Introduction to

Ada:
Laboratories

Gustavo A. Hoffmann

LEARN.
ADA CORE.COM
Introduction to Ada:
Laboratories

Release 2024-06

Gustavo A. Hoffmann

Jun 30, 2024
## 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 ........................................... 11  
2.5 States #2 ........................................ 12  
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 ....................................... 24  
4.3 Temperatures .................................. 28  

5 Records ........................................ 33  
5.1 Directions ..................................... 33  
5.2 Colors ......................................... 35  
5.3 Inventory ..................................... 39  

6 Arrays ........................................ 43  
6.1 Constrained Array ......................... 43  
6.2 Colors: Lookup-Table .................... 45  
6.3 Unconstrained Array .................... 48  
6.4 Product info ................................ 51  
6.5 String_10 ....................................... 54  
6.6 List of Names ................................. 56  

7 More About Types ......................... 61  
7.1 Aggregate Initialization .................. 61  
7.2 Versioning ................................... 63  
7.3 Simple todo list .............................. 65  
7.4 Price list ..................................... 67  

8 Privacy ............................................ 73  
8.1 Directions .................................... 73
8.2 Limited Strings ........................................ 75
8.3 Bonus exercise ........................................ 79
  8.3.1 Colors ............................................. 80
  8.3.2 List of Names ....................................... 80
  8.3.3 Price List .......................................... 80

9 Generics .................................................. 81
  9.1 Display Array .......................................... 81
  9.2 Average of Array of Float ............................. 83
  9.3 Average of Array of Any Type ......................... 85
  9.4 Generic list .......................................... 88

10 Exceptions ................................................ 91
  10.1 Uninitialized Value ................................... 91
  10.2 Numerical Exception .................................. 93
  10.3 Re-raising Exceptions ................................. 95

11 Tasking .................................................... 99
  11.1 Display Service ...................................... 99
  11.2 Event Manager ...................................... 100
  11.3 Generic Protected Queue .............................. 102

12 Design by contracts .................................... 105
  12.1 Price Range .......................................... 105
  12.2 Pythagorean Theorem: Predicate ...................... 106
  12.3 Pythagorean Theorem: Precondition .................. 108
  12.4 Pythagorean Theorem: Postcondition ................ 110
  12.5 Pythagorean Theorem: Type Invariant ............... 112
  12.6 Primary Color ...................................... 114

13 Object-oriented programming ......................... 119
  13.1 Simple type extension ............................... 119
  13.2 Online Store ....................................... 121

14 Standard library: Containers ......................... 127
  14.1 Simple todo list ................................... 127
  14.2 List of unique integers ............................. 129

15 Standard library: Dates & Times ..................... 133
  15.1 Holocene calendar ................................ 133
  15.2 List of events ..................................... 134

16 Standard library: Strings ............................. 139
  16.1 Concatenation ..................................... 139
  16.2 List of events ..................................... 141

17 Standard library: Numerics ........................... 145
  17.1 Decibel Factor .................................... 145
  17.2 Root-Mean-Square ................................ 147
  17.3 Rotation .......................................... 150

18 Solutions .................................................. 155
  18.1 Imperative Language ................................ 155
    18.1.1 Hello World .................................. 155
    18.1.2 Greetings ...................................... 155
    18.1.3 Positive Or Negative ............................ 156
    18.1.4 Numbers ...................................... 156
  18.2 Subprograms ......................................... 157
    18.2.1 Subtract Procedure ............................. 157
    18.2.2 Subtract Function .............................. 158
18.2.3 Equality function ................................................. 159
18.2.4 States ......................................................... 160
18.2.5 States #2 .................................................... 161
18.2.6 States #3 .................................................... 162
18.2.7 States #4 .................................................... 163
18.3 Modular Programming .............................................. 164
18.3.1 Months ....................................................... 164
18.3.2 Operations ................................................... 165
18.4 Strongly typed language ........................................... 167
18.4.1 Colors ......................................................... 167
18.4.2 Integers ....................................................... 169
18.4.3 Temperatures ............................................... 172
18.5 Records .......................................................... 174
18.5.1 Directions ..................................................... 174
18.5.2 Colors ......................................................... 176
18.5.3 Inventory ..................................................... 179
18.6 Arrays ............................................................ 181
18.6.1 Constrained Array ........................................... 181
18.6.2 Colors: Lookup-Table ...................................... 183
18.6.3 Unconstrained Array ....................................... 185
18.6.4 Product info ................................................ 187
18.6.5 String_10 ................................................... 189
18.6.6 List of Names ............................................... 191
18.7 More About Types ............................................... 194
18.7.1 Aggregate Initialization .................................... 194
18.7.2 Versioning ................................................... 196
18.7.3 Simple todo list ............................................ 197
18.7.4 Price list .................................................... 199
18.8 Privacy .......................................................... 201
18.8.1 Directions ..................................................... 201
18.8.2 Limited Strings ............................................ 203
18.9 Generics .......................................................... 206
18.9.1 Display Array ............................................... 206
18.9.2 Average of Array of Float ................................ 208
18.9.3 Average of Array of Any Type ......................... 209
18.9.4 Generic list ................................................ 211
18.10 Exceptions ....................................................... 213
18.10.1 Uninitialized Value ...................................... 213
18.10.2 Numerical Exception ..................................... 215
18.10.3 Re-raising Exceptions ................................... 217
18.11 Tasking .......................................................... 218
18.11.1 Display Service ........................................... 218
18.11.2 Event Manager ............................................ 220
18.11.3 Generic Protected Queue ................................. 221
18.12 Design by contracts ............................................. 224
18.12.1 Price Range ............................................... 224
18.12.2 Pythagorean Theorem: Predicate ...................... 225
18.12.3 Pythagorean Theorem: Precondition ................... 227
18.12.4 Pythagorean Theorem: Postcondition ................ 228
18.12.5 Pythagorean Theorem: Type Invariant ................ 230
18.12.6 Primary Colors .......................................... 232
18.13 Object-oriented programming ................................. 234
18.13.1 Simple type extension ................................. 234
18.13.2 Online Store .............................................. 236
18.14 Standard library: Containers ................................. 239
18.14.1 Simple todo list ......................................... 239
18.14.2 List of unique integers .................................. 240
18.15 Standard library: Dates & Times .............................. 242
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.15 Holocene calendar</td>
<td>242</td>
</tr>
<tr>
<td>18.15.2 List of events</td>
<td>243</td>
</tr>
<tr>
<td>18.16 Standard library: Strings</td>
<td>245</td>
</tr>
<tr>
<td>18.16.1 Concatenation</td>
<td>245</td>
</tr>
<tr>
<td>18.16.2 List of events</td>
<td>247</td>
</tr>
<tr>
<td>18.17 Standard library: Numerics</td>
<td>250</td>
</tr>
<tr>
<td>18.17.1 Decibel Factor</td>
<td>250</td>
</tr>
<tr>
<td>18.17.2 Root-Mean-Square</td>
<td>251</td>
</tr>
<tr>
<td>18.17.3 Rotation</td>
<td>253</td>
</tr>
</tbody>
</table>
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.
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!".

Listing 1: main.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  begin
    null;
  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 (`&`).
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.
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**

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

**Listing 7: display_numbers.adb**

```ada
procedure Display_Numbers (A, B : Integer) is
begin
  -- Implement the application here!
  null;
end Display_Numbers;
```

**Listing 8: main.adb**

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO;   use Ada.Text_IO;
with Display_Numbers;
procedure Main is
  A, B : Integer;
  begin
    if Argument_Count < 2 then
      Put_Line ("ERROR: missing arguments! Exiting...");
    else
      null;
    end if;
  end Main;
```
11
12
13
14
15
16
17
18
19
20

return;
elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...);
end if;

A := Integer'Value (Argument (1));
B := Integer'Value (Argument (2));

Display_Numbers (A, B);
end Main;
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 1: subtract.ads

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

Listing 2: subtract.adb

```ada
procedure Subtract is
begin
  -- Implement the procedure here.
  null;
end Subtract;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Subtract;

procedure Main is
  type Test_Case_Index is
    (Sub_10_1_Chk,
     Sub_10_100_Chk,
     Sub_0_5_Chk,
     Sub_0_Minus_5_Chk);

procedure Check (TC : Test_Case_Index) is
  Result : Integer;
begin
  case TC is
    when Sub_10_1_Chk =>
      Subtract (10, 1, Result);
      Put_Line ("Result: " & Integer'Image (Result));
    when Sub_10_100_Chk =>
      Subtract (10, 100, Result);
```

(continues on next page)
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.
Sub_0_Minus_5_CHK);

procedure Check (TC : Test_Case_Index) is
  Result : Integer;
begin
  case TC is
    when Sub_10_1_CHK =>
      Result := Subtract (10, 1);
      Put_Line ("Result: " & Integer'Image (Result));
    when Sub_10_100_CHK =>
      Result := Subtract (10, 100);
      Put_Line ("Result: " & Integer'Image (Result));
    when Sub_0_5_CHK =>
      Result := Subtract (0, 5);
      Put_Line ("Result: " & Integer'Image (Result));
    when Sub_0_Minus_5_CHK =>
      Result := Subtract (0, -5);
      Put_Line ("Result: " & Integer'Image (Result));
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
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 7: is_equal.ads

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

Listing 8: is_equal.adb

```
procedure Is_Equal is
begin
  -- Implement the function here!
  null;
end Is_Equal;
```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Is_Equal;

procedure Main is
  type Test_Case_Index is
    (Equal_Chk, Inequal_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Display_Equal (A, B : Integer; Equal : Boolean) is
      begin
        Put (Integer'Image (A));
        if Equal then
          Put (" is equal to ");
        else
          Put (" isn't equal to ");
        end if;
        Put_Line (Integer'Image (B) & ".");
      end Display_Equal;

      Result : Boolean;
      begin
        case TC is
          when Equal_Chk =>
            for I in 0 .. 10 loop
              Result := Is_Equal (I, I);
              Display_Equal (I, I, Result);
            end loop;
          when Inequal_Chk =>
            for I in 0 .. 10 loop
              Result := Is_Equal (I, I - 1);
              Display_Equal (I, I - 1, Result);
            end loop;
        end case;
      end Check;

      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
      return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
  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:

<table>
<thead>
<tr>
<th>Number</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Off</td>
</tr>
<tr>
<td>1</td>
<td>On: Simple Processing</td>
</tr>
<tr>
<td>2</td>
<td>On: Advanced Processing</td>
</tr>
</tbody>
</table>

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 10: display_state.ads**

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

**Listing 11: display_state.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_State (State : Integer) is
begin
null;
end Display_State;
```

**Listing 12: main.adb**

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_State;

procedure Main is
    State : Integer;
begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
        end if;
    State := Integer'Value (Argument (1));
    Display_State (State);
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 13: get_hello.ads
   1. `function Get_Hello return String;`

   Listing 14: get_hello.adb
   1. `function Get_Hello return String is`
   2. `begin`
   3. `return "Hello";`
   4. `end Get_Hello;`

   Listing 15: main.adb
   1. `with Ada.Text_IO; use Ada.Text_IO;`
   2. `with Get_Hello;`
   3. `procedure Main is`
   4. `S : constant String := Get_Hello;`
   5. `begin`
   6. `Put_Line (S);`
   7. `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 16: get_state.ads
   1. `function Get_State (State : Integer) return String;`

   Listing 17: get_state.adb
   1. `function Get_State (State : Integer) return String is`
   2. `begin`
   3. `return "";`
   4. `end Get_State;`

   Listing 18: main.adb
   1. `with Ada.Command_Line; use Ada.Command_Line;`
   2. `with Ada.Text_IO; use Ada.Text_IO;`
   3. `with Get_State;`
   (continues on next page)
procedure Main is
  State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  State := Integer'Value (Argument (1));
  Put_Line (Get_State (State));
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 19: is_on.ads

function Is_On (State : Integer) return Boolean;

Listing 20: is_on.adb

function Is_On (State : Integer) return Boolean is
begin
  return False;
end Is_On;

Listing 21: display_on_off.ads

procedure Display_On_Off (State : Integer);
Listing 22: display_on_off.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Is_On;

procedure Display_On_Off (State : Integer) is
begin
  Put_Line ("");
end Display_On_Off;
```

Listing 23: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_On_Off;
with Is_On;

procedure Main is
  State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting..."ัญ);
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments..."ัญ);
  end if;
  State := Integer'Value (Argument (1));
  Display_On_Off (State);
  Put_Line (Boolean'Image (Is_On (State)));
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 most cases, the next state of `N` is simply the next number (`N + 1`).
   - However, if the state is the last one (which is 2 for our machine), the next state must be the first one (in our case: 0).

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

Listing 24: set_next.ads

procedure Set_Next (State : in out Integer);

Listing 25: set_next.adb

procedure Set_Next (State : in out Integer) is
  begin
    null;
  end Set_Next;

Listing 26: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Set_Next;

procedure Main is
  State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  State := Integer'Value (Argument (1));
  Set_Next (State);
  Put_Line (Integer'Image (State));
end Main;
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 1: months.ads

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

Listing 2: months.adb

```
-- Create body of Months package, which includes
-- the implementation of the Display_Months procedure.
procedure Months is
  procedure Display_Months is
    begin
      Put_Line ("Months:"");
      Put_Line ("- " & Jan);
      Put_Line ("- " & Feb);
      Put_Line ("- " & Mar);
```

(continues on next page)
### 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 4: operations.ads

```
package Operations is

    -- Create specification for Operations package, including the declaration of the functions mentioned above.

end Operations;
```

Listing 5: operations.adb

```
package body Operations is

    -- Create body of Operations package.

end Operations;
```

Listing 6: operations-test.ads

```
package Operations.Test is

    -- Create specification for Operations package, including the declaration of the Display procedure:

    -- procedure Display (A, B : Integer);

end Operations.Test;
```

Listing 7: operations-test.adb

```
package body Operations.Test is

    -- Implement body of Operations.Test package.

    procedure Display (A, B : Integer) is

        A_Str : constant String := Integer'Image (A);
        B_Str : constant String := Integer'Image (B);

(continues on next page)"
Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Operations;
with Operations.Test; use Operations.Test;

procedure Main is

   type Test_Case_Index is
      (Operations_Chk,
       Operations_Display_Chk);

   procedure Check (TC : Test_Case_Index) is
     begin
       case TC is
         when Operations_Chk =>
           Put_Line ("Add (100, 2) = "
                     & Integer'Image (Operations.Add (100, 2)));
           Put_Line ("Subtract (100, 2) = "
                     & Integer'Image (Operations.Subtract (100, 2)));
           Put_Line ("Multiply (100, 2) = "
                     & Integer'Image (Operations.Multiply (100, 2)));
           Put_Line ("Divide (100, 2) = "
                     & Integer'Image (Operations.Divide (100, 2)));
         when Operations_Display_Chk =>
           Display (10, 5);
           Display (1, 2);
       end case;
   end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
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:
<table>
<thead>
<tr>
<th>Color</th>
<th>HTML color code (hexa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salmon</td>
<td>#FA8072</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#B22222</td>
</tr>
<tr>
<td>Red</td>
<td>#FF0000</td>
</tr>
<tr>
<td>Darkred</td>
<td>#8B0000</td>
</tr>
<tr>
<td>Lime</td>
<td>#00FF00</td>
</tr>
<tr>
<td>Forestgreen</td>
<td>#228B22</td>
</tr>
<tr>
<td>Green</td>
<td>#00FF00</td>
</tr>
<tr>
<td>Darkgreen</td>
<td>#006400</td>
</tr>
<tr>
<td>Blue</td>
<td>#0000FF</td>
</tr>
<tr>
<td>Mediumblue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>Darkblue</td>
<td>#00008B</td>
</tr>
</tbody>
</table>

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 1: color_types.ads

```ada
package Color_Types is
  -- Include type declaration for HTML_Color!
  -- type HTML_Color is [...] --

  -- Include function declaration for:
  -- function To_Integer (C : HTML_Color) return Integer;

  -- Include type declaration for Basic_HTML_Color!
  -- type Basic_HTML_Color is [...] --

  -- Include function declaration for:
  -- - Basic_HTML_Color => HTML_Color
  -- function To_HTML_Color [...] end Color_Types;
```

Listing 2: color_types.adb

```ada
package body Color_Types is
  -- Implement the conversion from HTML_Color to Integer here!
  -- function To_Integer (C : HTML_Color) return Integer is
  -- begin
  -- Hint: use 'case' for the HTML colors;
  -- use 16#...# for the hexadecimal values.
  -- end To_Integer;

  -- Implement the conversion from Basic_HTML_Color to HTML_Color here!
  -- function To_HTML_Color [...] is
```

(continues on next page)
```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (HTML_Color_Range,
     HTML_Color_To_Integer,
     Basic_HTML_Color_To_HTML_Color);
  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when HTML_Color_Range =>
        for I in HTML_Color'Range loop
          Put_Line (HTML_Color'Image (I));
        end loop;
      when HTML_Color_To_Integer =>
        for I in HTML_Color'Range loop
          Ada.Integer_Text_IO.Put (Item => To_Integer (I),
                                    Width => 6,
                                    Base => 16);
          New_Line;
        end loop;
      when Basic_HTML_Color_To_HTML_Color =>
        for I in Basic_HTML_Color'Range loop
          Put_Line (HTML_Color'Image (To_HTML_Color (I)));
        end loop;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

4.1. Colors
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.
package Int_Types is
  -- Include type declarations for I_100 and U_100!
  -- type I_100 is [...]
  -- type U_100 is [...]
  --
  function To_I_100 (V : U_100) return I_100;
  function To_U_100 (V : I_100) return U_100;
  -- Include type declarations for D_50 and S_50!
  -- [... D_50 is [...]
  -- [... S_50 is [...]
  --
  function To_D_50 (V : I_100) return D_50;
  function To_S_50 (V : I_100) return S_50;
  function To_I_100 (V : D_50) return I_100;
end Int_Types;

package body Int_Types is
  function To_I_100 (V : U_100) return I_100 is
    begin
      null;
    end To_I_100;
  function To_U_100 (V : I_100) return U_100 is
    begin
      null;
    end To_U_100;
  function To_D_50 (V : I_100) return D_50 is
    Min : constant I_100 := I_100 (D_50'First);
    Max : constant I_100 := I_100 (D_50'Last);
    begin
      null;
    end To_D_50;
  function To_S_50 (V : I_100) return S_50 is
    begin
      null;
    end To_S_50;
end Int_Types;

(continues on next page)
-- Remark: don't forget to verify whether an explicit conversion like
-- $S_{50}$ $(V)$ is needed.
null;
end To_S_50;

function To_I_100 $(V : D_{50})$ return I_100 is
begin
-- Implement the conversion from I_100 to D_{50} here!
-- Remark: don't forget to verify whether an explicit conversion like
-- I_100 $(V)$ is needed.
null;
end To_I_100;
end Int_Types;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Int_Types; use Int_Types;

procedure Main is
  package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
  package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
  package D_50_IO is new Ada.Text_IO.Integer_IO (D_50);

  use I_100_IO;
  use U_100_IO;
  use D_50_IO;

type Test_Case_Index is
  (I_100_Range,
   U_100_Range,
   U_100_Wraparound,
   U_100_To_I_100,
   I_100_To_U_100,
   D_50_Range,
   S_50_Range,
   I_100_To_D_50,
   I_100_To_S_50,
   D_50_To_I_100,
   S_50_To_I_100);

procedure Check $(TC : Test_Case_Index)$ is
begin
  I_100_IO.Default_Width := 1;
  U_100_IO.Default_Width := 1;
  D_50_IO.Default_Width := 1;

  case TC is
    when I_100_Range =>
      Put (I_100'First);
      New_Line;
      Put (I_100'Last);
      New_Line;
    when U_100_Range =>
      -- Continue with remaining cases
  end case;
end Check;
(continued from previous page)

```ada
Put (U_100'First);
New_Line;
Put (U_100'Last);
New_Line;
when U_100'Wraparound =>
  Put (U_100'First - 1);
New_Line;
  Put (U_100'Last + 1);
New_Line;
when U_100'To_I_100 =>
  for I in U_100'Range loop
    I_100_IO.Put (To_I_100 (I));
    New_Line;
  end loop;
when I_100'To_U_100 =>
  for I in I_100'Range loop
    Put (To_U_100 (I));
    New_Line;
  end loop;
when D_50'Range =>
  Put (D_50'First);
New_Line;
  Put (D_50'Last);
New_Line;
when S_50'Range =>
  Put (S_50'First);
New_Line;
  Put (S_50'Last);
New_Line;
when I_100'To_D_50 =>
  for I in I_100'Range loop
    Put (To_D_50 (I));
    New_Line;
  end loop;
when I_100'To_S_50 =>
  for I in I_100'Range loop
    Put (To_S_50 (I));
    New_Line;
  end loop;
when D_50'To_I_100 =>
  for I in D_50'Range loop
    Put (To_I_100 (I));
    New_Line;
  end loop;
when S_50'To_I_100 =>
  for I in S_50'Range loop
    Put (I);
    New_Line;
  end loop;
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  return;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
```

(continues on next page)
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
      \[ K = C + 273.15, \text{ where:} \]
      - \( K \) 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 7: temperature_types.ads

```ada
package Temperature_Types is

  -- Include type declaration for Celsius!
  -- Celsius is [...];
  --   Int_Celsius is [...];

  function To_Celsius (T : Int_Celsius) return Celsius;

  function To_Int_Celsius (T : Celsius) return Int_Celsius;

