounded_String (" _is_ "),
31                 To_Unbounded_String (" a "),
32                 To_Unbounded_String (" _check "));
33         begin
34             Put_Line (To_String (Concat (S, True, False)));
35         end;
36     when String_Concat_Trim_WS_Chk =>
37         declare
38             S : constant Unbounded_Strings := (
39                 To_Unbounded_String (" This "),
40                 To_Unbounded_String (" is a "),
41                 To_Unbounded_String (" test. "));
42         begin
43             Put_Line (Concat (S, True, True));
44         end;
45     when Concat_Single_Element =>
46         declare
47             S : constant Unbounded_Strings := (
48                 1 => To_Unbounded_String (" Hi "));
49         begin
50             Put_Line (Concat (S, True, True));
51         end;
52     end case;
53 end Check;
54
55 begin
56     if Argument_Count < 1 then
57         Put_Line ("ERROR: missing arguments! Exiting...");
58         return;
59     elsif Argument_Count > 1 then
60         Put_Line ("Ignoring additional arguments...");
61     end if;
62
63     Check (Test_Case_Index'Value (Argument (1)));
64 end Main;

```

18.16.2 List of events

Listing 379: events.ads

```

1 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2 with Ada.Containers.Vectors;

```

(continues on next page)

(continued from previous page)

```

3
4 package Events is
5
6     subtype Event_Item is Unbounded_String;
7
8     package Event_Item_Containers is new
9         Ada.Containers.Vectors
10         (Index_Type => Positive,
11          Element_Type => Event_Item);
12
13     subtype Event_Items is Event_Item_Containers.Vector;
14
15 end Events;

```

Listing 380: events-lists.ads

```

1 with Ada.Calendar;           use Ada.Calendar;
2 with Ada.Containers.Ordered_Maps;
3
4 package Events.Lists is
5
6     type Event_List is tagged private;
7
8     procedure Add (Events      : in out Event_List;
9                  Event_Time   : Time;
10                 Event        : String);
11
12     procedure Display (Events : Event_List);
13
14 private
15
16     package Event_Time_Item_Containers is new
17         Ada.Containers.Ordered_Maps
18         (Key_Type      => Time,
19          Element_Type   => Event_Items,
20          "="            => Event_Item_Containers "=");
21
22     type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;

```

Listing 381: events-lists.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4 package body Events.Lists is
5
6     procedure Add (Events      : in out Event_List;
7                  Event_Time   : Time;
8                  Event        : String) is
9         use Event_Item_Containers;
10        E : constant Event_Item := To_Unbounded_String (Event);
11    begin
12        if not Events.Contains (Event_Time) then
13            Events.Include (Event_Time, Empty_Vector);
14        end if;
15        Events (Event_Time).Append (E);
16    end Add;
17

```

(continues on next page)

(continued from previous page)

```

18  function Date_Image (T : Time) return String is
19      Date_Img : constant String := Image (T);
20  begin
21      return Date_Img (1 .. 10);
22  end;
23
24  procedure Display (Events : Event_List) is
25      use Event_Time_Item_Containers;
26      T : Time;
27  begin
28      Put_Line ("EVENTS LIST");
29      for C in Events.Iterate loop
30          T := Key (C);
31          Put_Line ("- " & Date_Image (T));
32          for I of Events (C) loop
33              Put_Line ("    - " & To_String (I));
34          end loop;
35      end loop;
36  end Display;
37
38  end Events.Lists;

```

Listing 382: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
6
7  with Events;
8  with Events.Lists;          use Events.Lists;
9
10 procedure Main is
11     type Test_Case_Index is
12         (Unbounded_String_Chk,
13          Event_List_Chk);
14
15     procedure Check (TC : Test_Case_Index) is
16         EL : Event_List;
17     begin
18         case TC is
19             when Unbounded_String_Chk =>
20                 declare
21                     S : constant Events.Event_Item := To_Unbounded_String ("Checked");
22                 begin
23                     Put_Line (To_String (S));
24                 end;
25             when Event_List_Chk =>
26                 EL.Add (Time_Of (2018, 2, 16),
27                     "Final check");
28                 EL.Add (Time_Of (2018, 2, 16),
29                     "Release");
30                 EL.Add (Time_Of (2018, 12, 3),
31                     "Brother's birthday");
32                 EL.Add (Time_Of (2018, 1, 1),
33                     "New Year's Day");
34                 EL.Display;
35             end case;
36     end Check;
37

```

(continues on next page)

(continued from previous page)

```
38 begin
39   if Argument_Count < 1 then
40     Put_Line ("ERROR: missing arguments! Exiting...");
41     return;
42   elsif Argument_Count > 1 then
43     Put_Line ("Ignoring additional arguments...");
44   end if;
45
46   Check (Test_Case_Index'Value (Argument (1)));
47 end Main;
```

18.17 Standard library: Numerics

18.17.1 Decibel Factor

Listing 383: decibels.ads