  -- Include type declaration for Kelvin!
  --   type Kelvin is [...];
  -- Include function declarations for:
  --   - Kelvin => Celsius
  --   - Celsius => Kelvin

end Temperature_Types;
```

Listing 8: temperature_types.adb

```ada
package body Temperature_Types is

  function To_Celsius (T : Int_Celsius) return Celsius is
    begin
      null;
    end To_Celsius;

  function To_Int_Celsius (T : Celsius) return Int_Celsius is
    begin
      null;
    end To_Int_Celsius;

  -- Include function implementation for:
  --   - Kelvin => Celsius
  --   - Celsius => Kelvin

end Temperature_Types;
```

Listing 9: main.adb

```ada
with Ada.Command_Line;  use Ada.Command_Line;
with Ada.Text_IO;       use Ada.Text_IO;
with Temperature_Types; use Temperature_Types;

procedure Main is

  package Celsius_IO is new Ada.Text_IO.Float_IO (Celsius);
  package Kelvin_IO  is new Ada.Text_IO.Float_IO (Kelvin);
```

(continues on next page)
package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);

use Celsius_IO;
use Kelvin_IO;
use Int_Celsius_IO;

type Test_Case_Index is
  (Celsius_Range, Celsius_To_Int_Celsius, Int_Celsius_To_Celsius, Celsius_To_Kelvin);

procedure Check (TC : Test_Case_Index) is
begin
  Celsius_IO.Default_Fore := 1;
  Kelvin_IO.Default_Fore := 1;
  Int_Celsius_IO.Default_Width := 1;

  case TC is
    when Celsius_Range =>
      Put (Celsius'First);
      New_Line;
      Put (Celsius'Last);
      New_Line;
    when Celsius_To_Int_Celsius =>
      Put (To_Int_Celsius (Celsius'First));
      New_Line;
      Put (To_Int_Celsius (0.0));
      New_Line;
      Put (To_Int_Celsius (Celsius'Last));
      New_Line;
    when Int_Celsius_To_Celsius =>
      Put (To_Celsius (Int_Celsius'First));
      New_Line;
      Put (To_Celsius (0));
      New_Line;
      Put (To_Celsius (Int_Celsius'Last));
      New_Line;
    when Kelvin_To_Celsius =>
      Put (To_Celsius (Kelvin'First));
      New_Line;
      Put (To_Celsius (0));
      New_Line;
      Put (To_Celsius (Kelvin'Last));
      New_Line;
    when Celsius_To_Kelvin =>
      Put (To_Kelvin (Celsius'First));
      New_Line;
      Put (To_Kelvin (0));
      New_Line;
      Put (To_Kelvin (Celsius'Last));
      New_Line;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
(continues on next page)
Check (Test_Case_Index'Value (Argument (1)));
end Main;


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 1: directions.ads

```ada
package Directions is

   type Angle_Mod is mod 360;

   type Direction is
      (North,
       Northeast,
       East,
       Southeast,
       South,
       Southwest,
       West,
       Northwest);
```

(continues on next page)
function To_Direction (N: Angle_Mod) return Direction;

-- Include type declaration for Ext_Angle record type:
-- NOTE: Use the Angle_Mod and Direction types declared above!
-- type Ext_Angle is [...]

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);

end Directions;

Listing 2: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

procedure Display (N : Ext_Angle) is
    begin
        -- Uncomment the code below and fill the missing elements
        null;
    end Display;

function To_Direction (N : Angle_Mod) return Direction is
    begin
        case N is
            when 0 => return North;
            when 1 .. 89 => return Northeast;
            when 90 => return East;
            when 91 .. 179 => return Southeast;
            when 180 => return South;
            when 181 .. 269 => return Southwest;
            when 270 => return West;
            when 271 .. 359 => return Northwest;
        end case;
    end To_Direction;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
    begin
        -- Implement the conversion from Angle_Mod to Ext_Angle here!
        null;
    end To_Ext_Angle;

end Directions;

Chapter 5. Records
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:
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 4: color_types.ads

```ada
package Color_Types is
  type HTML_COLOR is
    (Salmon,
     Firebrick,
     Red,
     Darkred,
     Lime,
     Forestgreen,
     Green,
     Darkgreen,
     Blue,
     Mediumblue,
     Darkblue);

  function To_Integer (C : HTML_COLOR) return Integer;

  type Basic_HTML_COLOR is
    (Red,
```
Green,
Blue);

function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;

subtype Int_Color is Integer range 0 .. 255;
-- Replace type declaration for RGB record below
-- - NOTE: Use the Int_Color type declared above!
-- type RGB is [...] 

function To_RGB (C : HTML_Color) return RGB;

function Image (C : RGB) return String;
end Color_Types;

Listing 5: color_types.adb

with Ada.Integer_Text_IO;
package body Color_Types is

function To_Integer (C : HTML_Color) return Integer is
begin
  case C is
    when Salmon => return 16#FA8072#
    when Firebrick => return 16#B22222#
    when Red => return 16#FF0000#
    when Darkred => return 16#8B0000#
    when Lime => return 16#00FF00#
    when Forestgreen => return 16#228B22#
    when Green => return 16#008000#
    when Darkgreen => return 16#006400#
    when Blue => return 16#0000FF#
    when Mediumblue => return 16#0000CD#
    when Darkblue => return 16#00008B#
  end case;
end To_Integer;

function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
begin
  case C is
    when Red => return Red;
    when Green => return Green;
    when Blue => return Blue;
  end case;
end To_HTML_Color;

function To_RGB (C : HTML_Color) return RGB is
begin
  -- Implement the conversion from HTML_Color to RGB here!
  return (null record);
end To_RGB;

(continues on next page)
function Image (C : RGB) return String is
subtype Str_Range is Integer range 1 .. 10;
SR : String (Str_Range);
SG : String (Str_Range);
SB : String (Str_Range);
beginn
    -- Replace argument in the calls to Put below
    -- with the missing elements (red, green, blue)
    -- from the RGB record
    --
    Ada.Integer_Text_IO.Put (To => SR,
        Item => 0, -- REPLACE!
        Base => 16);
    Ada.Integer_Text_IO.Put (To => SG,
        Item => 0, -- REPLACE!
        Base => 16);
    Ada.Integer_Text_IO.Put (To => SB,
        Item => 0, -- REPLACE!
        Base => 16);
    return ("(Red => " & SR & ", Green => " & SG & ", Blue => " & SB & ")");
end Image;
end Color_Types;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
type Test_Case_Index is
   (HTML_Color_To_RGB);
procedure Check (TC : Test_Case_Index) is
begin
    case TC is
    when HTML_Color_To_RGB =>
        for I in HTML_Color'Range loop
            Put_Line (HTML_Color'Image (I) & " => "
                & Image (To_RGB (I)) & ");
        end loop;
    end case;
end Check;
beginn
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
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 7: inventory_pkg.ads

```ada
package Inventory_Pkg is

   type Item_Name is
      (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);

   function To_String (I : Item_Name) return String;

   -- Replace type declaration for Item record:
   --
   type Item is null record;

   function Init (Name : Item_Name;
                  Quantity : Natural;
                  Price : Float) return Item;

   procedure Add (Assets : in out Float;
                  I : Item);

end Inventory_Pkg;
```
**Listing 8: inventory_pkg.adb**

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Inventory_Pkg is

  function To_String (I : Item_Name) return String is
  begin
    case I is
      when Ballpoint_Pen => return "Ballpoint Pen";
      when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
      when Feather_Quill_Pen => return "Feather Quill Pen";
    end case;
  end To_String;

  function Init (Name : Item_Name;
                 Quantity : Natural;
                 Price : Float) return Item is
  begin
    Put_Line (
      "Item: " & To_String (Name) & "." );
    -- Replace return statement with the actual record initialization!
    return (null record);
  end Init;

  procedure Add (Assets : in out Float;
                 I : Item) is
  begin
    -- Implement the function that adds an item to the inventory here!
    --
    null;
  end Add;
end Inventory_Pkg;
```

**Listing 9: main.adb**

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Inventory_Pkg; use Inventory_Pkg;

procedure Main is
  F : array (1 .. 10) of Float := (others => 42.42);

  type Test_Case_Index is
    (Inventory_Chk);

  package Display (Assets : Float) is
    use F_IO is new Ada.Text_IO.Float_IO (Float);
  begin
    Put ("Assets: ");
    Put (Assets, 1, 2, 0);
    New_Line;
  end Display;
```

(continues on next page)
procedure Check (TC : Test_Case_Index) is
  I : Item;
  Assets : Float := 0.0;

-- Please ignore the following three lines!
pragma Warnings (Off, "default initialization");
for Assets'Address use F'Address;
pragma Warnings (On, "default initialization");
begin
  case TC is
    when Inventory_Chk =>
      I := Init (Ballpoint_Pen, 185, 0.15);
      Add (Assets, I);
      Display (Assets);

      I := Init (Oil Based Pen Marker, 100, 9.0);
      Add (Assets, I);
      Display (Assets);

      I := Init (Feather Quill Pen, 2, 40.0);
      Add (Assets, I);
      Display (Assets);
    end case;
  end case;
end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
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.
Introduction to Ada: Laboratories

Listing 1: constrained_arrays.ads

```ada
package Constrained_Arrays is

-- Complete the type and subprogram declarations:

-- type My_Index is [...]

-- type My_Array is [...]

-- function Init...

-- procedure Double...

-- function First_Element...

-- function Last_Element...

-- function Length...

-- A : ...

end Constrained_Arrays;
```

Listing 2: constrained_arrays.adb

```ada
package body Constrained_Arrays is

-- Create the implementation of the subprograms!

end Constrained_Arrays;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Constrained_Arrays; use Constrained_Arrays;

procedure Main is

    type Test_Case_Index is
        (Range_chk,
         Array_range_chk,
         A_Obj.chk,
         Init_chk,
         Doublechk,
         First_elem_chk,
         Last_elem_chk,
         Length_chk);

    procedure Check (TC : Test_Case_Index) is
        AA : My_Array;

    procedure Display (A : My_Array) is
        begin
            for I in A'Range loop
                Put_Line (Integer'image (A (I)));
            end loop;
        end Display;

    procedure Local_Init (A : in out My_Array) is
```

(continues on next page)
begin
A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
end Local_Init;
begin
  case TC is
    when Range_Chek =>
      for I in My_Index loop
        Put_Line (My_Index'Image (I));
      end loop;
    when Array_Range_Chek =>
      for I in My_Array'Range loop
        Put_Line (My_Index'Image (I));
      end loop;
    when A_Obj_Chek =>
      Display (A);
    when Init_Chek =>
      AA := Init;
      Display (AA);
    when Double_Chek =>
      Local_Init (AA);
      Display (AA);
    when First_Elem_Chek =>
      Local_Init (AA);
      Put_Line (Integer'Image (First_Elem (AA)));
    when Last_Elem_Chek =>
      Local_Init (AA);
      Put_Line (Integer'Image (Last_Elem (AA)));
    when Length_Chek =>
      Put_Line (Integer'Image (Length (AA)));
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
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 33)).

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:

<table>
<thead>
<tr>
<th>Color</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salmon</td>
<td>#FA</td>
<td>#80</td>
<td>#72</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#B2</td>
<td>#22</td>
<td>#22</td>
</tr>
<tr>
<td>Red</td>
<td>#FF</td>
<td>#00</td>
<td>#00</td>
</tr>
<tr>
<td>Darkred</td>
<td>#8B</td>
<td>#00</td>
<td>#00</td>
</tr>
<tr>
<td>Lime</td>
<td>#00</td>
<td>#FF</td>
<td>#00</td>
</tr>
<tr>
<td>Forestgreen</td>
<td>#22</td>
<td>#8B</td>
<td>#22</td>
</tr>
<tr>
<td>Green</td>
<td>#00</td>
<td>#80</td>
<td>#00</td>
</tr>
<tr>
<td>Darkgreen</td>
<td>#00</td>
<td>#64</td>
<td>#00</td>
</tr>
<tr>
<td>Blue</td>
<td>#00</td>
<td>#00</td>
<td>#FF</td>
</tr>
<tr>
<td>Mediumblue</td>
<td>#00</td>
<td>#00</td>
<td>#CD</td>
</tr>
<tr>
<td>Darkblue</td>
<td>#00</td>
<td>#00</td>
<td>#8B</td>
</tr>
</tbody>
</table>

Listing 4: color_types.ads

```ada
package Color_Types is

  type HTML_Color is
     (Salmon,
      Firebrick,
      Red,
      Darkred,
      Lime,
      Forestgreen,
      Green,
      Darkgreen,
      Blue,
      Mediumblue,
      Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  type RGB is record
     Red   : Int_Color;
     Green : Int_Color;
     Blue  : Int_Color;

(continues on next page)```
end record;

function To_RGB (C : HTML_Color) return RGB;

function Image (C : RGB) return String;

-- Declare array type for lookup table here:
-- type HTML_Color_RGB is ...

-- Declare lookup table here:
-- To_RGB_Lookup_Table : ...

end Color_Types;

Listing 5: color_types.adb

with Ada.Integer_Text_IO;  
package body Color_Types is  

function To_RGB (C : HTML_Color) return RGB is  
begin  
-- Implement To_RGB using To_RGB_Lookup_Table  
return (0, 0, 0);  

-- Use the code below from the previous version of the To_RGB  
-- function to declare the To_RGB_Lookup_Table:  
--  
-- case C is  
-- when Salmon => return (16#FA#, 16#80#, 16#72#);  
-- when Firebrick => return (16#B2#, 16#22#, 16#22#);  
-- when Red => return (16#FF#, 16#00#, 16#00#);  
-- when Darkred => return (16#8B#, 16#00#, 16#00#);  
-- when Lime => return (16#00#, 16#FF#, 16#00#);  
-- when Forestgreen => return (16#22#, 16#8B#, 16#22#);  
-- when Green => return (16#00#, 16#64#, 16#00#);  
-- when Darkgreen => return (16#00#, 16#64#, 16#00#);  
-- when Blue => return (16#00#, 16#00#, 16#FF#);  
-- when Mediumblue => return (16#00#, 16#00#, 16#CD#);  
-- when Darkblue => return (16#00#, 16#00#, 16#8B#);  
end case;

end To_RGB;

function Image (C : RGB) return String is  
subtype Str_Range is Integer range 1 .. 10;  
SR : String (Str_Range);  
SG : String (Str_Range);  
SB : String (Str_Range);  
begin  
Ada.Integer_Text_IO.Put (To => SR,  
Item => C.Red,  
Base => 16);  
Ada.Integer_Text_IO.Put (To => SG,  
Item => C.Green,  
Base => 16);  
Ada.Integer_Text_IO.Put (To => SB,  
Item => C.Blue,  
Base => 16);
return ("(Red => " & SR  
(continues on next page)
& ", Green => " & SG 
& ", Blue => " & SB 
"});
end Image;
end Color_Types;

---

**Listing 6: main.adb**

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is 
    (Color_Table_Chk, 
     HTML_Color_To_Integer_Chk);

  procedure Check (TC : Test_Case_Index) is 
  begin 
    case TC is 
      when Color_Table_Chk =>
        Put_Line ("Size of HTML_Color_RGB: " 
          & Integer'Image (HTML_Color_RGB'Length));
        Put_Line ("Firebrick: " 
          & Image (To_RGB_Lookup_Table (Firebrick)));
      when HTML_Color_To_Integer_Chk =>
        for I in HTML_Color'Range loop
          Put_Line (HTML_Color'Image (I) & " => " 
            & Image (To_RGB (I)) & ").");
        end loop;
    end case;
  end Check;

  begin 
    if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
    elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
      end if;
    Check (Test_Case_Index'Value (Argument (1)));
  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 ini-
   tialize 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 decla-
      ration.
Listing 7: unconstrained_arrays.ads

```ada
package Unconstrained_Arrays is

-- Complete the type and subprogram declarations:
-- type My_Array is ...;
-- procedure Init ...;

function Init (I, L : Positive) return My_Array;
-- procedure Double ...;
-- function Diff_Prev_Elem ...;

end Unconstrained_Arrays;
```

Listing 8: unconstrained_arrays.adb

```ada
package body Unconstrained_Arrays is

-- Implement the subprograms:
-- procedure Init is...
-- function Init (L : Positive) return My_Array is...
-- procedure Double ... is...
-- function Diff_Prev_Elem ... is...

end Unconstrained_Arrays;
```

Listing 9: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Unconstrained_Arrays; use Unconstrained_Arrays;

procedure Main is
  type Test_Case_Index is
    (Init_Chk,
     Init_Proc_Chk,
     Double_Chk,
     Diff_Prev_Chk,
     Diff_Prev_Single_Chk);

  procedure Check (TC : Test_Case_Index) is
    AA : My_Array (1 .. 5);
    AB : My_Array (5 .. 9);

    procedure Display (A : My_Array) is
      begin
        for I in A'Range loop
          Put_Line (Integer'Image (A (I)));
        end loop;
      end Display;
```

(continues on next page)
procedure Local_Init (A : in out My_Array) is
begin
  A := (1, 2, 5, 10, -10);
end Local_Init;

begin
  case TC is
    when Init_Chk =>
      AA := Init (AA'First, AA'Length);
      AB := Init (AB'First, AB'Length);
      Display (AA);
      Display (AB);
    when Init_Proc_Chk =>
      Init (AA);
      Init (AB);
      Display (AA);
      Display (AB);
    when Double_Chk =>
      Local_Init (AB);
      Double (AB);
      Display (AB);
    when Diff_Prev_Chk =>
      Local_Init (AB);
      AB := Diff_Prev_Elem (AB);
      Display (AB);
    when Diff_Prev_Single_Chk =>
      declare
        A1 : My_Array (1 .. 1) := (1 => 42);
      begin
        A1 := Diff_Prev_Elem (A1);
        Display (A1);
      end;
    end case;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
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 ProductInfos.
   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 \texttt{Integer} type and the Currency
      subtype is based on the \texttt{Float} type, so a conversion is necessary in calculations
      using those types.

Listing 10: product_info_pkg.ads

```ada
package Product_Info_Pkg is

  subtype Quantity is Natural;
  subtype Currency is Float;

  type Product_Info is record
    Units : Quantity;
    Price : Currency;
  end record;

  -- Complete the type declarations:
  -- type Product_Infos is ...
  -- type Currency_Array is ...

  procedure Total (P : Product_Infos;
                  ...);    
```

(continues on next page)
Introduction to Ada: Laboratories

(continued from previous page)

19
20
21
22
23
24
25

function Total (P : Product_Infos) return Currency_Array;
function Total (P : Product_Infos) return Currency;
end Product_Info_Pkg;

Listing 11: product_info_pkg.adb

package body Product_Info_Pkg is

-- Complete the subprogram implementations:
--
-- procedure Total (P : Product_Infos;
-- Tot : out Currency_Array) is ...
-- function Total (P : Product_Infos) return Currency_Array is ...
-- function Total (P : Product_Infos) return Currency is ...
end Product_Info_Pkg;

Listing 12: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Product_Info_Pkg; use Product_Info_Pkg;

procedure Main is
package Currency_IO is new Ada.Text_IO.Float_IO (Currency);

type Test_Case_Index is
(Total_Func_chk, Total_Proc_Chk, Total_Value_Chk);

procedure Check (TC : Test_Case_Index) is
subtype Test_Range is Positive range 1 .. 5;
P : Product_Infos (Test_Range);
Tots : Currency_Array (Test_Range);
Tot : Currency;

procedure Display (Tots : Currency_Array) is
begin
for I in Tots'Range loop
   Currency_IO.Put (Tots (I));
   New_Line;
end loop;
end Display;

procedure Local_Init (P : in out Product_Infos) is
begin
   P := ((1, 0.5),
         (2, 10.0),
         (5, 40.0),
         (10, 10.0),
         ...)
end Local_Init;

(continues on next page)

6.4. Product info 53
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 13: strings_10.ads

```ada
package Strings_10 is
  -- Complete the type and subprogram declarations:
  --
  -- subtype String_10 is ...;
  -- Using "type String_10 is..." is possible, too. However, it
  -- requires a custom Put_Line procedure that is called in Main:
  -- procedure Put_Line (S : String_10);
  -- function To_String_10 ...;

end Strings_10;
```

### Listing 14: strings_10.adb

```ada
package body Strings_10 is
  -- Complete the subprogram declaration and implementation:
  --
  -- function To_String_10 ... is

end Strings_10;
```

### Listing 15: main.adb

```ada
with Ada.Command_Line;  use Ada.Command_Line;
with Ada.Text_IO;       use Ada.Text_IO;
with Strings_10;        use Strings_10;

procedure Main is
  type Test_Case_Index is
    (String_10_Long_Chk,
     String_10_Short_Chk);
  procedure Check (TC : Test_Case_Index) is
    SL   : constant String := "And this is a long string just for testing...";
    SS   : constant String := "Hey!";
    S_10 : String_10;
  begin
    case TC is
      when String_10_Long_Chk =>
        S_10 := To_String_10 (SL);
        Put_Line (String (S_10));
      when String_10_Short_Chk =>
        S_10 := (others => ' ');
        S_10 := To_String_10 (SS);
        Put_Line (String (S_10));
    end case;
  end Check;
```

(continues on next page)
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 Person 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:
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 16: names_ages.ads

```ada
package Names_Ages is

    Max_People : constant Positive := 10;

    subtype Name_Type is String (1 .. 50);

    type Age_Type is new Natural;

    type Person is record
        Name : Name_Type;
        Age  : Age_Type;
    end record;

    -- Add type declaration for People_Array record:
    --
    -- type People_Array is ...;

    -- Replace type declaration for People record. You may use the
    -- following template:
    --
    -- type People is record
    --   People_A : People_Array ...;
    --   Last_Valid : Natural;
    -- end record;
    --

    type People is null record;

    procedure Reset (P : in out People);

    procedure Add (P : in out People;
                  Name : String);

    function Get (P : People;
                  Name : String) return Age_Type;

end Names_Ages;
```

(continues on next page)
### Listing 17: names_ages.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Names_Ages is

  procedure Reset (P : in out People) is
  begin
    null;
  end Reset;

  procedure Add (P : in out People;
                 Name : String) is
  begin
    null;
  end Add;

  function Get (P : People;
                Name : String) return Age_Type is
  begin
    return 0;
  end Get;

  procedure Update (P : in out People;
                   Name : String;
                   Age : Age_Type) is
  begin
    null;
  end Update;

  procedure Display (P : People) is
  begin
    null;
  end Display;

end Names_Ages;
```

### Listing 18: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Names_Ages; use Names_Ages;

procedure Main is
  type Test_Case_Index is
    (Names_Ages_Chk, Get_Age_Chk);

  procedure Check (TC : Test_Case_Index) is
```

(continues on next page)
P : People;

begin
  case TC is
    when Names_Ages_Chk =>
      Reset (P);
      Add (P, "John");
      Add (P, "Patricia");
      Add (P, "Josh");
      Display (P);
      Update (P, "John", 18);
      Update (P, "Patricia", 35);
      Update (P, "Josh", 53);
      Display (P);
    when Get_Age_Chk =>
      Reset (P);
      Add (P, "Peter");
      Update (P, "Peter", 45);
      Put_Line ("Peter is 
        & Age_Type'Image (Get (P, "Peter"))
        & " years old.");
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
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.
Introduction to Ada: Laboratories

Listing 1: aggregates.ads

```ada
package Aggregates is
    -- type Rec is ...;
    -- type Int_Arr is ...;
    procedure Init;
    -- procedure Init_Some ...;
    -- procedure Init ...;
end Aggregates;
```

Listing 2: aggregates.adb

```ada
package body Aggregates is
    procedure Init is null;
end Aggregates;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Aggregates; use Aggregates;

procedure Main is
    -- Remark: the following line is not relevant.
    F : array (1 .. 10) of Float := (others => 42.42)
        with Unreferenced;

    type Test_Case_Index is
        (Default_Rec_Chk,
         Init_Rec_Chk,
         Init_Some_Arr_Chk,
         Init_Arr_Chk);

    procedure Check (TC : Test_Case_Index) is
        A : Int_Arr;
        R : Rec;
        DR : constant Rec := (others => <>);
    begin
        case TC is
            when Default_Rec_Chk =>
                R := DR;
                Put_Line ("Record Default:");
                Put_Line ("W => " & Integer'Image (R.W));
                Put_Line ("X => " & Integer'Image (R.X));
                Put_Line ("Y => " & Integer'Image (R.Y));
                Put_Line ("Z => " & Integer'Image (R.Z));
            when Init_Rec_Chk =>
                Init (R);
                Put_Line ("Record Init:");
                Put_Line ("W => " & Integer'Image (R.W));
                Put_Line ("X => " & Integer'Image (R.X));
                Put_Line ("Y => " & Integer'Image (R.Y));
```

(continues on next page)
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
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 4: versioning.ads

```ada
package Versioning is
   -- type Version is record...
   -- function Convert ...
   -- function Convert
end Versioning;
```

Listing 5: versioning.adb

```ada
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Versioning is

   function Image_Trim (N : Natural) return String is
      S_N : constant String := Trim (Natural'Image (N), Left);
   begin
      return S_N;
   end Image_Trim;

   -- function Convert ...
   --  S_Major : constant String := Image_Trim (V.Major);
   --  S_Minor : constant String := Image_Trim (V.Minor);
   --  S_Maint : constant String := Image_Trim (V.Maintenance);
   -- begin
   --  end Convert;

   -- function Convert ...
   -- begin
   --  end Convert;

end Versioning;
```

Listing 6: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)
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 TO-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 7: todo_lists.ads

```ada
package Todo_Lists is

-- Replace by actual type declaration
type Todo_Item is null record;

-- Replace by actual type declaration
type Todo_Items is null record;

-- Replace by actual type declaration
type Todo_List is null record;

procedure Add (Todos : in out Todo_List;
                Item : String);

procedure Display (Todos : Todo_List);

end Todo_Lists;
```

Listing 8: todo_lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

procedure Add (Todos : in out Todo_List;
                Item : String) is
begin
  Put_Line ("ERROR: list is full!");
  Add;

procedure Display (Todos : Todo_List) is
begin
  null;
end Display;
```

(continues on next page)
end Todo_Lists;

Listing 9: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is
    type Test_Case_Index is (Todo_List_Chk);
    procedure Check (TC : Test_Case_Index) is
        T : Todo_List (10);
    begin
        case TC is
            when Todo_List_Chk =>
                Add (T, "Buy milk");
                Add (T, "Buy tea");
                Add (T, "Buy present");
                Add (T, "Buy tickets");
                Add (T, "Pay electricity bill");
                Add (T, "Schedule dentist appointment");
                Add (T, "Call sister");
                Add (T, "Revise spreadsheet");
                Add (T, "Edit entry page");
                Add (T, "Select new design");
                Add (T, "Create upgrade plan");
                Display (T);
        end case;
    end Check;

    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
        end if;
    Check (Test_Case_Index'Value (Argument (1)));
    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.
2. Implement the Reset procedure.
3. 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:

      ```ada
      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 10: price_lists.ads

```ada
package Price_Lists is
    -- Replace by actual type declaration
type Price_Type is new Float;
    -- Replace by actual type declaration
type Price_List is null record;
    -- Replace by actual type declaration
type Price_Result is null record;
    procedure Reset (Prices : in out Price_List);
    procedure Add (Prices : in out Price_List;
                   Item : Price_Type);
    function Get (Prices : Price_List;
                  Idx : Positive) return Price_Result;
    procedure Display (Prices : Price_List);
end Price_Lists;
```

Listing 11: price_lists.adb

```ada
package body Price_Lists is
    procedure Reset (Prices : in out Price_List) is
        begin
            null;
        end Reset;
    procedure Add (Prices : in out Price_List;
                   Item : Price_Type) is
        begin
            null;
        end Add;
    function Get (Prices : Price_List;
                  Idx : Positive) return Price_Result is
        begin
            null;
        end Get;
    procedure Display (Prices : Price_List) is
        begin
            null;
        end Display;
end Price_Lists;
```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Price_Lists; use Price_Lists;

procedure Main is
  type Test_Case_Index is
    (Price_Type_Chk,
     Price_List_Chk,
     Price_List_Get_Chk);

  procedure Check (TC : Test_Case_Index) is
    L : Price_List (10);
    begin
      Reset (L);
      Add (L, 1.45);
      Add (L, 2.37);
      Add (L, 3.21);
      Add (L, 4.14);
      Add (L, 5.22);
      Add (L, 6.69);
      Add (L, 7.77);
      Add (L, 8.14);
      Add (L, 9.99);
      Add (L, 10.01);
      end Local_Init_List;

  procedure Get_Display (Idx : Positive) is
    R : constant Price_Result := Get (L, Idx);
    begin
      Put_Line ("Attempt Get # " & Positive'Image (Idx));
      if R.Ok then
        Put_Line ("Element # " & Positive'Image (Idx) & " => " & Price_Type'Image (R.Price));
      else
        declare
        begin
          Put_Line ("Element # " & Positive'Image (Idx) & " => " & Price_Type'Image (R.Price));
        exception
          when others =>
            Put_Line ("Element not available (as expected)");
        end;
      end if;
      end Get_Display;
    begin
      case TC is
        when Price_Type_Chk =>
          Put_Line ("The delta value of Price_Type is " & Price_Type'Image (Price_Type'Delta) & ";");
          Put_Line ("The minimum value of Price_Type is " & Price_Type'Image (Price_Type'First) & ";");
          Put_Line ("The maximum value of Price_Type is " & Price_Type'Image (Price_Type'Last) & ";");
        when Price_List_Chk =>
          (continues on next page)
Local_Init_List;
Display (L);

when Price_List_Get_chk =>
Local_Init_List;
Get_Display (5);
Get_Display (40);
end case;
end Check;

begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elsif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;
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 33) 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 1: directions.ads

``` ada
package Directions is

  type Angle_Mod is mod 360;

  type Direction is
    (North, Northwest, West, Southwest, South, Southeast, East);

(continues on next page)```
Introduction to Ada: Laboratories

(continued from previous page)

function To_Direction (N : Angle_Mod) return Direction;

type Ext_Angle is private;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);

private

type Ext_Angle is record
  Angle_Elem : Angle_Mod;
  Direction_Elem : Direction;
end record;

end Directions;

Listing 2: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

  procedure Display (N : Ext_Angle) is
  begin
    Put_Line (
      "Angle: ",
      & Angle_Mod'Image (N.Angle_Elem) & " => " &
      & Direction'Image (N.Direction_Elem) & ".")
  end Display;

  function To_Direction (N : Angle_Mod) return Direction is
  begin
    case N is
    when 0   => return East;
    when 1 .. 89 => return Northwest;
    when 90   => return North;
    when 91 .. 179 => return Northwest;
    when 180   => return West;
    when 181 .. 269 => return Southwest;
    when 270   => return South;
    when 271 .. 359 => return Southeast;
    end case;
  end To_Direction;

  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
  begin
    return (Angle_Elem => N, Direction_Elem => To_Direction (N));
  end To_Ext_Angle;

end Directions;

Listing 3: test_directions.adb

with Directions; use Directions;

procedure Test_Directions is

  (continues on next page)
type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;

All Directions : constant Ext_Angle_Array (1 .. 6) := ((0, East),
          (45, Northwest),
          (90, North),
          (91, North),
          (180, West),
          (270, South));

begin
  for I in All Directions'Range loop
    Display (All Directions (I));
  end loop;
end Test_Directions;

Listing 4: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Test_Case_Index is
    (Direction_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
    when Direction_Chk =>
      Test_Directions;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
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":

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

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

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

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:

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

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.
      - 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:
      - The first one takes another string.
      - 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 5: limited_strings.ads

```ada
package Limited_Strings is

   type Lim_String is limited private;

   function Init (S : String) return Lim_String;
   function Init (Max : Positive) return Lim_String;
   procedure Put_Line (LS : Lim_String);
   procedure Copy (From : Lim_String; To : in out Lim_String);
   function "=" (Ref, Dut : Lim_String) return Boolean;

private

   type Lim_String is access String;

end Limited_Strings;
```

Listing 6: limited_strings.adb

```ada
with Ada.Text_IO;

package body Limited_Strings is

   function Init (S : String) return Lim_String is
      LS : constant Lim_String := new String'(S);
   begin
      return LS;
   end Init;

   function Init (Max : Positive) return Lim_String is
      LS : constant Lim_String := new String'(1 .. Max);
   begin
      LS.all := (others => '_');
      return LS;
   end Init;

   procedure Put_Line (LS : Lim_String) is
   begin
      Ada.Text_IO.Put_Line (LS.all);
   end Put_Line;

   function Get_Min_Last (A, B : Lim_String) return Positive is
      begin
         return Positive'Min (A'Last, B'Last);
      end Get_Min_Last;

   procedure Copy (From : Lim_String; To : in out Lim_String) is
      Min_Last : constant Positive := Get_Min_Last (From, To);
   begin
      -- Complete the implementation!
      null;
   end;

   (continues on next page)
```

8.2. Limited Strings
function "=" (Ref, Dut : Lim_String) return Boolean is
  Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
begin
  -- Complete the implementation!
  return True;
end;
end Limited_Strings;

Listing 7: check_lim_string.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Strings; use Limited_Strings;

procedure Check_Lim_String is
  S : constant String := "----------"
  S1 : constant Lim_String := Init ("Hello World")
  S2 : constant Lim_String := Init (30)
  S3 : Lim_String := Init (5)
  S4 : Lim_String := Init (S & S & S);
begin
  Put ("S1 => ");
  Put_Line (S1);
  Put ("S2 => ");
  Put_Line (S2);
  if S1 = S2 then
    Put_Line ("S1 is equal to S2.");
  else
    Put_Line ("S1 isn't equal to S2.");
  end if;
  Copy (From => S1, To => S3);
  Put ("S3 => ");
  Put_Line (S3);
  if S1 = S3 then
    Put_Line ("S1 is equal to S3.");
  else
    Put_Line ("S1 isn't equal to S3.");
  end if;
  Copy (From => S1, To => S4);
  Put ("S4 => ");
  Put_Line (S4);
  if S1 = S4 then
    Put_Line ("S1 is equal to S4.");
  else
    Put_Line ("S1 isn't equal to S4.");
  end if;
end Check_Lim_String;

Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Check_Lim_String;

procedure Main is
    type Test_Case_Index is
      (Lim_String_Chk);

    procedure Check (TC : Test_Case_Index) is
      begin
        case TC is
          when Lim_String_Chk =>
            Check_Lim_String;
        end case;
      end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    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 33)

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 43)

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 61)

Steps:
1. Change declaration of Price_List type to limited private.

Requirements:
1. Implementation must compile correctly and test cases must pass.
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 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 $T_{\text{Array}}$ using the $T_{\text{Range}}$ and $T_{\text{Element}}$ types;

4. a function $\text{Image}$ that converts a variable of $T_{\text{Element}}$ type to a $\text{String}$.

Listing 1: display_array.ads

```ada
generic
procedure Display_Array (Header : String;
A : T_Array);
```

Listing 2: display_array.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Array (Header : String;
A : T_Array) is
begin
null;
end Display_Array;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Array;

procedure Main is

type Test_Case_Index is (Int_Array_Chk,
Point_Array_Chk);

procedure Test_Int_Array is

type Int_Array is array (Positive range <>) of Integer;

procedure Display_Int_Array is new
Display_Array (T_Range => Positive,
T_Element => Integer,
T_Array => Int_Array,
Image => Integer'Image);

A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
begin
Display_Int_Array ("Integers", A);
end Test_Int_Array;

procedure Test_Point_Array is

type Point is record
X : Float;
Y : Float;
end record;

type Point_Array is array (Natural range <>) of Point;

function Image (P : Point) return String is
begin
return "(" & Float'Image (P.X)
& ", " & Float'Image (P.Y) & ")";
end Image;
```

(continues on next page)
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 4: average.ads

generic
function Average (A : T_Array) return T_Element;

Listing 5: average.adb

function Average (A : T_Array) return T_Element is
begin
    return 0.0;
end Average;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Average;

procedure Main is
    type Test_Case_Index is (Float_Array_Chk, Digits_7_Float_Array_Chk);

    procedure Test_Float_Array is
        type Float_Array is array (Positive range <>) of Float;

        function Average_Float is new Average (T_Range => Positive, T_Element => Float, T_Array => Float_Array);

        A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
        begin
            Put_Line ("Average: " & Float'Image (Average_Float (A)));
        end Test_Float_Array;

    procedure Test_Digits_7_Float_Array is
        type Custom_Float is digits 7 range 0.0 .. 1.0;

        type Float_Array is array (Integer range <>) of Custom_Float;

        function Average_Float is new Average (T_Range => Integer, T_Element => Custom_Float, T_Array => Float_Array);

        A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
        begin
            Put_Line ("Average: " & Custom_Float'Image (Average_Float (A)));
        end Test_Digits_7_Float_Array;

    procedure Check (TC : Test_Case_Index) is
    begin
        case TC is
            when Float_Array_Chk =>
                Test_Float_Array;
            when Digits_7_Float_Array_Chk =>
                Test_Digits_7_Float_Array;
        end case;
    end Check;

(continues on next page)
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`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Fore</th>
<th>Aft</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_Item</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

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 7: average.ads

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

Listing 8: average.adb

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

Listing 9: test_item.ads

```
1  procedure Test_Item;
```
Introduction to Ada: Laboratories

Listing 10: test_item.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

with Average;

procedure Test_Item is
  type Amount is delta 0.01 digits 12;
  type Item is record
    Quantity : Natural;
    Price : Amount;
  end record;

  type Item_Array is
    array (Positive range <>) of Item;
  A : constant Item_Array (1 .. 4)
  := ((Quantity => 5, Price => 10.00),
       (Quantity => 80, Price => 2.50),
       (Quantity => 40, Price => 5.00),
       (Quantity => 20, Price => 12.50));

  begin
    Put ("Average per item & quantity: ");
    F_IO.Put (Average_Total (A));
    New_Line;

    Put ("Average price: ");
    F_IO.Put (Average_Price (A));
    New_Line;

  end Test_Item;
```

Listing 11: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Test_Item;

procedure Main is
  type Test_Case_Index is (Item_Array_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Item_Array_Chk =>
        Test_Item;
    end case;
  end Check;

  begin
    if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
    elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
    end if;

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

9.3. Average of Array of Any Type 87
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 43) labs is an example.
   - In this exercise, you have to abstract those implementations to create the generic Gen_List package.

```ada
Listing 12: gen_list.ads

generic
package Gen_List is

   procedure Init;

   procedure Add (I : Item;
                   Status : out Boolean);

   procedure Display;

end Gen_List;
```

```ada
Listing 13: gen_list.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Gen_List is

   procedure Init is
   begin
      null;
   end Init;

   procedure Add (I : Item;
                  Status : out Boolean) is
   begin
      null;
   end Add;

   procedure Display is
   begin
      null;
   end Display;

end Gen_List;
```

```ada
Listing 14: test_int.ads

procedure Test_Int;
```

```ada
Listing 15: test_int.adb

with Ada.Text_IO; use Ada.Text_IO;

with Gen_List;

procedure Test_Int is

   type Integer_Array is array (Positive range <>) of Integer;
   A : Integer_Array (1 .. 3);
```

9.4. Generic list
L : Natural;
Success : Boolean;

procedure Display_Add_Success (Success : Boolean) is
begin
  if Success then
    Put_Line ("Added item successfully!");
  else
    Put_Line ("Couldn't add item!");
  end if;
end Display_Add_Success;

begin
  Int_List.Init;
  Int_List.Add (2, Success);
  Display_Add_Success (Success);
  Int_List.Add (5, Success);
  Display_Add_Success (Success);
  Int_List.Add (7, Success);
  Display_Add_Success (Success);
  Int_List.Add (8, Success);
  Display_Add_Success (Success);
  Int_List.Display;
end Test_Int;

Listing 16: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Test_Int;

procedure Main is
  type Test_Case_Index is (Int_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Int_Chk =>
        Test_Int;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
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 1: options.ads**

```ada
package Options is
   -- Declare the Option enumeration type!
   type Option is null record;

   function Image (O : Option) return String;

end Options;
```
Introduction to Ada: Laboratories

Listing 2: options.adb

package body Options is

  function Image (O : Option) return String is
  begin
    return "";
  end Image;

end Options;

Listing 3: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Options; use Options;

procedure Main is
  type Test_Case_Index is
    (Options_Chk);

  procedure Check (TC : Test_Case_Index) is

    procedure Check (O : Option) is
    begin
      Put_Line (Image (O));
    exception
      when E : Uninitialized_Value =>
        Put_Line (Exception_Message (E));
    end Check;

    begin
      case TC is
        when Options_Chk =>
          for O in Option loop
            Check (O);
          end loop;
        end case;
      end Check;

      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;

        Check (Test_Case_Index'Value (Argument (1)));
      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 4: tests.ads

```ada
package Tests is

  type Test_ID is (Test_1, Test_2);
  Custom_Exception : exception;

  procedure Num_Exception_Test (ID : Test_ID);

end Tests;
```

### Listing 5: tests.adb

```ada
package body Tests is

  pragma Warnings (Off, "variable ""C"", is assigned but never read");

  procedure Num_Exception_Test (ID : Test_ID) is
    A, B, C : Integer;
    begin
      case ID is
        when Test_1 =>
          A := Integer'Last;
          B := Integer'Last;
          C := A + B;
        when Test_2 =>
          raise Custom_Exception with "Custom_Exception raised!";
      end case;
    end Num_Exception_Test;

  pragma Warnings (On, "variable ""C"", is assigned but never read");

end Tests;
```

### Listing 6: check_exception.adb

```ada
with Tests; use Tests;
```

(continues on next page)
procedure Check_Exception (ID : Test_ID) is
begin
    Num_Exception_Test (ID);
end Check_Exception;

Listing 7: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
    type Test_Case_Index is
        (Exception_1_Chk,
         Exception_2_Chk);
    procedure Check (TC : Test_Case_Index) is
        procedure Check_Handle_Exception (ID : Test_ID) is
            begin
                Check_Exception (ID);
                exception
                    when Constraint_Error =>
                        Put_Line ("Constraint_Error"
                            & " (raised by Check_Exception) detected!");
                    when E : others =>
                        Put_Line (Exception_Name (E)
                            & " (raised by Check_Exception) detected!");
                end Check_Handle_Exception;
            begin
                case TC is
                    when Exception_1_Chk =>
                        Check_Handle_Exception (Test_1);
                    when Exception_2_Chk =>
                        Check_Handle_Exception (Test_2);
                end case;
            end Check;
        begin
            if Argument_Count < 1 then
                Put_Line ("ERROR: missing arguments! Exiting...");
                return;
            elsif Argument_Count > 1 then
                Put_Line ("Ignoring additional arguments...");
            end if;
            Check (Test_Case_Index'Value (Argument (1)));
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 8: tests.ads

```ada
package Tests is

   type Test_ID is (Test_1, Test_2);