```
1 package Decibels is
2
3   subtype Decibel is Float;
4   subtype Factor is Float;
5
6   function To_Decibel (F : Factor) return Decibel;
7
8   function To_Factor (D : Decibel) return Factor;
9
10 end Decibels;
```

Listing 384: decibels.adb

```
1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Decibels is
4
5   function To_Decibel (F : Factor) return Decibel is
6   begin
7     return 20.0 * Log (F, 10.0);
8   end To_Decibel;
9
10  function To_Factor (D : Decibel) return Factor is
11  begin
12    return 10.0 ** (D / 20.0);
13  end To_Factor;
14
15 end Decibels;
```

Listing 385: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Decibels; use Decibels;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Db_Chk,
9      Factor_Chk);
10
```

(continues on next page)

(continued from previous page)

```

11  procedure Check (TC : Test_Case_Index; V : Float) is
12
13      package F_IO is new Ada.Text_IO.Float_IO (Factor);
14      package D_IO is new Ada.Text_IO.Float_IO (Decibel);
15
16      procedure Put_Decibel_Cnvt (D : Decibel) is
17          F : constant Factor := To_Factor (D);
18      begin
19          D_IO.Put (D, 0, 2, 0);
20          Put (" dB => Factor of ");
21          F_IO.Put (F, 0, 2, 0);
22          New_Line;
23      end;
24
25      procedure Put_Factor_Cnvt (F : Factor) is
26          D : constant Decibel := To_Decibel (F);
27      begin
28          Put ("Factor of ");
29          F_IO.Put (F, 0, 2, 0);
30          Put (" => ");
31          D_IO.Put (D, 0, 2, 0);
32          Put_Line (" dB");
33      end;
34  begin
35      case TC is
36          when Db_Chk =>
37              Put_Decibel_Cnvt (Decibel (V));
38          when Factor_Chk =>
39              Put_Factor_Cnvt (Factor (V));
40      end case;
41  end Check;
42
43  begin
44      if Argument_Count < 2 then
45          Put_Line ("ERROR: missing arguments! Exiting...");
46          return;
47      elsif Argument_Count > 2 then
48          Put_Line ("Ignoring additional arguments...");
49      end if;
50
51      Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
52  end Main;

```

18.17.2 Root-Mean-Square

Listing 386: signals.ads

```

1  package Signals is
2
3      subtype Sig_Value is Float;
4
5      type Signal is array (Natural range <>) of Sig_Value;
6
7      function Rms (S : Signal) return Sig_Value;
8
9  end Signals;

```

Listing 387: signals.adb

```
1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Signals is
4
5     function Rms (S : Signal) return Sig_Value is
6         Acc : Float := 0.0;
7     begin
8         for V of S loop
9             Acc := Acc + V * V;
10        end loop;
11
12        return Sqrt (Acc / Float (S'Length));
13    end;
14
15 end Signals;
```

Listing 388: signals-std.ads

```
1 package Signals.Std is
2
3     Sample_Rate : Float := 8000.0;
4
5     function Generate_Sine (N : Positive; Freq : Float) return Signal;
6
7     function Generate_Square (N : Positive) return Signal;
8
9     function Generate_Triangular (N : Positive) return Signal;
10
11 end Signals.Std;
```

Listing 389: signals-std.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
3
4 package body Signals.Std is
5
6     function Generate_Sine (N : Positive; Freq : Float) return Signal is
7         S : Signal (0 .. N - 1);
8     begin
9         for I in S'First .. S'Last loop
10             S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
11        end loop;
12
13        return S;
14    end;
15
16     function Generate_Square (N : Positive) return Signal is
17         S : constant Signal (0 .. N - 1) := (others => 1.0);
18     begin
19         return S;
20     end;
21
22     function Generate_Triangular (N : Positive) return Signal is
23         S : Signal (0 .. N - 1);
24         S_Half : constant Natural := S'Last / 2;
25     begin
26         for I in S'First .. S_Half loop
27             S (I) := 1.0 * (Float (I) / Float (S_Half));
```

(continues on next page)

(continued from previous page)

```

28     end loop;
29     for I in S_Half .. S'Last loop
30         S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
31     end loop;
32
33     return S;
34 end;
35
36 end Signals.Std;

```

Listing 390: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Signals;               use Signals;
5  with Signals.Std;          use Signals.Std;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Sine_Signal_Chk,
10           Square_Signal_Chk,
11           Triangular_Signal_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14          package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
15
16          N : constant Positive := 1024;
17          S_Si : constant Signal := Generate_Sine (N, 440.0);
18          S_Sq : constant Signal := Generate_Square (N);
19          S_Tr : constant Signal := Generate_Triangular (N + 1);
20      begin
21          case TC is
22              when Sine_Signal_Chk =>
23                  Put ("RMS of Sine Signal: ");
24                  Sig_IO.Put (Rms (S_Si), 0, 2, 0);
25                  New_Line;
26              when Square_Signal_Chk =>
27                  Put ("RMS of Square Signal: ");
28                  Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
29                  New_Line;
30              when Triangular_Signal_Chk =>
31                  Put ("RMS of Triangular Signal: ");
32                  Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
33                  New_Line;
34          end case;
35      end Check;
36
37      begin
38          if Argument_Count < 1 then
39              Put_Line ("ERROR: missing arguments! Exiting...");
40              return;
41          elsif Argument_Count > 1 then
42              Put_Line ("Ignoring additional arguments...");
43          end if;
44
45          Check (Test_Case_Index'Value (Argument (1)));
46      end Main;

```

18.17.3 Rotation

Listing 391: rotation.ads

```
1 with Ada.Numerics.Complex_Types;
2 use Ada.Numerics.Complex_Types;
3
4 package Rotation is
5
6     type Complex_Points is array (Positive range <>) of Complex;
7
8     function Rotation (N : Positive) return Complex_Points;
9
10 end Rotation;
```

Listing 392: rotation.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2
3 package body Rotation is
4
5     function Rotation (N : Positive) return Complex_Points is
6         C_Angle : constant Complex :=
7             Compose_From_Polar (1.0, 2.0 * Pi / Float (N));
8     begin
9         return C : Complex_Points (1 .. N + 1) do
10             C (1) := Compose_From_Cartesian (1.0, 0.0);
11
12             for I in C'First + 1 .. C'Last loop
13                 C (I) := C (I - 1) * C_Angle;
14             end loop;
15         end return;
16     end;
17
18 end Rotation;
```

Listing 393: angles.ads

```
1 with Rotation; use Rotation;
2
3 package Angles is
4
5     subtype Angle is Float;
6
7     type Angles is array (Positive range <>) of Angle;
8
9     function To_Angles (C : Complex_Points) return Angles;
10
11 end Angles;
```

Listing 394: angles.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Complex_Types; use Ada.Numerics.Complex_Types;
3
4 package body Angles is
5
6     function To_Angles (C : Complex_Points) return Angles is
7     begin
8         return A : Angles (C'Range) do
9             for I in A'Range loop
```

(continues on next page)

(continued from previous page)

```

10         A (I) := Argument (C (I)) / Pi * 180.0;
11     end loop;
12 end return;
13 end To_Angles;
14
15 end Angles;

```

Listing 395: rotation-tests.ads

```

1 package Rotation.Tests is
2
3     procedure Test_Rotation (N : Positive);
4
5     procedure Test_Angles (N : Positive);
6
7 end Rotation.Tests;

```

Listing 396: rotation-tests.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO;
3 with Ada.Numerics;         use Ada.Numerics;
4
5 with Angles;               use Angles;
6
7 package body Rotation.Tests is
8
9     package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
10    package F_IO is new Ada.Text_IO.Float_IO (Float);
11
12    --
13    -- Adapt value due to floating-point inaccuracies
14    --
15
16    function Adapt (C : Complex) return Complex is
17        function Check_Zero (F : Float) return Float is
18            (if F <= 0.0 and F >= -0.01 then 0.0 else F);
19        begin
20            return C_Out : Complex := C do
21                C_Out.Re := Check_Zero (C_Out.Re);
22                C_Out.Im := Check_Zero (C_Out.Im);
23            end return;
24        end Adapt;
25
26    function Adapt (A : Angle) return Angle is
27        (if A <= -179.99 and A >= -180.01 then 180.0 else A);
28
29    procedure Test_Rotation (N : Positive) is
30        C : constant Complex_Points := Rotation (N);
31    begin
32        Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
33        for V of C loop
34            Put ("Point: ");
35            C_IO.Put (Adapt (V), 0, 1, 0);
36            New_Line;
37        end loop;
38    end Test_Rotation;
39
40    procedure Test_Angles (N : Positive) is
41        C : constant Complex_Points := Rotation (N);

```

(continues on next page)

(continued from previous page)

```

42     A : constant Angles.Angles := To_Angles (C);
43   begin
44     Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
45     for V of A loop
46       Put ("Angle: ");
47       F_IO.Put (Adapt (V), 0, 2, 0);
48       Put_Line (" degrees");
49     end loop;
50   end Test_Angles;
51
52 end Rotation.Tests;

```

Listing 397: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Rotation.Tests;        use Rotation.Tests;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Rotation_Chk,
9       Angles_Chk);
10
11    procedure Check (TC : Test_Case_Index; N : Positive) is
12    begin
13      case TC is
14        when Rotation_Chk =>
15          Test_Rotation (N);
16        when Angles_Chk =>
17          Test_Angles (N);
18      end case;
19    end Check;
20
21  begin
22    if Argument_Count < 2 then
23      Put_Line ("ERROR: missing arguments! Exiting...");
24      return;
25    elsif Argument_Count > 2 then
26      Put_Line ("Ignoring additional arguments...");
27    end if;
28
29    Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
30  end Main;

```