   Custom_Exception : exception;

   procedure Num_Exception_Test (ID : Test_ID);

end Tests;
```

Listing 9: tests.adb

```ada
package body Tests is

   pragma Warnings (Off, "variable ""C"" is assigned but never read");

   procedure Num_Exception_Test (ID : Test_ID) is
      A, B, C : Integer;
   begin
      case ID is
         when Test_1 =>
            A := Integer'Last;
            B := Integer'Last;
            C := A + B;
         when Test_2 =>
            raise Custom_Exception with "Custom_Exception raised!";
      end case;
   end Num_Exception_Test;

   pragma Warnings (On, "variable ""C"" is assigned but never read");
```

(continues on next page)
end Tests;

Listing 10: check_exception.ads

with Tests; use Tests;

procedure Check_Exception (ID : Test_ID);

Listing 11: check_exception.adb

procedure Check_Exception (ID : Test_ID) is
begin
  Num_Exception_Test (ID);
end Check_Exception;

Listing 12: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk,
     Exception_2_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Handle_Exception (ID : Test_ID) is
    begin
      Check_Exception (ID);
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error" & " (raised by Check_Exception) detected!");
        when E =>
          Put_Line (Exception_Name (E) & " (raised by Check_Exception) detected!");
    end Check_Handle_Exception;

    begin
      case TC is
        when Exception_1_Chk =>
          Check_Handle_Exception (Test_1);
        when Exception_2_Chk =>
          Check_Handle_Exception (Test_2);
      end case;
    end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments... ");
      end if;

(continues on next page)
43 Check (Test_Case_Index'Value (Argument 1));
45 end Main;
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 1: display_services.ads
```ada
package Display_Services is
end Display_Services;
```

Listing 2: display_services.adb
```ada
package body Display_Services is
end Display_Services;
```

Listing 3: main.adb
```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Services; use Display_Services;
procedure Main is
   type Test_Case_Index is (Display_Service_Chk);
   procedure Check (TC : Test_Case_Index) is
      (continues on next page)```
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 4: event_managers.ads

```ada
package Event_Managers is
end Event_Managers;
```

Listing 5: event_managers.adb

```ada
package body Event_Managers is
  -- Don't forget to display the event ID:
  --
  -- Put_Line("Event #" & Natural'Image (Event_ID));
end Event_Managers;
```

Listing 6: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Event_Managers; use Event_Managers;
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
  type Test_Case_Index is (Event_Manager_Chk);
  procedure Check (TC : Test_Case_Index) is
  begin
    Ev_Mng : array (1 .. 5) of Event_Manager;
    case TC is
      when Event_Manager_Chk =>
        for I in Ev_Mng'Range loop
          Ev_Mng (I).Start (I);
          Ev_Mng (I).Event (Clock + Seconds (5));
        end loop;
        Ev_Mng (2).Event (Clock + Seconds (3));
        Ev_Mng (3).Event (Clock + Seconds (1));
        Ev_Mng (4).Event (Clock + Seconds (2));
        Ev_Mng (5).Event (Clock + Seconds (4));
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
end Main;
```

11.2. Event Manager

---

**Introduction to Ada: Laboratories**

---

```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Event_Managers; use Event_Managers;
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
  type Test_Case_Index is (Event_Manager_Chk);
  procedure Check (TC : Test_Case_Index) is
  begin
    Ev_Mng : array (1 .. 5) of Event_Manager;
    case TC is
      when Event_Manager_Chk =>
        for I in Ev_Mng'Range loop
          Ev_Mng (I).Start (I);
          Ev_Mng (I).Event (Clock + Seconds (5));
        end loop;
        Ev_Mng (2).Event (Clock + Seconds (3));
        Ev_Mng (3).Event (Clock + Seconds (1));
        Ev_Mng (4).Event (Clock + Seconds (2));
        Ev_Mng (5).Event (Clock + Seconds (4));
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line("Ignoring additional arguments...");
    end if;
    Check (Test_Case_Index'Value (Argument (1)));
end Main;
```
11.3 Generic Protected Queue

**Goal:** create a queue container using a protected type.

**Steps:**
1. Implement the generic package Gen_Queue.
   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_Queue:
   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 pro-
   tected type (Queue) as part of a generic package (Gen_Queue).
2. As a bonus exercise, you can analyze the body of the Queue_Tests package and un-
   derstand 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 7: gen_queues.ads
```
package Gen_Queue is
end Gen_Queue;
```

Listing 8: gen_queues.adb
```
package body Gen_Queue is
end Gen_Queue;
```
Listing 9: queue_tests.ads

```ada
package Queue_Tests is
    procedure Simple_Test;
    procedure Concurrent_Test;
end Queue_Tests;
```

Listing 10: queue_tests.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Gen_Queues;

package body Queue_Tests is
    Max : constant := 10;
    type Queue_Mod is mod Max;

    procedure Simple_Test is
        package Queues_Float is new Gen_Queues (Queue_Mod, Float);
        Q_F : Queues_Float.Queue;
        V : Float;
    begin
        V := 10.0;
        while not Q_F.Full loop
            Q_F.Push (V);
            V := V + 1.5;
        end loop;

        while not Q_F.Empty loop
            Q_F.Pop (V);
            Put_Line ("Value from queue: " & Float'Image (V));
        end loop;
    end Simple_Test;

    procedure Concurrent_Test is
        package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
        Q_I : Queues_Integer.Queue;
        task T_Producer;
        task T_Consumer;

        task body T_Producer is
            V : Integer := 100;
        begin
            for I in 1 .. 2 * Max loop
                Q_I.Push (V);
                V := V + 1;
            end loop;
        end T_Producer;

        task body T_Consumer is
            V : Integer;
        begin
            delay 1.5;
        end T_Consumer;
    end Concurrent_Test;
end Queue_Tests;
```

(continues on next page)
while not Q_I.Empty loop
    Q_I.Pop (V);
    Put_Line ("Value from queue: " & Integer'Image (V));
delay 0.2;
end loop;
end T_Consumer;
begin
null;
end Concurrent_Test;
end Queue_Tests;

Listing 11: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Tests; use Queue_Tests;

procedure Main is
    type Test_Case_Index is
        (Simple_Queue_Chk, Concurrent_Queue_Chk);

    procedure Check (TC : Test_Case_Index) is
        begin
            case TC is
                when Simple_Queue_Chk =>
                    Simple_Test;
                when Concurrent_Queue_Chk =>
                    Concurrent_Test;
            end case;
        end Check;

    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
            end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
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:

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

2. Interestingly, you can replace ranges by predicates, which is the goal of this exercise.

Listing 1: prices.ads

```plaintext
package Prices is
  type Amount is delta 10.0 ** (-2) digits 12;
  subtype Price is Amount range 0.0 .. Amount'Last;
end Prices;
```

Listing 2: main.adb

```plaintext
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Prices; use Prices;

procedure Main is
  type Test_Case_Index is
    (Price_Range_Chk);
```

(continues on next page)
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\(^2\) 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 3: triangles.ads

```ada
package Triangles is

    subtype Length is Integer;

end Triangles;
```

\(^2\) [https://en.wikipedia.org/wiki/Pythagorean_theorem](https://en.wikipedia.org/wiki/Pythagorean_theorem)
type Right_Triangle is record
  H : Length := 0;  -- Hypotenuse
  C1, C2 : Length := 0;  -- Catheti / legs
end record;

function Init (H, C1, C2 : Length) return Right_Triangle is
  ((H, C1, C2));
end Triangles;

package Triangles.IO is
  function Image (T : Right_Triangle) return String;
end Triangles.IO;

package body Triangles.IO is
  function Image (T : Right_Triangle) return String is
    "(",
      Length'Image (T.H),
      ", ",
      Length'Image (T.C1),
      ", ",
      Length'Image (T.C2),
    ")";
end Triangles.IO;

procedure Main is
  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk,  
    Triangle_8_6_Fail_Chk,  
    Triangle_10_24_Pass_Chk,  
    Triangle_10_24_Fail_Chk,  
    Triangle_18_24_Pass_Chk,  
    Triangle_18_24_Fail_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
      exception
      when Constraint_Error =>
        
      end Check_Triangle;

    check
      case TC is
      when Triangle_8_6_Pass_Chk => Check_Triangle (8, 6);
      when Triangle_8_6_Fail_Chk => Check_Triangle (6, 8);
      when Triangle_10_24_Pass_Chk => Check_Triangle (10, 24);
      when Triangle_10_24_Fail_Chk => Check_Triangle (24, 10);
      when Triangle_18_24_Pass_Chk => Check_Triangle (18, 24);
      when Triangle_18_24_Fail_Chk => Check_Triangle (24, 18);
      end case;
    end Check;

  begin
    Check (Triangle_8_6_Pass_Chk);
    Check (Triangle_8_6_Fail_Chk);
    Check (Triangle_10_24_Pass_Chk);
    Check (Triangle_10_24_Fail_Chk);
    Check (Triangle_18_24_Pass_Chk);
    Check (Triangle_18_24_Fail_Chk);
  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 7: triangles.ads**

```ada
package Triangles is

  subtype Length is Integer;

  type Right_Triangle is record
    H    : Length := 0;  -- Hypotenuse
    C1, C2 : Length := 0;  -- Catheti / legs
  end record;

end Triangles;
```

(continues on next page)
end record;

function Init (H, C1, C2 : Length) return Right_Triangle is
  ((H, C1, C2));
end Triangles;

Listing 8: triangles-io.ads

package Triangles.IO is
  function Image (T : Right_Triangle) return String;
end Triangles.IO;

package body Triangles.IO is
  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H)
    & ", " & Length'Image (T.C1)
    & ", " & Length'Image (T.C2)
    & ")";
end Triangles.IO;

Listing 9: triangles-io.adb

procedure Main is
  type Test_Case_Index is
    (Triangle_8_6_Pass_CHK, Triangle_8_6_Fail_CHK, Triangle_10_24_Pass_CHK, Triangle_10_24_Fail_CHK, Triangle_18_24_Pass_CHK, Triangle_18_24_Fail_CHK);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
        exception
          when Constraint_Error =>
            Put_Line ("Constraint_Error detected (NOT as expected)."");
          when Assert_Failure =>
            Put_Line ("Assert_Failure detected (as expected)."");
    end Check_Triangle;

(continues on next page)
begin
  case TC is
    when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
    when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
    when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
    when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
    when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
    when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
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 11: triangles.ads

package Triangles is
  subtype Length is Integer;

type Right_Triangle is record
    H : Length := 0;
    -- Hypotenuse
    C1, C2 : Length := 0;
    -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2));
end Triangles;
**Listing 12: triangles-io.ads**

```ada
package Triangles.IO is

  function Image (T : Right_Triangle) return String;

end Triangles.IO;
```

**Listing 13: triangles-io.adb**

```ada
package body Triangles.IO is

  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ")";

end Triangles.IO;
```

**Listing 14: main.adb**

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;

with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk, Triangle_8_6_Fail_Chk, Triangle_10_24_Pass_Chk, Triangle_10_24_Fail_Chk, Triangle_18_24_Pass_Chk, Triangle_18_24_Fail_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
        exception
          when Constraint_Error =>
            Put_Line ("Constraint_Error detected (NOT as expected).");
          when Assert_Failure =>
            Put_Line ("Assert_Failure detected (as expected).");
        end Check_Triangle;
      begin
        case TC is
          when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
          when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
          when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
          when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
          when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
          when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
        end case;
      end begin;
    end Check;

begin
  case TC is
    when Triangle_8_6_Pass_Chek == Check_Triangle (10, 8, 6);
    when Triangle_8_6_Fail_Chek => Check_Triangle (12, 8, 6);
    when Triangle_10_24_Pass_Chek => Check_Triangle (26, 10, 24);
    when Triangle_10_24_Fail_Chek => Check_Triangle (12, 10, 24);
    when Triangle_18_24_Pass_Chek => Check_Triangle (30, 18, 24);
    when Triangle_18_24_Fail_Chek => Check_Triangle (32, 18, 24);
```

(continues on next page)
```ada
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test_Case_Index'Value (Argument (1)));
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 15: triangles.ads

```ada
package Triangles is

  subtype Length is Integer;

  type Right_Triangle is private;

  function Init (H, C1, C2 : Length) return Right_Triangle;

private

  type Right_Triangle is record
    H : Length := 0;

end Right_Triangle;
```

(continues on next page)
-- Hypotenuse
C1, C2 : Length := 0;
-- Catheti / legs

function Init (H, C1, C2 : Length) return Right_Triangle is
((H, C1, C2));

end Triangles;

package Triangles.IO is

function Image (T : Right_Triangle) return String;

end Triangles.IO;

package body Triangles.IO is

function Image (T : Right_Triangle) return String is
("(" & Length’Image (T.H)
& ", " & Length’Image (T.C1)
& ", " & Length’Image (T.C2)
& ")");

end Triangles.IO;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

type Test_Case_Index is
(Triangle 8 6 Pass_Chk,
 Triangle 8 6 Fail_Chk,
 Triangle 10 24 Pass_Chk,
 Triangle 10 24 Fail_Chk,
 Triangle 18 24 Pass_Chk,
 Triangle 18 24 Fail_Chk);

procedure Check (TC : Test_Case_Index) is

procedure Check_Triangle (H, C1, C2 : Length) is
T : Right_Triangle;
begin
T := Init (H, C1, C2);
Put_Line (Image (T));
exception
when Constraint_Error =>
Put_Line ("Constraint_Error detected (NOT as expected)."_constants
when Assert_Failure =>

(continues on next page)
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 43) labs.
2. These are the hexadecimal values of the colors that we used in the original exercise:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salmon</td>
<td>#FA8072</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#B22222</td>
</tr>
<tr>
<td>Red</td>
<td>#FF0000</td>
</tr>
<tr>
<td>Darkred</td>
<td>#8B0000</td>
</tr>
<tr>
<td>Lime</td>
<td>#00FF00</td>
</tr>
<tr>
<td>Forestgreen</td>
<td>#228B22</td>
</tr>
<tr>
<td>Green</td>
<td>#008000</td>
</tr>
<tr>
<td>Darkgreen</td>
<td>#006400</td>
</tr>
<tr>
<td>Blue</td>
<td>#0000FF</td>
</tr>
<tr>
<td>Mediumblue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>Darkblue</td>
<td>#00008B</td>
</tr>
</tbody>
</table>

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 19: color_types.ads

```ada
package Color_Types is

type HTML_Color is
  (Salmon, Firebrick, Red, Darkred, Lime, Forestgreen, Green, Darkgreen, Blue, Mediumblue, Darkblue);

subtype Int_Color is Integer range 0 .. 255;

function Image (I : Int_Color) return String;

type RGB is record
  Red : Int_Color;
  Green : Int_Color;
  Blue : Int_Color;
end record;

function To_RGB (C : HTML_Color) return RGB;
```

(continues on next page)
function Image (C : RGB) return String;

type HTML_Color_RGB_Array is array (HTML_Color) of RGB;

To_RGB_Lookup_Table : constant HTML_Color_RGB_Array :=
(Salmon => (16#FA#, 16#80#, 16#72#),
Firebrick => (16#B2#, 16#22#, 16#22#),
Red => (16#FF#, 16#00#, 16#00#),
Lime => (16#00#, 16#FF#, 16#00#),
Forestgreen => (16#22#, 16#8B#, 16#22#),
Green => (16#00#, 16#80#, 16#00#),
Darkgreen => (16#00#, 16#64#, 16#00#),
Blue => (16#00#, 16#00#, 16#FF#),
Mediumblue => (16#00#, 16#00#, 16#CD#),
Darkblue => (16#00#, 16#00#, 16#8B#));
end Color_Types;

Listing 21: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (HTML_Color_Red_Chk,
     HTML_Color_Green_Chk,
     HTML_Color_Blue_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_HTML_Colors (S : HTML_RGB_Color) is
      begin
        Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
        for I in HTML_Color'Range loop
          Put_Line (HTML_Color'Image (I) & " => " & Image (To_Int_Color (I, S)) & ".");
        end loop;
      end Check_HTML_Colors;
      begin
        case TC is
          when HTML_Color_Red_Chk =>
            Check_HTML_Colors (Red);
          when HTML_Color_Green_Chk =>
            Check_HTML_Colors (Green);
          when HTML_Color_Blue_Chk =>
            Check_HTML_Colors (Blue);
        end case;
      end Check;
      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;
        Check (Test_Case_Index'Value (Argument (1)));
      end Main;

12.6. Primary Color 117
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 1: type_extensions.ads

```ada
package Type_Extensions is
   -- Create declaration of T_Float type!
type T_Float is null record;

   -- function Init ...
   -- function Image ...

   -- Create declaration of T_Mixed type!
type T_Mixed is null record;
end Type_Extensions;
```

Listing 2: type_extensions.adb

```ada
package body Type_Extensions is
   end Type_Extensions;
```

Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Type_Extensions; use Type_Extensions;

procedure Main is
   type Test_Case_Index is
      (Type_Extension_Chk);

   procedure Check (TC : Test_Case_Index) is
      F1, F2 : T_Float;
      M1, M2 : T_Mixed;
      begin
         case TC is
            when Type_Extension_Chk =>
               F1 := Init (2.0);
               F2 := Init (3);
               M1 := Init (4.0);
               M2 := Init (5);
               if M2 in T_Float'Class then
                  Put_Line("T_Mixed is in T_Float'Class as expected");
               end if;
```

(continues on next page)
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:

<table>
<thead>
<tr>
<th>#</th>
<th>Membership status</th>
<th>Start (year)</th>
<th>Discount</th>
<th>Full Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Associate</td>
<td>2010</td>
<td>N/A</td>
<td>250.00</td>
</tr>
<tr>
<td>2</td>
<td>Full</td>
<td>1998</td>
<td>10.0 %</td>
<td>160.00</td>
</tr>
<tr>
<td>3</td>
<td>Full</td>
<td>1987</td>
<td>20.0 %</td>
<td>400.00</td>
</tr>
<tr>
<td>4</td>
<td>Associate</td>
<td>2013</td>
<td>N/A</td>
<td>110.00</td>
</tr>
</tbody>
</table>

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:
**Introduction to Ada: Laboratories**

**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:

<table>
<thead>
<tr>
<th>#</th>
<th>Membership status</th>
<th>Start (year)</th>
<th>Discount</th>
<th>Full Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Associate</td>
<td>2002</td>
<td>N/A</td>
<td>100.00</td>
</tr>
<tr>
<td>2</td>
<td>Full</td>
<td>2005</td>
<td>10.0%</td>
<td>100.00</td>
</tr>
</tbody>
</table>

   - 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 4: online_store.ads**

   ```
   with Ada.Calendar; use Ada.Calendar;
   
   package Online_Store is
   
   type Amount is delta 10.0**(-2) digits 10;
   
   subtype Percentage is Amount range 0.0 .. 1.0;
   
   -- Create declaration of Member type!
   -- You can use Year_Number from Ada.Calendar for the membership starting year.
   --
   type Member is null record;
   
   function Get_Status (M : Member) return String;
   
   function Get_Price (M : Member; P : Amount) return Amount;
   ```

   (continues on next page)
-- Create declaration of Full_Member type!
-- Use the Percentage type for storing the membership discount.

type Full_Member is null record;

function Get_Status (M : Full_Member) return String;

function Get_Price (M : Full_Member;
    P : Amount) return Amount;

end Online_Store;

Listing 5: online_store.adb

package body Online_Store is

function Get_Status (M : Member) return String is
    ("");

function Get_Status (M : Full_Member) return String is
    ("");

function Get_Price (M : Member;
    P : Amount) return Amount is (0.0);

function Get_Price (M : Full_Member;
    P : Amount) return Amount is
    (0.0);

end Online_Store;

Listing 6: online_store-tests.ads

package Online_Store.Tests is

procedure Simple_Test;

end Online_Store.Tests;

Listing 7: online_store-tests.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Online_Store.Tests is

procedure Simple_Test is
    begin
        null;
    end Simple_Test;

end Online_Store.Tests;

Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Online_Store; use Online_Store;
with Online_Store.Tests; use Online_Store.Tests;

procedure Main is

  type Test_Case_Index is
    (Type_Chk,
     Unit_Test_Chk);

  procedure Check (TC : Test_Case_Index) is

    function Result_Image (Result : Boolean) return String is
      (if Result then "OK" else "not OK");

    begin
      case TC is
        when Type_Chk =>
          declare
            AM : constant Member := (Start => 2002);
            FM : constant Full_Member := (Start => 1990,
                                         Discount => 0.2);
          begin
            Put_Line ("Testing Status of Associate Member Type => 
                       & Result_Image (AM.Get_Status = "Associate Member");
            Put_Line ("Testing Status of Full Member Type => 
                       & Result_Image (FM.Get_Status = "Full Member");
            Put_Line ("Testing Discount of Associate Member Type => 
                       & Result_Image (AM.Get_Price (100.0) = 100.0));
            Put_Line ("Testing Discount of Full Member Type => 
                       & Result_Image (FM.Get_Price (100.0) = 80.0));
          end;
          when Unit_Test_Chk =>
            Simple_Test;
      end case;
    end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;

      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
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 61).
   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.

```ada
package Todo_Lists is

  type Todo_Item is access String;

  type Todo_List is null record;

  procedure Add (Todos : in out Todo_List;
                  Item   : String);

  procedure Display (Todos : Todo_List);
```

(continues on next page)
end Todo_Lists;

Listing 2: todo_lists.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

procedure Add (Todos : in out Todo_List; Item : String) is
begin
    null;
end Add;

procedure Display (Todos : Todo_List) is
begin
    Put_Line ("TO-DO LIST");
end Display;

end Todo_Lists;

Listing 3: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is

    type Test_Case_Index is
        (Todo_List_Chk);

    procedure Check (TC : Test_Case_Index) is
        T : Todo_List;
    begin
        case TC is
            when Todo_List_Chk =>
                Add (T, "Buy milk");
                Add (T, "Buy tea");
                Add (T, "Buy present");
                Add (T, "Buy tickets");
                Add (T, "Pay electricity bill");
                Add (T, "Schedule dentist appointment");
                Add (T, "Call sister");
                Add (T, "Revise spreadsheet");
                Add (T, "Edit entry page");
                Add (T, "Select new design");
                Add (T, "Create upgrade plan");
                Display (T);
        end case;
    end Check;

begin
    if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
    end if;

(continues on next page)
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.

```
with Ada.Containers.Ordered_Sets;

package Ops is
  -- type Int_Array is ...
  -- package Integer_Sets is ...
  subtype Int_Set is Integer_Sets.Set;
  function Get_Unique (A : Int_Array) return Int_Set;
  function Get_Unique (A : Int_Array) return Int_Array;
```

(continues on next page)
end Ops;

Listing 5: ops.adb

package body Ops is

  function Get_Unique (A : Int_Array) return Int_Set is
    begin
      null;
    end Get_Unique;

  function Get_Unique (A : Int_Array) return Int_Array is
    begin
      null;
    end Get_Unique;

end Ops;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ops; use Ops;

procedure Main is

  type Test_Case_Index is
    (Get_Unique_Set_Chk,
     Get_Unique_Array_Chk);

  procedure Check (TC : Test_Case_Index; A : Int_Array) is

    procedure Display_Unique_Set (A : Int_Array) is
      S : constant Int_Set := Get_Unique (A);
      begin
        for E of S loop
          Put_Line (Integer'Image (E));
        end loop;
      end Display_Unique_Set;

    procedure Display_Unique_Array (A : Int_Array) is
      AU : constant Int_Array := Get_Unique (A);
      begin
        for E of AU loop
          Put_Line (Integer'Image (E));
        end loop;
      end Display_Unique_Array;

      begin
        case TC is
          when Get_Unique_Set_Chk => Display_Unique_Set (A);
          when Get_Unique_Array_Chk => Display_Unique_Array (A);
        end case;
      end Check;

      begin
        if Argument_Count < 3 then
          Put_Line ("ERROR: missing arguments! Exiting...");
        end if;
      end begin;

return;
else
    declare
        A : Int_Array (1 .. Argument_Count - 1);
    begin
        for I in A'Range loop
            A (I) := Integer'Value (Argument (I + 1));
        end loop;
        Check (Test_Case_Index'Value (Argument (1)), A);
    end if;
end Main;
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$^3$.
   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 1: to_holocene_year.adb

```ada
with Ada.Calendar; use Ada.Calendar;

function To_Holocene_Year (T : Time) return Integer is
begin
  return 0;
end To_Holocene_Year;
```

Listing 2: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;

with To_Holocene_Year;

procedure Main is
  type Test_Case_Index is
    (Holocene_CHK);
  HY : Integer;

  procedure Display_Holocene_Year (Y : Year_Number) is
begin
  (continues on next page)
```

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;

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:

   ```ada
   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 #2
     - Item #1
   - 2019-04-16
     - Item #2
     - Item #3

Listing 3: events.ads

```ada
package Events is
type Event_Item is null record;
type Event_Items is null record;
end Events;
```
Listing 4: events-lists.ads

with Ada.Calendar; use Ada.Calendar;

package Events.Lists is

  type Event_List is tagged private;

  procedure Add (Events : in out Event_List;
                  Event_Time : Time;
                  Event : String);

  procedure Display (Events : Event_List);

private

  type Event_List is tagged null record;

end Events.Lists;

Listing 5: events-lists.adb

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

package body Events.Lists is

  procedure Add (Events : in out Event_List;
                 Event_Time : Time;
                 Event : String) is
  begin
    null;
  end Add;

  function Date_Image (T : Time) return String is
    Date_Img : constant String := Image (T);
  begin
    return Date_Img (1 .. 10);
  end;

  procedure Display (Events : Event_List) is
  begin
    T : Time;
    --- You should use Date_Image (T) here!
  end Display;

end Events.Lists;

Listing 6: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Events.Lists; use Events.Lists;

procedure Main is
  type Test_Case_Index is
    (Event_List_Chk);
  (continues on next page)
procedure Check (TC : Test_Case_Index) is
  EL : Event_List;
begin
  case TC is
    when Event_List_Chk =>
      EL.Add (Time_Of (2018, 2, 16),
             "Final check");
      EL.Add (Time_Of (2018, 2, 16),
             "Release");
      EL.Add (Time_Of (2018, 12, 3),
             "Brother's birthday");
      EL.Add (Time_Of (2018, 1, 1),
             "New Year's Day");
      EL.Display;
    end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments..."); end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
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 1: str_concat.ads**

```ada

package Str_Concat is

   type Unbounded_Strings is array (Positive range <>) of Unbounded_String;

function Concat (USA : Unbounded_Strings;
                  Trim_Str : Boolean;
                  Add_Whitespace : Boolean) return Unbounded_String;

function Concat (USA : Unbounded_Strings;
                  Trim_Str : Boolean;
                  Add_Whitespace : Boolean) return String;
```

(continues on next page)
with Ada.Strings; use Ada.Strings;

package body Str_Concat is

function Concat (USA : Unbounded_Strings;
Trim_Str : Boolean;
Add_Whitespace : Boolean)
return Unbounded_String is
begin
return "";
end Concat;

function Concat (USA : Unbounded_Strings;
Trim_Str : Boolean;
Add_Whitespace : Boolean)
return String is
begin
return "";
end Concat;

end Str_Concat;

Listing 2: str_concat.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Str_Concat; use Str_Concat;

procedure Main is

type Test_Case_Index is
(Unbounded_Concat_No_Trim_No_WS_Chk,
Unbounded_Concat_Trim_No_WS_Chk,
String_Concat_Trim_WS_Chk,
Concat_Single_Element);

procedure Check (TC : Test_Case_Index) is
begin
case TC is
when Unbounded_Concat_No_Trim_No_WS_Chk =>
declare
S : constant Unbounded_Strings := (
To_Unbounded_String ("Hello"),
To_Unbounded_String (" World"),
To_Unbounded_String ("!"));
begin
Put_Line (To_String (Concat (S, False, False)));
end;
when Unbounded_Concat_Trim_No_WS_Chk =>
declare
S : constant Unbounded_Strings := (
To_Unbounded_String (" This "),
To_Unbounded_String (" is "),
To_Unbounded_String (" a "),
To_Unbounded_String (" check "));
begin
(continues on next page)
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 133)) as a starting point.

Listing 4: events.ads

```ada
with Ada.Containers.Vectors;

package Events is

-- subtype Event_Item is

package Event_Item_Collectors is new
Ada.Containers.Vectors
(Index_Type => Positive,
Element_Type => Event_Item);

subtype Event_Items is Event_Item_Collectors.Vector;

end Events;
```

Listing 5: events-lists.ads

```ada
with Ada.Calendar;
use Ada.Calendar;

with Ada.Containers.Ordered_Maps;

package Events.Lists is

type Event_List is tagged private;

procedure Add (Events : in out Event_List;
Event_Time : Time;
Event : String);

procedure Display (Events : Event_List);

private

package Event_Time_Item_Collections is new
Ada.Containers.Ordered_Maps
(Key_Type => Time,
Element_Type => Event_Items,
"=" => Event_Item_Collectors."=");

type Event_List is new Event_Time_Item_Collections.Map with null record;

end Events.Lists;
```

Listing 6: events-lists.adb

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

package body Events.Lists is

procedure Add (Events : in out Event_List;
Event_Time : Time;
Event : String) is
use Event_Item_Collectors;
E : constant Event_Item := new String'(Event);
begin
if not Events.Contains (Event_Time) then
Events.Include (Event_Time, Empty_Vector);
end if;
end Add;
```

(continues on next page)
end if;
    Events (Event_Time).Append (E);
end Add;

function Date_Image (T : Time) return String is
begin
    return Date_Img (1 .. 10);
end;

procedure Display (Events : Event_List) is
begin
    for C in Events.Iterate loop
        T := Key (C);
        Put_Line ("- " & Date_Image (T));
        for I of Events (C) loop
            Put_Line (" - " & I.all);
        end loop;
    end loop;
end Display;

end Events.Lists;

Listing 7: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Events;
with Events.Lists; use Events.Lists;

procedure Main is
    type Test_Case_Index is
        (Unbounded_String_Chk, Event_List_Chk);

    procedure Check (TC : Test_Case_Index) is
begin
    case TC is
        when Unbounded_String_Chk =>
            declare
                S : constant Events.Event_Item := To_Unbounded_String ("Checked");
            begin
                Put_Line (To_String (S));
            end;
        when Event_List_Chk =>
            EL.Add (Time_Of (2018, 2, 16), "Final check");
            EL.Add (Time_Of (2018, 2, 16), "Release");
            EL.Add (Time_Of (2018, 12, 3), "Brother's birthday");
            EL.Add (Time_Of (2018, 1, 1), "New Year's Day");
    end case;
end Check;

...
34   end case;
35
36   end Check;

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
47   end Main;
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 \( 20 \times \log_{10}(F) \), 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\(^4\) 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 1: decibels.ads

```ada
package Decibels is

  subtype Decibel is Float;
  subtype Factor is Float;

  function To_Decibel (F : Factor) return Decibel;
  function To_Factor (D : Decibel) return Factor;

end Decibels;
```

\(^4\) https://en.wikipedia.org/wiki/Decibel
### Listing 2: decibels.adb

```ada
package body Decibels is

  function To_Decibel (F : Factor) return Decibel is
  begin
    return 0.0;
  end To_Decibel;

  function To_Factor (D : Decibel) return Factor is
  begin
    return 0.0;
  end To_Factor;

end Decibels;
```

### Listing 3: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Decibels; use Decibels;

procedure Main is
  type Test_Case_Index is
    (Db_Chk,
     Factor_Chk);

  procedure Check (TC : Test_Case_Index; V : Float) is
    package F_IO is new Ada.Text_IO.Float_IO (Factor);
    package D_IO is new Ada.Text_IO.Float_IO (Decibel);

    procedure Put_Decibel_Cnvt (D : Decibel) is
      F : constant Factor := To_Factor (D);
    begin
      D_IO.Put (D, 0, 2, 0);
      F_IO.Put (F, 0, 2, 0);
      New_Line;
    end;

    procedure Put_Factor_Cnvt (F : Factor) is
      D : constant Decibel := To_Decibel (F);
    begin
      Put ("Factor of ");
      F_IO.Put (F, 0, 2, 0);
      D_IO.Put (D, 0, 2, 0);
      Put_Line (" dB");
    end;

    begin
      case TC is
        when Db_Chk =>
          Put_Decibel_Cnvt (Decibel (V));
        when Factor_Chk =>
          Put_Factor_Cnvt (Factor (V));
      end case;
    end Check;

begin

(continues on next page)
if Argument_Count < 2 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
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\(^5\) (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 4: signals.ads

```ada
package Signals is
    subtype Sig_Value is Float;
    type Signal is array (Natural range <>) of Sig_Value;
    function Rms (S : Signal) return Sig_Value;
end Signals;
```

---

\(^5\) https://en.wikipedia.org/wiki/Root_mean_square

package body Signals is

function Rms (S : Signal) return Sig_Value is
begin
  return 0.0;
end;
end Signals;

package Signals.Std is

Sample_Rate : Float := 8000.0;

function Generate_Sine (N : Positive; Freq : Float) return Signal;
function Generate_Square (N : Positive) return Signal;
function Generate_Triangular (N : Positive) return Signal;
end Signals.Std;

package body Signals.Std is

function Generate_Sine (N : Positive; Freq : Float) return Signal is
  S : Signal (0 .. N - 1);
begin
  for I in S'First .. S'Last loop
    S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
  end loop;

  return S;
end;

function Generate_Square (N : Positive) return Signal is
  S : constant Signal (0 .. N - 1) := (others => 1.0);
begin
  return S;
end;

function Generate_Triangular (N : Positive) return Signal is
  S : Signal (0 .. N - 1);
  S_Half : constant Natural := S'Last / 2;
begin
  for I in S'First .. S_Half loop
    S (I) := 1.0 * (Float (I) / Float (S_Half));
  end loop;

  for I in S_Half .. S'Last loop
    S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
  end loop;
end;
return S;
end;
end Signals.Std;

Listing 8: main.adb

with Ada.Command_Line;  use Ada.Command_Line;
with Ada.Text_IO;  use Ada.Text_IO;
with Signals;  use Signals;
with Signals.Std;  use Signals.Std;

procedure Main is
  type Test_Case_Index is
    (Sine_Signal_Chk,
     Square_Signal_Chk,
     Triangular_Signal_Chk);

  procedure Check (TC : Test_Case_Index) is
    package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
    N : constant Positive := 1024;
    S_Si : constant Signal := Generate_Sine (N, 440.0);
    S_Sq : constant Signal := Generate_Square (N);
    S_Tr : constant Signal := Generate_Triangular (N + 1);
    begin
      case TC is
        when Sine_Signal_Chk =>
          Put ("RMS of Sine Signal:");
          Sig_IO.Put (Rms (S_Si), 0, 2, 0);
          New_Line;
        when Square_Signal_Chk =>
          Put ("RMS of Square Signal:");
          Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
          New_Line;
        when Triangular_Signal_Chk =>
          Put ("RMS of Triangular Signal:");
          Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
          New_Line;
      end case;
    end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;

17.2. Root-Mean-Square
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.
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 the following points / positions:

<table>
<thead>
<tr>
<th>Point #1</th>
<th>( 1.0, 0.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point #2</td>
<td>( 0.8, 1.0)</td>
</tr>
<tr>
<td>Point #3</td>
<td>( -1.0, 0.0)</td>
</tr>
<tr>
<td>Point #4</td>
<td>( 0.8, -1.0)</td>
</tr>
<tr>
<td>Point #5</td>
<td>( 1.0, 0.0)</td>
</tr>
</tbody>
</table>

   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:

<table>
<thead>
<tr>
<th>Point #1</th>
<th>0.00 degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point #2</td>
<td>90.00 degrees</td>
</tr>
<tr>
<td>Point #3</td>
<td>180.00 degrees</td>
</tr>
<tr>
<td>Point #4</td>
<td>-90.00 degrees (= 270 degrees)</td>
</tr>
<tr>
<td>Point #5</td>
<td>0.00 degrees</td>
</tr>
</tbody>
</table>
1. To rotate a complex number simply multiply it by a unit vector whose arg is the radian angle to be rotated: \( Z = e^{i\theta} \)

**Listing 9: rotation.ads**
```
with Ada.Numerics.Complex_Types;
use Ada.Numerics.Complex_Types;

package Rotation is
  type Complex_Points is array (Positive range <>) of Complex;
  function Rotation (N : Positive) return Complex_Points;
end Rotation;
```

**Listing 10: rotation.adb**
```
with Ada.Numerics; use Ada.Numerics;

package body Rotation is
  function Rotation (N : Positive) return Complex_Points is
    C : Complex_Points (1 .. 1) := (others => (0.0, 0.0));
  begin
    return C;
  end;
end Rotation;
```

**Listing 11: angles.ads**
```
with Rotation; use Rotation;

package Angles is
  subtype Angle is Float;
  type Angles is array (Positive range <>) of Angle;
  function To_Angles (C : Complex_Points) return Angles;
end Angles;
```

**Listing 12: angles.adb**
```
with Ada.Numerics; use Ada.Numerics;

package body Angles is
  function To_Angles (C : Complex_Points) return Angles is
  begin
    return A : Angles (C'Range) do
      for I in A'Range loop
        A (I) := Argument (C (I)) / Pi * 180.0;
      end loop;
    end return;
end To_Angles;
end Angles;
```
Introduction to Ada: Laboratories

**Listing 13: rotation-tests.ads**

```ada
package Rotation.Tests is

  procedure Test_Rotation (N : Positive);

  procedure Test_Angles (N : Positive);

end Rotation.Tests;
```

**Listing 14: rotation-tests.adb**

```ada
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Text_IO.Complex_IO; use Ada.Text_IO.Complex_IO (Complex_Types);
with Ada.Numerics;  use Ada.Numerics;
with Angles;  use Angles;

package body Rotation.Tests is

  package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
  package F_IO is new Ada.Text_IO.Float_IO (Float);

  -- Adapt value due to floating-point inaccuracies
  --

  function Adapt (C : Complex) return Complex is
    function Check_Zero (F : Float) return Float is
      (if F < 0.0 and F >= -0.01 then 0.0 else F);
    begin
      return C_Out : Complex := C do
        C_Out.Re := Check_Zero (C_Out.Re);
        C_Out.Im := Check_Zero (C_Out.Im);
      end return;
    end Adap;

  function Adapt (A : Angle) return Angle is
    (if A <= -179.99 and A >= -180.01 then 180.0 else A);
  begin
    procedure Test_Rotation (N : Positive) is
      C : constant Complex_Points := Rotation (N);
    begin
      Put_Line ("---- Points for " & Positive’Image (N) & " slices ----");
      for V of C loop
        Put ("Point: ");
        C_IO.Put (Adapt (V), 0, 1, 0);
        New_Line;
      end loop;
    end Test_Rotation;

    procedure Test_Angles (N : Positive) is
      C : constant Complex_Points := Rotation (N);
      A : constant Angles.Angles := To_Angles (C);
    begin
      Put_Line ("---- Angles for " & Positive’Image (N) & " slices ----");
      for V of A loop
        Put ("Angle: ");
        F_IO.Put (Adapt (V), 0, 2, 0);
        Put_Line (" degrees");
      end loop;
  end Test_Rotation;

(continues on next page)
```
end Test_Angles;
end Rotation.Tests;

Listing 15: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Rotation.Tests; use Rotation.Tests;

procedure Main is
  type Test_Case_Index is
    (Rotation_Chk, Angles_Chk);
  procedure Check (TC : Test_Case_Index; N : Positive) is
    begin
      case TC is
        when Rotation_Chk =>
          Test_Rotation (N);
        when Angles_Chk =>
          Test_Angles (N);
      end case;
    end Check;

begin
  if Argument_Count < 2 then
    Put_Line ("ERROR: missing arguments! Exiting..." );
    return;
  elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments..." );
  end if;

  Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
end Main;
18.1 Imperative Language

18.1.1 Hello World

Listing 1: main.adb

``` ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
  Put_Line ("Hello World!");
end Main;
```

18.1.2 Greetings

Listing 2: main.adb

``` ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  procedure Greet (Name : String) is
  begin
    Put_Line ("Hello " & Name & "!");
  end Greet;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Greet (Argument (1));
end Main;
```
18.1.3 Positive Or Negative

Listing 3: classify_number.ads

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

Listing 4: classify_number.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Classify_Number (X : Integer) is
begin
  if X > 0 then
    Put_Line ("Positive");
  elsif X < 0 then
    Put_Line ("Negative");
  else
    Put_Line ("Zero");
  end if;
end Classify_Number;
```

Listing 5: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Classify_Number;

procedure Main is
  A : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  A := Integer'Value (Argument (1));
  Classify_Number (A);
end Main;
```

18.1.4 Numbers

Listing 6: display_numbers.ads

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

Listing 7: display_numbers.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Display_Numbers (A, B : Integer) is
  X, Y : Integer;
begin
  if A <= B then
    X := A;
  else
    X := B;
  end if;
end Display_Numbers;
```
8
9  Y := B;
10  else
11  X := B;
12  Y := A;
13  end if;
14  for I in X .. Y loop
15  Put_Line (Integer'Image (I));
16  end loop;
17 end Display_Numbers;

Listing 8: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Numbers;

procedure Main is
  A, B : Integer;
begin
  if Argument_Count < 2 then
    Put_Line ("ERROR: missing arguments! Exiting..."就餐);
    return;
  elsif Argument_Count > 2 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  A := Integer'Value (Argument (1));
  B := Integer'Value (Argument (2));
  Display_Numbers (A, B);
end Main;

18.2 Subprograms

18.2.1 Subtract Procedure

Listing 9: subtract.ads

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

Listing 10: subtract.adb

begin
  Result := A - B;
end Subtract;

Listing 11: main.adb

(with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
(continues on next page)
with Subtract;

procedure Main is
  type Test_Case_Index is
    (Sub_10_1_Chk, Sub_10_100_Chk, Sub_0_5_Chk, Sub_0_Minus_5_Chk);

  procedure Check (TC : Test_Case_Index) is
    Result : Integer;
  begin
    case TC is
      when Sub_10_1_Chk =>
        Subtract (10, 1, Result);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_10_100_Chk =>
        Subtract (10, 100, Result);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_0_5_Chk =>
        Subtract (0, 5, Result);
        Put_Line ("Result: " & Integer'Image (Result));
      when Sub_0_Minus_5_Chk =>
        Subtract (0, -5, Result);
        Put_Line ("Result: " & Integer'Image (Result));
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.2.2 Subtract Function

Listing 12: subtract.ads

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

Listing 13: subtract.adb

function Subtract (A, B : Integer) return Integer is
begin
  return A - B;
end Subtract;

Listing 14: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Subtract;
procedure Main is
    type Test_Case_Index is
        (Sub_10_1_Check, Sub_10_100_Check, Sub_0_5_Check, Sub_0_Minus_5_Check);

    procedure Check (TC : Test_Case_Index) is
        Result : Integer;
    begin
        case TC is
            when Sub_10_1_Check =>
                Result := Subtract (10, 1);
                Put_Line ("Result: " & Integer'Image (Result));
            when Sub_10_100_Check =>
                Result := Subtract (10, 100);
                Put_Line ("Result: " & Integer'Image (Result));
            when Sub_0_5_Check =>
                Result := Subtract (0, 5);
                Put_Line ("Result: " & Integer'Image (Result));
            when Sub_0_Minus_5_Check =>
                Result := Subtract (0, -5);
                Put_Line ("Result: " & Integer'Image (Result));
        end case;
    end Check;

    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
        end if;
    Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.2.3 Equality function

Listing 15: is_equal.ads
function Is_Equal (A, B : Integer) return Boolean;

Listing 16: is_equal.adb
function Is_Equal (A, B : Integer) return Boolean is
    begin
        return A = B;
    end Is_Equal;

Listing 17: main.adb
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Is_Equal;

18.2. Subprograms
procedure Main is
  type Test_Case_Index is
    (Equal_Chk,
     Inequal_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Display_Equal (A, B : Integer;
                             Equal : Boolean) is
      begin
        Put (Integer'Image (A));
        if Equal then
          Put (" is equal to ");
        else
          Put (" isn't equal to ");
        end if;
        Put_Line (Integer'Image (B) & ".");
      end Display_Equal;
      Result : Boolean;
      begin
        case TC is
          when Equal_Chk =>
            for I in 0 .. 10 loop
              Result := Is_Equal (I, I);
              Display_Equal (I, I, Result);
            end loop;
          when Inequal_Chk =>
            for I in 0 .. 10 loop
              Result := Is_Equal (I, I - 1);
              Display_Equal (I, I - 1, Result);
            end loop;
          end case;
        end Check;
      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;

18.2.4 States

Listing 18: display_state.ads

procedure Display_State (State : Integer);

Listing 19: display_state.adb

with Ada.Text_IO; use Ada.Text_IO;

procedure Display_State (State : Integer) is
begin
case State is
  when 0 =>
    Put_Line ("Off");
  when 1 =>
    Put_Line ("On: Simple Processing");
  when 2 =>
    Put_Line ("On: Advanced Processing");
  when others =>
    null;
end case;
end Display_State;

Listing 20: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_State;

procedure Main is
  State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  State := Integer'Value (Argument (1));
  Display_State (State);
end Main;

18.2.5 States #2

Listing 21: get_state.ads

function Get_State (State : Integer) return String;

Listing 22: get_state.adb

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

Listing 23: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Get_State;
**18.2.6 States #3**

Listing 24: is_on.ads

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

Listing 25: is_on.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Is_On;

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

Listing 26: display_on_off.ads

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

Listing 27: display_on_off.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Is_On;

procedure Display_On_Off (State : Integer) is
begin
  Put_Line (if Is_On (State) then "On" else "Off");
end Display_On_Off;
```

Listing 28: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_On_Off; with Is_On;

procedure Main is
State : Integer;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;

  State := Integer'Value (Argument (1));
  Put_Line (Get_State (State));
end Main;
```
18.2.7 States #4

Listing 29: set_next.ads

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

Listing 30: set_next.adb

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

Listing 31: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Set_Next;

procedure Main is
    State : Integer;
    begin
        if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
        elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
        end if;
        State := Integer'Value (Argument (1));
        Set_Next (State);
        Put_Line (Integer'Image (State));
end Main;
```

18.2. Subprograms
18.3 Modular Programming

18.3.1 Months

Listing 32: months.ads

```
package Months is

    Jan : constant String := "January";
    Feb : constant String := "February";
    Mar : constant String := "March";
    Apr : constant String := "April";
    May : constant String := "May";
    Jun : constant String := "June";
    Jul : constant String := "July";
    Aug : constant String := "August";
    Sep : constant String := "September";
    Oct : constant String := "October";
    Nov : constant String := "November";
    Dec : constant String := "December";

    procedure Display_Months;

end Months;
```

Listing 33: months.adb

```
with Ada.Text_IO; use Ada.Text_IO;

package body Months is

    procedure Display_Months is
        begin
            Put_Line ("Months:");
            Put_Line ("- " & Jan);
            Put_Line ("- " & Feb);
            Put_Line ("- " & Mar);
            Put_Line ("- " & Apr);
            Put_Line ("- " & May);
            Put_Line ("- " & Jun);
            Put_Line ("- " & Jul);
            Put_Line ("- " & Aug);
            Put_Line ("- " & Sep);
            Put_Line ("- " & Oct);
            Put_Line ("- " & Nov);
            Put_Line ("- " & Dec);
        end Display_Months;

end Months;
```

Listing 34: main.adb

```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Months; use Months;

procedure Main is

    type Test_Case_Index is
```

(continues on next page)
18.3.2 Operations

Listing 35: operations.ads

```ada
package Operations is

  function Add (A, B : Integer) return Integer;
  function Subtract (A, B : Integer) return Integer;
  function Multiply (A, B : Integer) return Integer;
  function Divide (A, B : Integer) return Integer;

end Operations;
```

Listing 36: operations.adb

```ada
package body Operations is

  function Add (A, B : Integer) return Integer is
    begin
      return A + B;
    end Add;

  function Subtract (A, B : Integer) return Integer is
    begin
      return A - B;
    end Subtract;

  function Multiply (A, B : Integer) return Integer is
    begin
      return A * B;
    end Multiply;

  function Divide (A, B : Integer) return Integer is
    begin
      return A / B;
    end Divide;

end Operations;
```

(continues on next page)
return A / B;
end Divide;
end Operations;

Listing 37: operations-test.ads

package Operations.Test is
  procedure Display (A, B : Integer);
end Operations.Test;

Listing 38: operations-test.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Operations.Test is
  procedure Display (A, B : Integer) is
    A_Str : constant String := Integer'Image (A);
    B_Str : constant String := Integer'Image (B);
    begin
      Put_Line ("Operations:");
      Put_Line (A_Str & " + " & B_Str & " = " & Integer'Image (Add (A, B)) & ",");
      Put_Line (A_Str & " - " & B_Str & " = " & Integer'Image (Subtract (A, B)) & ",");
      Put_Line (A_Str & " * " & B_Str & " = " & Integer'Image (Multiply (A, B)) & ",");
      Put_Line (A_Str & " / " & B_Str & " = " & Integer'Image (Divide (A, B)) & ",");
    end Display;
end Operations.Test;

Listing 39: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Operations; use Operations.Test;

procedure Main is
  type Test_Case_Index is
    (Operations_Chk, Operations_Display_Chk);
  procedure Check ( TC : Test_Case_Index ) is
    begin
      case TC is
        when Operations_Chk =>
          Put_Line ("Add (100, 2) = 
            & Integer'Image (Operations.Add (100, 2))");
(continues on next page)
18.4 Strongly typed language

18.4.1 Colors

Listing 40: color_types.ads

package Color_Types is

   type HTML_Color is
      (Salmon,
       Firebrick,
       Red,
       Darkred,
       Lime,
       Forestgreen,
       Green,
       Darkgreen,
       Blue,
       Mediumblue,
       Darkblue);

   function To_Integer (C : HTML_Color) return Integer;

   type Basic_HTML_Color is
      (Red,
       Green,
       Blue);

   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;

end Color_Types;
package body Color_Types is

  function To_Integer (C : HTML_Color) return Integer is
  begin
    case C is
      when Salmon => return 16#FA8072#;
      when Firebrick => return 16#B22222#;
      when Red => return 16#FF0000#;
      when Darkred => return 16#8B0000#;
      when Lime => return 16#00FF00#;
      when Forestgreen => return 16#228B22#;
      when Green => return 16#008000#;
      when Darkgreen => return 16#006400#;
      when Blue => return 16#0000FF#;
      when Mediumblue => return 16#0000CD#;
      when Darkblue => return 16#00008B#;
      end case;
  end To_Integer;

  function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
  begin
    case C is
      when Red => return Red;
      when Green => return Green;
      when Blue => return Blue;
      end case;
  end To_HTML_Color;

end Color_Types;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (HTML_Color_Range, 
     HTML_Color_To_Integer, 
     Basic_HTML_Color_To_HTML_Color);

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when HTML_Color_Range =>
        for I in HTML_Color'Range loop 
          Put_Line (HTML_Color'Image (I));
        end loop;
      when HTML_Color_To_Integer =>
        for I in HTML_Color'Range loop 
          Ada.Integer_Text_IO.Put (Item => To_Integer (I), 
                                    Width => 1, 
                                    Base => 16);
          New_Line;
        end loop;
    end case;
  end Check;
end Main;
when Basic_HTML_Color_To_HTML_Color =>
  for I in Basic_HTML_Color'Range loop
    Put_Line (HTML_Color'Image (To_HTML_Color (I)));
  end loop;
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.4.2 Integers

Listing 43: int_types.ads

package Int_Types is

  type I_100 is range 0 .. 100;
  type U_100 is mod 101;
  function To_I_100 (V : U_100) return I_100;
  function To_U_100 (V : I_100) return U_100;

  type D_50 is new I_100 range 10 .. 50;
  subtype S_50 is I_100 range 10 .. 50;
  function To_D_50 (V : I_100) return D_50;
  function To_S_50 (V : I_100) return S_50;
  function To_I_100 (V : D_50) return I_100;

end Int_Types;

Listing 44: int_types.adb

package body Int_Types is

  function To_I_100 (V : U_100) return I_100 is
    begin
      return I_100 (V);
    end To_I_100;

  function To_U_100 (V : I_100) return U_100 is
    begin
      return U_100 (V);
    end To_U_100;

(continues on next page)
function To_D_50 (V : I_100) return D_50 is
  Min : constant I_100 := I_100 (D_50'First);
  Max : constant I_100 := I_100 (D_50'Last);
begin
  if V > Max then
  return D_50'Last;
  elsif V < Min then
  return D_50'First;
  else
  return D_50 (V);
  end if;
end To_D_50;

function To_S_50 (V : I_100) return S_50 is
begin
  if V > S_50'Last then
  return S_50'Last;
  elsif V < S_50'First then
  return S_50'First;
  else
  return V;
  end if;
end To_S_50;

function To_I_100 (V : D_50) return I_100 is
begin
  return I_100 (V);
end To_I_100;
end Int_Types;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Int_Types; use Int_Types;

procedure Main is
  package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
  package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
  package D_50_IO is new Ada.Text_IO.Integer_IO (D_50);

  use I_100_IO;
  use U_100_IO;
  use D_50_IO;

  type Test_Case_Index is
    (I_100_Range,
     U_100_Range,
     U_100_Wraparound,
     U_100_To_I_100,
     I_100_To_U_100,
     D_50_Range,
     S_50_Range,
     I_100_To_D_50,
     I_100_To_S_50,
     D_50_To_I_100,
     S_50_To_I_100);
  procedure Check (TC : Test_Case_Index) is
begin
I_100_IO.Default_Width := 1;
U_100_IO.Default_Width := 1;
D_50_IO.Default_Width := 1;

case TC is
  when I_100_Range =>
    Put (I_100'First);
    New_Line;
    Put (I_100'Last);
    New_Line;
  when U_100_Range =>
    Put (U_100'First);
    New_Line;
    Put (U_100'Last);
    New_Line;
  when U_100_Wraparound =>
    Put (U_100'First - 1);
    New_Line;
    Put (U_100'Last + 1);
    New_Line;
  when U_100_To_I_100 =>
    for I in U_100'Range loop
      I_100_IO.Put (To_I_100 (I));
      New_Line;
    end loop;
  when I_100_To_U_100 =>
    for I in I_100'Range loop
      Put (To_U_100 (I));
      New_Line;
    end loop;
  when D_50_Range =>
    Put (D_50'First);
    New_Line;
    Put (D_50'Last);
    New_Line;
  when S_50_Range =>
    Put (S_50'First);
    New_Line;
    Put (S_50'Last);
    New_Line;
  when I_100_To_D_50 =>
    for I in I_100'Range loop
      Put (To_D_50 (I));
      New_Line;
    end loop;
  when I_100_To_S_50 =>
    for I in I_100'Range loop
      Put (To_S_50 (I));
      New_Line;
    end loop;
  when D_50_To_I_100 =>
    for I in D_50'Range loop
      Put (To_I_100 (I));
      New_Line;
    end loop;
  when S_50_To_I_100 =>
    for I in S_50'Range loop
      Put (I);
      New_Line;
    end loop;
end case;

(continues on next page)
end case;
end Check;

begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting..."神通);
return;
elsif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...展"神通);
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.4.3 Temperatures

Listing 46: temperature_types.ads

package Temperature_Types is

type Celsius is digits 6 range -273.15 .. 5504.85;

type Int_Celsius is range -273 .. 5505;

function To_Celsius (T : Int_Celsius) return Celsius;

function To_Int_Celsius (T : Celsius) return Int_Celsius;

type Kelvin is digits 6 range 0.0 .. 5778.00;

function To_Celsius (T : Kelvin) return Celsius;

function To_Kelvin (T : Celsius) return Kelvin;

end Temperature_Types;

Listing 47: temperature_types.adb

package body Temperature_Types is

function To_Celsius (T : Int_Celsius) return Celsius is
Min : constant Float := Float (Celsius'First);
Max : constant Float := Float (Celsius'Last);

F : constant Float := Float (T);

begin
if F > Max then
return Celsius (Max);
elsif F < Min then
return Celsius (Min);
else
return Celsius (F);
end if;
end To_Celsius;

function To_Int_Celsius (T : Celsius) return Int_Celsius is
begin
return Int_Celsius (T);
end To_Int_Celsius;

function To_Celsius (T : Kelvin) return Celsius is
  F : constant Float := Float (T);
begin
  return Celsius (F - 273.15);
end To_Celsius;

function To_Kelvin (T : Celsius) return Kelvin is
  F : constant Float := Float (T);
begin
  return Kelvin (F + 273.15);
end To_Kelvin;
end Temperature_Types;

Listing 48: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Temperature_Types; use Temperature_Types;

procedure Main is
package Celsius_IO is new Ada.Text_IO.Float_IO (Celsius);
package Kelvin_IO is new Ada.Text_IO.Float_IO (Kelvin);
package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);
use Celsius_IO;
use Kelvin_IO;
use Int_Celsius_IO;

type Test_Case_Index is
  (Celsius_Range, 
   Celsius_To_Int_Celsius, 
   Int_Celsius_To_Celsius, 
   Kelvin_To_Celsius, 
   Celsius_To_Kelvin);

procedure Check (TC : Test_Case_Index) is 
begin
  Celsius_IO.Default_Fore := 1;
  Kelvin_IO.Default_Fore := 1;
  Int_Celsius_IO.Default_Width := 1;
  
case TC is
when Celsius_Range =>
  Put (Celsius'First);
  New_Line;
  Put (Celsius'Last);
  New_Line;
when Celsius_To_Int_Celsius =>
  Put (To_Int_Celsius (Celsius'First));
  New_Line;
  Put (To_Int_Celsius (0.0));
  New_Line;
  Put (To_Int_Celsius (Celsius'Last));
  New_Line;
when Int_Celsius_To_Celsius =>
  Put (To_Celsius (Int_Celsius'First));
  New_Line;
(continues on next page)
Put (To_Celsius (0));
New_Line;
Put (To_Celsius (Int_Celsius'Last));
New_Line;
when Kelvin_To_Celsius =>
  Put (To_Celsius (Kelvin'First));
New_Line;
Put (To_Celsius (0));
New_Line;
Put (To_Celsius (Kelvin'Last));
New_Line;
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.5 Records

18.5.1 Directions

Listing 49: directions.ads

package Directions is

  type Angle_Mod is mod 360;

  type Direction is
    (North,
     Northeast,
     East,
     Southeast,
     South,
     Southwest,
     West,
     Northwest);

  function To_Direction (N: Angle_Mod) return Direction;

  type Ext_Angle is record
    Angle_Elem : Angle_Mod;
    Direction_Elem : Direction;
  end record;

(continues on next page)
function To_EXT_Angle (N : Angle_Mod) return Ext_Angle;
procedure Display (N : Ext_Angle);
end Directions;

Listing 50: directions.adb

with Ada.Text_IO; use Ada.Text_IO;
package body Directions is
  procedure Display (N : Ext_Angle) is
    begin
      Put_Line ("Angle: "
        & Angle_Mod'Image (N.Angle_Elem)
        & " => "
        & Direction'Image (N.Direction_Elem)
        & ".");
    end Display;
  function To_Direction (N : Angle_Mod) return Direction is
    begin
      case N is
        when 0 => return North;
        when 1 .. 89 => return Northeast;
        when 90 => return East;
        when 91 .. 179 => return Southeast;
        when 180 => return South;
        when 181 .. 269 => return Southwest;
        when 270 => return West;
        when 271 .. 359 => return Northwest;
      end case;
    end To_Direction;
  function To_EXT_Angle (N : Angle_Mod) return Ext_Angle is
    begin
      return (Angle_Elem => N,
              Direction_Elem => To_Direction (N));
    end To_EXT_Angle;
end Directions;

Listing 51: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Directions; use Directions;
procedure Main is
  type Test_Case_Index is (Direction_Chk);
  procedure Check (TC : Test_Case_Index) is
    begin
      case TC is
        when Direction_Chk =>
          Display (To_EXT_Angle (0));
          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  Check (Test_Case_Index'Value (Argument (1)));
34  end Main;
35

18.5.2 Colors

Listing 52: color_types.ads

package Color_Types is

  type HTML_Color is
    (Salmon,
     Firebrick,
     Red,
     Darkred,
     Lime,
     Forestgreen,
     Green,
     Darkgreen,
     Blue,
     Mediumblue,
     Darkblue);

  function To_Integer (C : HTML_Color) return Integer;

  type Basic_HTML_Color is
    (Red,
     Green,
     Blue);

  function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;

  subtype Int_Color is Integer range 0 .. 255;

  type RGB is record
    Red : Int_Color;
    Green : Int_Color;
    Blue : Int_Color;
  end record;

  function To_RGB (C : HTML_Color) return RGB;

(continues on next page)
function Image (C : RGB) return String;
end Color_Types;

Listing 53: color_types.adb

with Ada.Integer_Text_IO;
package body Color_Types is

function To_Integer (C : HTML_Color) return Integer is
begin
  case C is
    when Salmon => return 16#FA8072#;
    when Firebrick => return 16#B22222#;
    when Red => return 16#FF0000#;
    when Darkred => return 16#8B0000#;
    when Lime => return 16#00FF00#;
    when Forestgreen => return 16#228B22#;
    when Green => return 16#008000#;
    when Darkgreen => return 16#006400#;
    when Blue => return 16#0000FF#;
    when Mediumblue => return 16#0000CD#;
    when Darkblue => return 16#00008B#;
  end case;
end To_Integer;

function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
begin
  case C is
    when Red => return Red;
    when Green => return Green;
    when Blue => return Blue;
  end case;
end To_HTML_Color;

function To_RGB (C : HTML_Color) return RGB is
begin
  case C is
    when Salmon => return (16#FA#, 16#80#, 16#72#);
    when Firebrick => return (16#B2#, 16#22#, 16#22#);
    when Red => return (16#FF#, 16#00#, 16#00#);
    when Darkred => return (16#8B#, 16#00#, 16#00#);
    when Lime => return (16#00#, 16#FF#, 16#00#);
    when Forestgreen => return (16#22#, 16#BB#, 16#22#);
    when Green => return (16#00#, 16#80#, 16#00#);
    when Darkgreen => return (16#00#, 16#64#, 16#00#);
    when Blue => return (16#00#, 16#00#, 16#FF#);
    when Mediumblue => return (16#00#, 16#00#, 16#CD#);
    when Darkblue => return (16#00#, 16#00#, 16#8B#);
  end case;
end To_RGB;

function Image (C : RGB) return String is
  subtype Str_Range is Integer range 1 .. 10;
  SR : String (Str_Range);
  SG : String (Str_Range);
  SB : String (Str_Range);
begin
  (continues on next page)
Ada.Integer_Text_IO.Put (To => SR, Item => C.Red, Base => 16);
Ada.Integer_Text_IO.Put (To => SG, Item => C.Green, Base => 16);
Ada.Integer_Text_IO.Put (To => SB, Item => C.Blue, Base => 16);
return "(Red => " & SR & ", Green => " & SG & ", Blue => " & SB & ")";
end Image;
end Color_Types;

Listing 54: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is (HTML_Color_To_RGB);

  procedure Check (TC : Test_Case_Index) is
    begin
      case TC is
        when HTML_Color_To_RGB =>
          for I in HTML_Color'Range loop
            Put_Line (HTML_Color'Image (I) & " => " & Image (To_RGB (I)) & ".");
          end loop;
        end case;
      end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
18.5.3 Inventory

Listing 55: inventory_pkg.ads

```ada
package Inventory_Pkg is

  type Item_Name is
    (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);

  function To_String (I : Item_Name) return String;

  type Item is record
    Name : Item_Name;
    Quantity : Natural;
    Price : Float;
  end record;

  function Init (Name : Item_Name;
    Quantity : Natural;
    Price : Float) return Item;

  procedure Add (Assets : in out Float;
    I : Item);

end Inventory_Pkg;
```

Listing 56: inventory_pkg.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Inventory_Pkg is

  function To_String (I : Item_Name) return String is
    begin
      case I is
        when Ballpoint_Pen => return "Ballpoint Pen";
        when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
        when Feather_Quill_Pen => return "Feather Quill Pen";
      end case;
    end To_String;

  function Init (Name : Item_Name;
    Quantity : Natural;
    Price : Float) return Item is
    begin
      Put_Line ("Item: " & To_String (Name) & ".");
      return (Name => Name,
        Quantity => Quantity,
        Price => Price);
    end Init;

  procedure Add (Assets : in out Float;
    I : Item) is
    begin
      Assets := Assets + Float (I.Quantity) * I.Price;
    end Add;

end Inventory_Pkg;
```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Inventory_Pkg; use Inventory_Pkg;

procedure Main is
   -- Remark: the following line is not relevant.
   F : array (1 .. 10) of Float := (others => 42.42);

type Test_Case_Index is
   (Inventory_Chk);

procedure Display (Assets : Float) is
   package F_IO is new Ada.Text_IO.Float_IO (Float);
   use F_IO;
begin
   Put ("Assets: ");
   Put (Assets, 1, 2, 0);
   Put (".");
   New_Line;
end Display;

procedure Check (TC : Test_Case_Index) is
I : Item;
Assets : Float := 0.0;

   -- Please ignore the following three lines!
pragma Warnings (Off, "default initialization");
for Assets'Address use F'Address;
pragma Warnings (On, "default initialization");
begin
   case TC is
      when Inventory_Chk =>
         I := Init (Ballpoint_Pen, 185, 0.15);
         Add (Assets, I);
         Display (Assets);

         I := Init (Oil_Based_Pen_Marker, 100, 9.0);
         Add (Assets, I);
         Display (Assets);

         I := Init (Feather_Quill_Pen, 2, 40.0);
         Add (Assets, I);
         Display (Assets);
      end case;
   end Check;

begin
   if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
   elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
   end if;

   Check (Test_Case_Index'Value (Argument (1)));
end Main;
18.6 Arrays

18.6.1 Constrained Array

Listing 58: constrained_arrays.ads

```ada
package Constrained_Arrays is

  type My_Index is range 1 .. 10;
  type My_Array is array (My_Index) of Integer;

  function Init return My_Array;
  procedure Double (A : in out My_Array);
  function First_Elem (A : My_Array) return Integer;
  function Last_Elem (A : My_Array) return Integer;
  function Length (A : My_Array) return Integer;

  A : My_Array := (1, 2, others => 42);

end Constrained_Arrays;
```

Listing 59: constrained_arrays.adb

```ada
package body Constrained_Arrays is

  function Init return My_Array is
    A : My_Array;
  begin
    for I in My_Array'Range loop
      A (I) := Integer (I);
    end loop;

    return A;
  end Init;

  procedure Double (A : in out My_Array) is
  begin
    for I in A'Range loop
      A (I) := A (I) * 2;
    end loop;
  end Double;

  function First_Elem (A : My_Array) return Integer is
  begin
    return A (A'First);
  end First_Elem;

  function Last_Elem (A : My_Array) return Integer is
  begin
    return A (A'Last);
  end Last_Elem;

  function Length (A : My_Array) return Integer is
  begin
    return A'Length;
  end Length;
```

(continues on next page)
Listing 60: main.adb

with Ada.Command_Line;  use Ada.Command_Line;
with Ada.Text_IO;  use Ada.Text_IO;
with Constrained_Arrays;  use Constrained_Arrays;

procedure Main is
  type Test_Case_Index is
    (Range_Chk,
     Array_Range_Chk,
     A_Obj_Chk,
     Init_Chk,
     Double_Chk,
     FirstElem_Chk,
     LastElem_Chk,
     Length_Chk);

  procedure Check (TC : Test_Case_Index) is
    begin
      AA : My_Array;
      procedure Display (A : My_Array) is
        begin
          for I in A'Range loop
            Put_Line (Integer'Image (A (I)));
          end loop;
        end Display;
      procedure Local_Init (A : in out My_Array) is
        begin
          A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
        end Local_Init;
      begin
        case TC is
          when Range_Chk =>
            for I in My_Index loop
              Put_Line (My_Index'Image (I));
            end loop;
          when Array_Range_Chk =>
            for I in My_Array'Range loop
              Put_Line (My_Index'Range (I));
            end loop;
          when A_Obj_Chk =>
            Display (A);
          when Init_Chk =>
            AA := Init;
            Display (AA);
          when Double_Chk =>
            Local_Init (AA);
            Display (AA);
          when FirstElem_Chk =>
            Local_Init (AA);
            Put_Line (Integer'Image (First_Elem (AA)));
          when LastElem_Chk =>
            Local_Init (AA);
            Put_Line (Integer'Image (Last_Elem (AA)));
          end case;
      begin
        end case;
    end Check;
end Main;
when Length_Chk =>
  Put_Line (Integer'Image (Length (AA)));
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.6.2 Colors: Lookup-Table

Listing 61: color_types.ads

package Color_Types is
  type HTML_Color is
    (Salmon,
     Firebrick,
     Red,
     Darkred,
     Lime,
     Forestgreen,
     Green,
     Darkgreen,
     Blue,
     Mediumblue,
     Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  type RGB is record
    Red : Int_Color;
    Green : Int_Color;
    Blue : Int_Color;
  end record;

  function To_RGB (C : HTML_Color) return RGB;

  function Image (C : RGB) return String;

  type HTML_Color_RGB is array (HTML_Color) of RGB;

  To_RGB_Lookup_Table : constant HTML_Color_RGB := (Salmon => (16#FA#, 16#80#, 16#72#),
                                                          Firebrick => (16#B2#, 16#22#, 16#22#),
                                                          Red => (16#FF#, 16#00#, 16#00#),
                                                          Darkred => (16#8B#, 16#00#, 16#00#),
                                                          Lime => (16#00#, 16#FF#, 16#00#),
                                                          Forestgreen => (16#22#, 16#8B#, 16#22#),
                                                          Green => (16#00#, 16#80#, 16#00#),
                                                          Darkgreen => (16#00#, 16#64#, 16#00#),
                                                          Blue => (16#00#, 16#00#, 16#FF#),

  (continues on next page)
with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;

package body Color_Types is

function To_RGB (C : HTML_Color) return RGB is
begin
    return To_RGB_Lookup_Table (C);
end To_RGB;

function Image (C : RGB) return String is
    subtype Str_Range is Integer range 1 .. 10;
    SR : String (Str_Range);
    SG : String (Str_Range);
    SB : String (Str_Range);
begin
    Ada.Integer_Text_IO.Put (To => SR,
                            Item => C.Red,
                            Base => 16);
    Ada.Integer_Text_IO.Put (To => SG,
                            Item => C.Green,
                            Base => 16);
    Ada.Integer_Text_IO.Put (To => SB,
                            Item => C.Blue,
                            Base => 16);

    return ((Red => " & SR
             & ", Green => " & SG
             & ", Blue => " & SB
             & ")");
end Image;

end Color_Types;

with Ada.Command_Line;   use Ada.Command_Line;
with Ada.Text_IO;        use Ada.Text_IO;
with Color_Types;        use Color_Types;

procedure Main is
    type Test_Case_Index is
        (Color_Table_Chk,  
         HTML_Color_To_Integer_Chk);

    procedure Check (TC : Test_Case_Index) is
begin
    case TC is
        when Color_Table_Chk =>
            Put_Line ("Size of HTML_Color_RGB: 
              & Integer'Image (HTML_Color_RGB'Length));
        when HTML_Color_To_Integer_Chk =>
            for I in HTML_Color'Range loop
                Put_Line ("Firebrick: 
              & Image (To_RGB_Lookup_Table (Firebrick)));
    end case;
end Check;
end Main;
Put_Line (HTML_Color'Image (I) & " => " & Image (To_RGB (I)) & ".");
end loop;
end case;
end Check;
begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.6.3 Unconstrained Array

Listing 64: unconstrained_arrays.ads

package Unconstrained_Arrays is

  type My_Array is array (Positive range <>) of Integer;

  procedure Init (A : in out My_Array);

  function Init (I, L : Positive) return My_Array;

  procedure Double (A : in out My_Array);

  function Diff_Prev_Elem (A : My_Array) return My_Array;

end Unconstrained_Arrays;

Listing 65: unconstrained_arrays.adb

procedure Init (A : in out My_Array) is
  Y : Natural := A'Last;
begin
  for I in A'Range loop
    A (I) := Y;
    Y := Y - 1;
  end loop;
end Init;

function Init (I, L : Positive) return My_Array is
  A : My_Array (I .. I + L - 1);
begin
  Init (A);
  return A;
end Init;

procedure Double (A : in out My_Array) is
begin
  for I in A'Range loop
    A (I) := A (I) * 2;
  end loop;
end Double;
A (I) := A (I) * 2;
end loop;
end Double;

function Diff_Prev_Elem (A : My_Array) return My_Array is
A_Out : My_Array (A'Range);
begint
A_Out (A'First) := 0;
for I in A'First + 1 .. A'Last loop
A_Out (I) := A (I) - A (I - 1);
end loop;
return A_Out;
end Diff_Prev_Elem;
end Unconstrained_Arrays;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Unconstrained_Arrays; use Unconstrained_Arrays;

procedure Main is

type Test_Case_Index is
(Init_Chk,
 Init_Proc_Chk,
 Double_Chk,
 Diff_Prev_Chk,
 Diff_Prev_Single_Chk);

procedure Check (TC : Test_Case_Index) is
AA : My_Array (1 .. 5);
AB : My_Array (5 .. 9);

procedure Display (A : My_Array) is
begin
for I in A'Range loop
Put_Line (Integer'image (A (I)));
end loop;
end Display;

procedure Local_Init (A : in out My_Array) is
begin
A := (1, 2, 5, 10, -10);
end Local_Init;
begin
case TC is
when Init_Chk =>
AA := Init (AA'First, AA'Length);
AB := Init (AB'First, AB'Length);
Display (AA);
Display (AB);
when Init_Proc_Chk =>
Init (AA);
Init (AB);
Display (AA);
Display (AB);
when Double_Chk =>
Introduction to Ada: Laboratories

(continued from previous page)

Local_Init (AB);
Double (AB);
Display (AB);

when Diff_Prev_Check =>
Local_Init (AB);
AB := Diff_Prev_Elem (AB);
Display (AB);

when Diff_Prev_Single_Chk =>

declare
A1 : My_Array (1..1) := (1 => 42);
begin
A1 := Diff_Prev_Elem (A1);
Display (A1);
end;
end case;
end Check;

begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elsif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.6.4 Product info

Listing 67: product_info_pkg.ads

package Product_Info_Pkg is

  subtype Quantity is Natural;

  subtype Currency is Float;

  type Product_Info is record
    Units : Quantity;
    Price : Currency;
  end record;

  type ProductInfos is array (Positive range <>) of Product_Info;

  type Currency_Array is array (Positive range <>) of Currency;

  procedure Total (P : ProductInfos;
                  Tot : out Currency_Array);

  function Total (P : ProductInfos) return Currency_Array;

  function Total (P : ProductInfos) return Currency;

end Product_Info_Pkg;

18.6. Arrays
### Listing 68: product_info_pkg.adb

```ada
package body Product_Info_Pkg is

-- Get total for single product
function Total (P : Product_INFO) return Currency is
  (Currency (P.Units) * P.Price);

procedure Total (P : Product_Infos;
                 Tot : out Currency_Array) is
begin
  for I in P'Range loop
    Tot (I) := Total (P (I));
  end loop;
end Total;

function Total (P : Product_Infos) return Currency_Array
  is
  Tot : Currency_Array (P'Range);
begin
  Total (P, Tot);
  return Tot;
end Total;

function Total (P : Product_Infos) return Currency
  is
  Tot : Currency := 0.0;
begin
  for I in P'Range loop
    Tot := Tot + Total (P (I));
  end loop;
  return Tot;
end Total;

end Product_Info_Pkg;
```

### Listing 69: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Product_Info_Pkg; use Product_Info_Pkg;

procedure Main is
  package Currency_IO is new Ada.Text_IO.Float_IO (Currency);

  type Test_Case_Index is
    (Total_Func_Chk,
     Total_Proc_Chk,
     Total_Value_Chk);

  procedure Check (TC : Test_Case_Index) is
    subtype Test_Range is Positive range 1 .. 5;

    P : Product_Infos (Test_Range);
    Tots : Currency_Array (Test_Range);
    Tot : Currency;

  procedure Display (Tots : Currency_Array) is
    begin
      for I in Tots'Range loop
```

(continues on next page)
Introduction to Ada: Laboratories

procedure Local_Init (P: in out ProductInfos) is
begin
  P := ((1, 0.5),
        (2, 10.0),
        (5, 40.0),
        (10, 10.0),
        (10, 20.0));
end Local_Init;

begin
  Currency_IO.Default_Fore := 1;
  Currency_IO.Default_Aft := 2;
  Currency_IO.Default_Exp := 0;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
end Main;

18.6.5 String_10

Listing 70: strings_10.ads

package Strings_10 is

  subtype String_10 is String (1 .. 10);

  -- Using "type String_10 is..." is possible, too.

function To_String_10 (S: String) return String_10;

(continues on next page)
Introduction to Ada: Laboratories

end Strings_10;

Listing 71: strings_10.adb

package body Strings_10 is

function To_String_10 (S : String) return String_10 is
  S_Out : String_10;
begin
  for I in String_10'First .. Integer'Min (String_10'Last, S'Last) loop
    S_Out (I) := S (I);
  end loop;
  for I in Integer'Min (String_10'Last + 1, S'Last + 1) .. String_10'Last loop
    S_Out (I) := ' ';
  end loop;
  return S_Out;
end To_String_10;
end Strings_10;

Listing 72: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Strings_10; use Strings_10;

procedure Main is
  type Test_Case_Index is
    (String_10_Long_Chk, String_10_Short_Chk);

  procedure Check (TC : Test Case Index) is
    SL : constant String := "And this is a long string just for testing...";
    SS : constant String := "Hey!";
    S_10 : String_10;
begin
  case TC is
    when String_10_Long_Chk =>
      S_10 := To_String_10 (SL);
      Put_Line (String (S_10));
    when String_10_Short_Chk =>
      S_10 := (others => ' ');
      S_10 := To_String_10 (SS);
      Put_Line (String (S_10));
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
  end if;

  Check (Test Case Index'Value (Argument (1)));
end Main;
18.6.6 List of Names

Listing 73: names_ages.ads

```ada
package Names_Ages is

    Max_People : constant Positive := 10;

    subtype Name_Type is String (1 .. 50);

type Age_Type is new Natural;

type Person is record
    Name : Name_Type;
    Age : Age_Type;
end record;

type People_Array is array (Positive range <>) of Person;

type People is record
    People_A : People_Array (1 .. Max_People);
    Last_Valid : Natural;
end record;

procedure Reset (P : in out People);

procedure Add (P : in out People;
    Name : String);

function Get (P : People;
    Name : String) return Age_Type;

procedure Update (P : in out People;
    Name : String;
    Age : Age_Type);

procedure Display (P : People);

end Names_Ages;
```

Listing 74: names_ages.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Names_Ages is

    function To_Name_Type (S : String) return Name_Type is
        S_Out : Name_Type := (others => ' ');
    begin
        for I in 1 .. Integer'Min (S'Last, Name_Type'Last) loop
            S_Out (I) := S (I);
        end loop;
    return S_Out;
end To_Name_Type;

procedure Init (P : in out Person;
    Name : String) is
begin

(continues on next page)```
P.Name := To_Name_Type (Name);
P.Age := 0;
end Init;

function Match (P : Person;
   Name : String) return Boolean is
begin
   return P.Name = To_Name_Type (Name);
end Match;

function Get (P : Person) return Age_Type is
begin
   return P.Age;
end Get;

procedure Update (P : in out Person;
   Age : Age_Type) is
begin
   P.Age := Age;
end Update;

procedure Display (P : Person) is
begin
   Put_Line ("NAME: " & Trim (P.Name, Right));
   Put_Line ("AGE: " & Age_Type'Image (P.Age));
end Display;

procedure Reset (P : in out People) is
begin
   P.Last_Valid := 0;
end Reset;

procedure Add (P : in out People;
   Name : String) is
begin
   P.Last_Valid := P.Last_Valid + 1;
   Init (P.People_A (P.Last_Valid), Name);
end Add;

function Get (P : People;
   Name : String) return Age_Type is
begin
   for I in P.People_A'First .. P.Last_Valid loop
      if Match (P.People_A (I), Name) then
         return Get (P.People_A (I));
      end if;
   end loop;
   return 0;
end Get;

procedure Update (P : in out People;
   Name : String;
   Age : Age_Type) is
begin
   for I in P.People_A'First .. P.Last_Valid loop
      if Match (P.People_A (I), Name) then
         Update (P.People_A (I), Age);
      end if;
   end loop;
end Update;
procedure Display (P : People) is
begin
  Put_Line ("LIST OF NAMES:" );
  for I in P.People_A'First .. P.Last_Valid loop
    Display (P.People_A (I));
  end loop;
end Display;
end Names_Ages;

Listing 75: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Names_Ages; use Names_Ages;

procedure Main is
  type Test_Case_Index is
    (Names_Ages_Chk,
     Get_Age_Chk);
  procedure Check (TC : Test_Case_Index) is
    P : People;
    begin
      case TC is
        when Names_Ages_Chk =>
          Reset (P);
          Add (P, "John");
          Add (P, "Patricia");
          Add (P, "Josh");
          Display (P);
          Update (P, "John", 18);
          Update (P, "Patricia", 35);
          Update (P, "Josh", 53);
          Display (P);
        when Get_Age_Chk =>
          Reset (P);
          Add (P, "Peter");
          Update (P, "Peter", 45);
          Put_Line ("Peter is 
             & Age_Type'Image (Get (P, "Peter"))
             & " years old.");
      end case;
    end Check;
begin
  if Argument_Count < 1 then
    Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
18.7 More About Types

18.7.1 Aggregate Initialization

Listing 76: aggregates.ads

package Aggregates is

  type Rec is record
    W : Integer := 10;
    X : Integer := 11;
    Y : Integer := 12;
    Z : Integer := 13;
  end record;

  type Int_Arr is array (1 .. 20) of Integer;

  procedure Init (R : out Rec);

  procedure Init_Some (A : out Int_Arr);

  procedure Init (A : out Int_Arr);

end Aggregates;

Listing 77: aggregates.adb

package body Aggregates is

  procedure Init (R : out Rec) is
  begin
    R := (X => 100,
        Y => 200,
        others => <>);
  end Init;

  procedure Init_Some (A : out Int_Arr) is
  begin
    A := (1 .. 5 => 99,
        others => 100);
  end Init_Some;

  procedure Init (A : out Int_Arr) is
  begin
    A := (others => 5);
  end Init;

end Aggregates;

Listing 78: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Aggregates; use Aggregates;

procedure Main is
  -- Remark: the following line is not relevant.
  F : array (1 .. 10) of Float := (others => 42.42)
  with Unreferenced;
(continues on next page)
type Test_Case_Index is
  (Default_Rec_Chk,
   Init_Rec_Chk,
   Init_Some_Arr_Chk,
   Init_Arr_Chk);

procedure Check (TC : Test_Case_Index) is
  A : Int_Arr;
  R : Rec;
  DR : constant Rec := (others => <>);
begin
  case TC is
    when Default_Rec_Chk =>
      R := DR;
      Put_Line ("Record Default:");
      Put_Line ("W => " & Integer'Image (R.W));
      Put_Line ("X => " & Integer'Image (R.X));
      Put_Line ("Y => " & Integer'Image (R.Y));
      Put_Line ("Z => " & Integer'Image (R.Z));
    when Init_Rec_Chk =>
      Init (R);
      Put_Line ("Record Init:");
      Put_Line ("W => " & Integer'Image (R.W));
      Put_Line ("X => " & Integer'Image (R.X));
      Put_Line ("Y => " & Integer'Image (R.Y));
      Put_Line ("Z => " & Integer'Image (R.Z));
    when Init_Some_Arr_Chk =>
      Init_Some (A);
      for I in A'Range loop
        Put_Line ("Array Init_Some:"
                  & Integer'Image (I) & "
                  & Integer'Image (A (I))) ;
      end loop;
    when Init_Arr_Chk =>
      Init (A);
      Put_Line ("Array Init:");
      for I in A'Range loop
        Put_Line ("Integer'Image (I) & "
                  & Integer'Image (A (I))) ;
      end loop;
  end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
18.7.2 Versioning

Listing 79: versioning.ads

```ada
package Versioning is

  type Version is record
    Major : Natural;
    Minor : Natural;
    Maintenance : Natural;
  end record;

  function Convert (V : Version) return String;

  function Convert (V : Version) return Float;

end Versioning;
```

Listing 80: versioning.adb

```ada
with Ada.Strings; use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Versioning is

  function Image_Trim (N : Natural) return String is
    S_N : constant String := Trim ('Image' (N), Left);
  begin
    return S_N;
  end Image_Trim;

  function Convert (V : Version) return String is
    S_Major : constant String := Image_Trim (V.Major);
    S_Minor : constant String := Image_Trim (V.Minor);
    S_Maint : constant String := Image_Trim (V.Maintenance);
  begin
    return (S_Major & "." & S_Minor & "." & S_Maint);
  end Convert;

  function Convert (V : Version) return Float is
  begin
    return Float (V.Major) + (Float (V.Minor) / 10.0);
  end Convert;

end Versioning;
```

Listing 81: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Versioning; use Versioning;

procedure Main is
  type Test_Case_Index is
    (Ver_String_Chk, Ver_Float_Chk);

  procedure Check (TC : Test_Case_Index) is
  begin
    V : constant Version := (1, 3, 23);
  end Check;

end Main;
```

(continues on next page)
case TC is
   when Ver_String_Chk =>
      Put_Line (Convert (V));
   when Ver_Float_Chk =>
      Put_Line (Float'Image (Convert (V)));
end case;
end Check;
begin
   if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
   elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
   end if;
   Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.7.3 Simple todo list

Listing 82: todo_lists.ads
package Todo_Lists is
   type Todo_Item is access String;
   type Todo_Items is array (Positive range <>) of Todo_Item;
   type Todo_List (Max.Len : Natural) is record
      Items : Todo_Items (1 .. Max.Len);
      Last : Natural := 0;
   end record;
   procedure Add (Todos : in out Todo_List;
                  Item     : String);
   procedure Display (Todos : Todo_List);
end Todo_Lists;

Listing 83: todo_lists.adb
with Ada.Text_IO; use Ada.Text_IO;
package body Todo_Lists is
   procedure Add (Todos : in out Todo_List;
                  Item : String) is
begin
   if Todos.Last < Todos.Items'Last then
      Todos.Last := Todos.Last + 1;
      Todos.Items (Todos.Last) := new String' (Item);
   else
      Put_Line ("ERROR: list is full!");
   end if;
end Add;
(continues on next page)
```
procedure Display (Todos : Todo_List) is
begin
  Put_Line ("TO-DO LIST");
  for I in Todos.Items'First .. Todos.Last loop
  Put_Line (Todos.Items (I).all);
  end loop;
  end Display;
end Todo_Lists;
```

Listing 84: main.adb

```
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is
  type Test_Case_Index is
    (Todo_List_Chk);
  procedure Check (TC : Test_Case_Index) is
    T : Todo_List (10);
    begin
      case TC is
        when Todo_List_Chk =>
          Add (T, "Buy milk");
          Add (T, "Buy tea");
          Add (T, "Buy present");
          Add (T, "Buy tickets");
          Add (T, "Pay electricity bill");
          Add (T, "Schedule dentist appointment");
          Add (T, "Call sister");
          Add (T, "Revise spreadsheet");
          Add (T, "Edit entry page");
          Add (T, "Select new design");
          Add (T, "Create upgrade plan");
          Display (T);
        end case;
      end Check;
      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exitting...");
          return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;
      Check (Test_Case_Index'Value (Argument (1)));
end Main;
```
18.7.4 Price list

Listing 85: price_lists.ads

```ada
package Price_Lists is

  type Price_Type is delta 0.01 digits 12;

  type Price_List_Array is array (Positive range <>) of Price_Type;

  type Price_List (Max : Positive) is record
    List : Price_List_Array (1 .. Max);
    Last : Natural := 0;
  end record;

  type Price_Result (Ok : Boolean) is record
    case Ok is
      when False => null;
      when True =>
        Price : Price_Type;
    end case;
  end record;

  procedure Reset (Prices : in out Price_List);
  procedure Add (Prices : in out Price_List; Item : Price_Type);
  function Get (Prices : Price_List; Idx : Positive) return Price_Result;
  procedure Display (Prices : Price_List);
end Price_Lists;
```

Listing 86: price_lists.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Price_Lists is

  procedure Reset (Prices : in out Price_List) is begin
    Prices.Last := 0;
  end Reset;

  procedure Add (Prices : in out Price_List; Item : Price_Type) is begin
    if Prices.Last < Prices.List'Last then
      Prices.Last := Prices.Last + 1;
      Prices.List (Prices.Last) := Item;
    else
      Put_Line ("ERROR: list is full!");
    end if;
  end Add;

  function Get (Prices : Price_List; Idx : Positive) return Price_Result is begin
    (continues on next page)
  end Get;

end Price_Lists;
```

(continues on next page)
if (Idx >= Prices.List'First and then
  Idx <= Prices.Last) then
  return Price_Result'(Ok => True,
                           Price => Prices.List (Idx));
else
  return Price_Result'(Ok => False);
end if;
end Get;

procedure Display (Prices : Price_List) is
begin
  Put_Line ("PRICE LIST");
  for I in Prices.List'First .. Prices.Last loop
    Put_Line (Price_Type'Image (Prices.List (I)));
  end loop;
  end Display;
end Price_Lists;

Listing 87: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Price_Lists; use Price_Lists;

procedure Main is
  type Test_Case_Index is
    (Price_Type_Chk,
     Price_List_Chk,
     Price_List_Get_Chk);
  L : Price_List (10);
  procedure Local_Init_List is
begin
  Reset (L);
  Add (L, 1.45);
  Add (L, 2.37);
  Add (L, 3.21);
  Add (L, 4.14);
  Add (L, 5.22);
  Add (L, 6.69);
  Add (L, 7.77);
  Add (L, 8.14);
  Add (L, 9.99);
  Add (L, 10.01);
end Local_Init_List;

procedure Get_Display (Idx : Positive) is
  R : constant Price_Result := Get (L, Idx);
begin
  Put_Line ("Attempt Get # " & Positive'Image (Idx));
  if R.Ok then
    Put_Line ("Element # " & Positive'Image (Idx)
           & " => " & Price_Type'Image (R.Price));
  else
    declare
      begin
        Put_Line ("Element # " & Positive'Image (Idx)
                     & " => " & Price_Type'Image (R.Price));
      end begin;
end if;
18.8 Privacy

18.8.1 Directions

Listing 88: directions.ads

```ada
package Directions is

  type Angle_Mod is mod 360;
  
  type Direction is
    (North, Northwest, West, Southwest, South, Southeast, East);

end Directions;
```

(continues on next page)
function To_Direction (N : Angle_Mod) return Direction;

type Ext_Angle is private;

function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;

procedure Display (N : Ext_Angle);

private

type Ext_Angle is record
   Angle_Elem : Angle_Mod;
   Direction_Elem : Direction;
end record;

end Directions;

Listing 89: directions.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Directions is

   procedure Display (N : Ext_Angle) is
      begin
         Put_Line ("Angle: ",
                  & Angle_Mod'Image (N.Angle_Elem)
                  & " => ",
                  & Direction'Image (N.Direction_Elem)
                  & ".");
      end Display;

   function To_Direction (N : Angle_Mod) return Direction is
      begin
         case N is
            when 0 => return East;
            when 1 .. 89 => return Northwest;
            when 90 => return North;
            when 91 .. 179 => return Northwest;
            when 180 => return West;
            when 181 .. 269 => return Southwest;
            when 270 => return South;
            when 271 .. 359 => return Southeast;
            end case;
      end To_Direction;

   function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
      begin
         return (Angle_Elem => N,
                  Direction_Elem => To_Direction (N));
      end To_Ext_Angle;

end Directions;

Listing 90: test_directions.adb

with Directions; use Directions;

procedure Test_Directions is
type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;

All_Directions : constant Ext_Angle_Array (1 .. 6) := (To_Ext_Angle (0),
                           To_Ext_Angle (45),
                           To_Ext_Angle (90),
                           To_Ext_Angle (91),
                           To_Ext_Angle (180),
                           To_Ext_Angle (270));

begin
   for I in All_Directions'Range loop
      Display (All_Directions (I));
   end loop;
end Test_Directions;

Listing 91: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Test_Case_Index is
      (Direction_Chk);

   procedure Check (TC : Test_Case_Index) is
   begin
      case TC is
         when Direction_Chk =>
            Test_Directions;
      end case;
   end Check;

begin
   if Argument_Count < 1 then
      Put_Line ("ERROR: missing arguments! Exiting...");
      return;
   elsif Argument_Count > 1 then
      Put_Line ("Ignoring additional arguments...");
      end if;
   Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.8.2 Limited Strings

Listing 92: limited_strings.ads

package Limited_Strings is
   type Lim_String is limited private;
   function Init (S : String) return Lim_String;
   function Init (Max : Positive) return Lim_String;
end Limited_Strings;
procedure Put_Line (LS : Lim_String);
procedure Copy (From : Lim_String;
                To : in out Lim_String);
function "=" (Ref, Dut : Lim_String) return Boolean;

private

type Lim_String is access String;

end Limited_Strings;

Listing 93: limited_strings.adb

with Ada.Text_IO;

package body Limited_Strings
is

function Init (S : String) return Lim_String is
  LS : constant Lim_String := new String' (S);
begin
  return LS;
end Init;

function Init (Max : Positive) return Lim_String is
  LS : constant Lim_String := new String (1 .. Max);
begin
  LS.all := (others => '_');
  return LS;
end Init;

procedure Put_Line (LS : Lim_String) is
begin
  Ada.Text_IO.Put_Line (LS.all);
end Put_Line;

function Get_Min_Last (A, B : Lim_String) return Positive is
begin
  return Positive'Min (A'Last, B'Last);
end Get_Min_Last;

procedure Copy (From : Lim_String;
                To : in out Lim_String) is
  Min_Last : constant Positive := Get_Min_Last (From, To);
begin
  To (To'First .. Min_Last) := From (To'First .. Min_Last);
  To (Min_Last + 1 .. To'Last) := (others => '_');
end;

function "=" (Ref, Dut : Lim_String) return Boolean is
begin
  for I in Dut'First .. Min_Last loop
    if Dut (I) /= Ref (I) then
      return False;
    end if;
  end loop;

end Limited_Strings;

(continues on next page)
Listing 94: check_lim_string.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Strings; use Limited_Strings;

procedure Check_Lim_String is
  S  : constant String   := "----------";
  S1 : constant Lim_String := Init ("Hello World");
  S2 : constant Lim_String := Init (30);
  S3 : Lim_String := Init (5);
  S4 : Lim_String := Init (S & S & S);
begin
  Put ("S1 => ");
  Put_Line (S1);
  Put ("S2 => ");
  Put_Line (S2);
  if S1 = S2 then
    Put_Line ("S1 is equal to S2.");
  else
    Put_Line ("S1 isn't equal to S2.");
  end if;
  Copy (From => S1, To => S3);
  Put ("S3 => ");
  Put_Line (S3);
  if S1 = S3 then
    Put_Line ("S1 is equal to S3.");
  else
    Put_Line ("S1 isn't equal to S3.");
  end if;
  Copy (From => S1, To => S4);
  Put ("S4 => ");
  Put_Line (S4);
  if S1 = S4 then
    Put_Line ("S1 is equal to S4.");
  else
    Put_Line ("S1 isn't equal to S4.");
  end if;
end Check_Lim_String;

Listing 95: main.adb

with Ada.Text_IO; use Ada.Text_IO;
with Limited_Strings; use Limited_Strings;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Check_Lim_String;

procedure Main is
  type Test_Case_Index is
    (Lim_String_Chk);
(continues on next page)
procedure Check (TC : Test_Case_Index) is
begin
  case TC is
  when Lim_String Chk =>
    Check.Lim_String;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting..."),
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.9 Generics

18.9.1 Display Array

Listing 96: display_array.ads

generic
  type T_Range is range <>;
  type T_Element is private;
  type T_Array is array (T_Range range <>) of T_Element;
  with function Image (E : T_Element) return String;
procedure Display_Array (Header : String;
                        A : T_Array);

Listing 97: display_array.adb

with Ada.Text_IO; use Ada.Text_IO;
procedure Display_Array (Header : String;
                        A : T_Array) is
begin
  Put_Line (Header);
  for I in A'Range loop
    Put_Line (T_Range'Image (I) & " : " & Image (A (I)));
  end loop;
end Display_Array;

Listing 98: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Display_Array;
procedure Main is
  type Test_Case Index is (Int_Array Chk,
procedure Test_Int_Array is
  type Int_Array is array (Positive range <>) of Integer;

  procedure Display_Int_Array is new
    Display_Array (T_Range => Positive,
                   T_Element => Integer,
                   T_Array => Int_Array,
                   Image => Integer'Image);

  A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
  begin
    Display_Int_Array ("Integers", A);
  end Test_Int_Array;

procedure Test_Point_Array is
  type Point is record
    X : Float;
    Y : Float;
  end record;

  type Point_Array is array (Natural range <>) of Point;

  function Image (P : Point) return String is
    begin
      return "(" & Float'Image (P.X) & ", " & Float'Image (P.Y) & ")";
    end Image;

  procedure Display_Point_Array is new
    Display_Array (T_Range => Natural,
                   T_Element => Point,
                   T_Array => Point_Array,
                   Image => Image);

  A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
                                            (5.0, 2.0), (-0.5, 2.0));
  begin
    Display_Point_Array ("Points", A);
  end Test_Point_Array;

procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Int_Array_Chk =>
        Test_Int_Array;
      when Point_Array_Chk =>
        Test_Point_Array;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
  (continues on next page)
18.9.2 Average of Array of Float

Listing 99: average.ads

```ada
generic
  type T_Range is range <>;
  type T_Element is digits <>;
  type T_Array is array (T_Range range <>) of T_Element;
function Average (A : T_Array) return T_Element;
``` 

Listing 100: average.adb

```ada
function Average (A : T_Array) return T_Element is
  Acc : Float := 0.0;
begins
for I in A'Range loop
  Acc := Acc + Float (A (I));
end loop;
return T_Element (Acc / Float (A'Length));
end Average;
```

Listing 101: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Average;

procedure Main is
  type Test_Case_Index is
    (Float_Array_CHK, Digits_7_Float_Array_CHK);

  procedure Test_Float_Array is
    type Float_Array is array (Positive range <>) of Float;
  function Average_Float is new
    Average (T_Range => Positive,
     T_Element => Float,
     T_Array => Float_Array);
  A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
begins
  Put_Line ("Average: " & Float'Image (Average_Float (A))); end Test_Float_Array;

  type Custom_Float is digits 7 range 0.0 .. 1.0;
  type Float_Array is
    array (Integer range <>) of Custom_Float;
  function Average_Float is new
    Average (T_Range => Integer,
     T_Element => Custom_Float,
     T_Array => Float_Array);
``` 

(continues on next page)
A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
begin
  Put_Line ("Average: 
    & Custom_Float'Image (Average_Float (A)));
end Test_Digits_7_Float_Array;

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Float_Array_Chk =>
      Test_Float_Array;
    when Digits_7_Float_Array_Chk =>
      Test_Digits_7_Float_Array;
  end case;
  end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.9.3 Average of Array of Any Type

Listing 102: average.ads

1 generic
  type T_Range is range <>;
  type T_Element is private;
  type T_Array is array (T_Range range <>) of T_Element;
  with function To_Float (E : T_Element) return Float is <>;
function Average (A : T_Array) return Float is
begin
  Acc := 0.0;
  for I in A'Range loop
    Acc := Acc + To_Float (A (I));
  end loop;
  return Acc / Float (A'Length);
end Average;

Listing 103: average.adb

Listing 104: test_item.ads

18.9. Generics
with Ada.Text_IO; use Ada.Text_IO;
with Average;
procedure Test_Item is
  package F_IO is new Ada.Text_IO.Float_IO (Float);
  type Amount is delta 0.01 digits 12;
  type Item is record
    Quantity : Natural;
    Price   : Amount;
  end record;
  type Item_Array is array (Positive range <>) of Item;
  function Get_Total (I : Item) return Float is (Float (I.Quantity) * Float (I.Price));
  function Get_Price (I : Item) return Float is (Float (I.Price));
  function Average_Total is new Average (T_Range => Positive,
    T_Element => Item,
    T_Array => Item_Array,
    To_Float => Get_Total);
  function Average_Price is new Average (T_Range => Positive,
    T_Element => Item,
    T_Array => Item_Array,
    To_Float => Get_Price);
  A : constant Item_Array (1..4) := ((Quantity => 5, Price => 10.00),
    (Quantity => 80, Price => 2.50),
    (Quantity => 40, Price => 5.00),
    (Quantity => 20, Price => 12.50));
begin
  Put ("Average per item & quantity: ");
  F_IO.Put (Average_Total (A), 3, 2, 0);
  New_Line;
  Put ("Average price: ");
  F_IO.Put (Average_Price (A), 3, 2, 0);
  New_Line;
end Test_Item;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Test_Item;
procedure Main is
  (continues on next page)
type Test_Case_Index is (Item_Array.chk);

procedure Check (TC : Test_Case_Index) is
begin
  case TC is
    when Item_Array.chk =>
      Test_Item;
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
    end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.9.4 Generic list

Listing 107: gen_list.ads

generic
  type Item is private;
  type Items is array (Positive range <>) of Item;
  Name : String;
  List_Array : in out Items;
  Last : in out Natural;
with procedure Put (I : Item) is <>;
package Gen_List is
  procedure Init;
  procedure Add (I : Item;
                 Status : out Boolean);
  procedure Display;
end Gen_List;

Listing 108: gen_list.adb

with Ada.Text_IO; use Ada.Text_IO;
package body Gen_List is
  procedure Init is
  begin
    Last := List_Array'First - 1;
    end Init;
  procedure Add (I : Item;
                 Status : out Boolean) is
  begin
    Status := Last < List_Array'Last;
end Add;
if Status then
    Last := Last + 1;
    List_Array (Last) := I;
end if;
end Add;

procedure Display is
begin
    Put_Line (Name);
    for I in List_Array'First .. Last loop
        Put (List_Array (I));
        New_Line;
    end loop;
end Display;
end Gen_List;

Listing 109: test_int.ads

procedure Test_Int;

Listing 110: test_int.adb

with Ada.Text_IO; use Ada.Text_IO;
with Gen_List;

procedure Test_Int is
    procedure Put (I : Integer) is
    begin
        Ada.Text_IO.Put (Integer'Image (I));
    end Put;

type Integer_Array is array (Positive range <>) of Integer;

A : Integer_Array (1 .. 3);
L : Natural;

package Int_List is new
Gen_List (Item => Integer,
        Items => Integer_Array,
        Name => "List of integers",
        List_Array => A,
        Last => L);

Success : Boolean;

procedure Display_Add_Success (Success : Boolean) is
begin
    if Success then
        Put_Line ("Added item successfully!");
    else
        Put_Line ("Couldn't add item!");
    end if;
end Display_Add_Success;

begin
Int_List.Init;

Int_List.Add (2, Success);
Display_Add_Success (Success);

Int_List.Add (5, Success);
Display_Add_Success (Success);

Int_List.Add (7, Success);
Display_Add_Success (Success);

Int_List.Add (8, Success);
Display_Add_Success (Success);

Int_List.Display;
end Test_Int;

Listing 111: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;

with Test_Int;

procedure Main is
  type Test_Case_Index is (Int_Chk);

  procedure Check (TC : Test_Case_Index) is begin
    case TC is
      when Int_Chk =>
        Test_Int;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1))); end Main;

18.10 Exceptions

18.10.1 Uninitialized Value

Listing 112: options.ads

package Options is
  type Option is (Uninitialized, Option_1, Option_2);
end Options;

18.10. Exceptions
Introduction to Ada: Laboratories

(continued from previous page)

Option_3);

Uninitialized_Value : exception;

function Image (O : Option) return String;

end Options;

Listing 113: options.adb

package body Options is

function Image (O : Option) return String is
begin
  case O is
    when Uninitialized =>
      raise Uninitialized_Value with "Uninitialized value detected!";
    when others =>
      return Option'Image (O);
  end case;
end Image;

end Options;

Listing 114: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Options; use Options;

procedure Main is
  type Test_Case_Index is
    (Options_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check (O : Option) is
    begin
      Put_Line (Image (O));
      exception
        when E : Uninitialized_Value =>
          Put_Line (Exception_Message (E));
          Check;
    end Check;

    begin
      case TC is
        when Options_Chk =>
          for O in Option loop
            Check (O);
          end loop;
      end case;
    end Check;

    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        (continues on next page)
Put_Line ("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.10.2 Numerical Exception

Listing 115: tests.ads

package Tests is

  type Test_ID is (Test_1, Test_2);
  Custom_Exception : exception;

  procedure Num_Exception_Test (ID : Test_ID);

end Tests;

Listing 116: tests.adb

package body Tests is

  pragma Warnings (Off, "variable ""C"" is assigned but never read");

  procedure Num_Exception_Test (ID : Test_ID) is
  begin
    case ID is
    when Test_1 =>
      A := Integer'Last;
      B := Integer'Last;
      C := A + B;
    when Test_2 =>
      raise Custom_Exception with "Custom_Exception raised!";
    end case;
  end Num_Exception_Test;

  pragma Warnings (On, "variable ""C"" is assigned but never read");

end Tests;

Listing 117: check_exception.adb

with Tests; use Tests;

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Check_Exception (ID : Test_ID) is
  begin
    Num_Exception_Test (ID);
  exception
    when Constraint_Error =>
      Put_Line ("Constraint_Error detected!");
    when E : others =>
      Put_Line (Exception_Message (E));
  end Check_Exception;

  (continues on next page)
Listing 118: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk,
     Exception_2_Chk);

  procedure Check (TC : Test_Case_Index) is
      procedure Check_Handle_Exception (ID : Test_ID) is
        begin
          Check_Exception (ID);
          exception
            when Constraint_Error =>
                Put_Line ("Constraint_Error" & " (raised by Check_Exception) detected!");
            when E : others =>
                Put_Line (Exception_Name (E) & " (raised by Check_Exception) detected!");
          end Check_Handle_Exception;
        begin
          case TC is
            when Exception_1_Chk =>
              Check_Handle_Exception (Test_1);
            when Exception_2_Chk =>
              Check_Handle_Exception (Test_2);
          end case;
        end Check;
      begin
        if Argument_Count < 1 then
          Put_Line ("ERROR: missing arguments! Exiting..." untward); return;
        elsif Argument_Count > 1 then
          Put_Line ("Ignoring additional arguments...");
        end if;

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

Chapter 18. Solutions
18.10.3 Re-raising Exceptions

Listing 119: tests.ads

```ada
package Tests is
  type Test_ID is (Test_1, Test_2);
  Custom_Exception, Another_Exception : exception;
  procedure Num_Exception_Test (ID : Test_ID);
end Tests;
```

Listing 120: tests.adb

```ada
package body Tests is
  pragma Warnings (Off, "variable "C" is assigned but never read");
  procedure Num_Exception_Test (ID : Test_ID) is
    A, B, C : Integer;
    begin
      case ID is
        when Test_1 =>
          A := Integer'Last;
          B := Integer'Last;
          C := A + B;
        when Test_2 =>
          raise Custom_Exception with "Custom_Exception raised!";
      end case;
      end Num_Exception_Test;
  pragma Warnings (On, "variable "C" is assigned but never read");
end Tests;
```

Listing 121: check_exception.ads

```ada
with Tests; use Tests;
procedure Check_Exception (ID : Test_ID);
```

Listing 122: check_exception.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
procedure Check_Exception (ID : Test_ID) is
  begin
    Num_Exception_Test (ID);
  exception
    when Constraint_Error =>
      Put_Line ("Constraint_Error detected!");
      raise;
    when E : others =>
      Put_Line (Exception_Message (E));
      raise Another_Exception;
  end Check_Exception;
```
Introduction to Ada: Laboratories

Listing 123: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
with Tests; use Tests;
with Check_Exception;

procedure Main is
  type Test_Case_Index is
    (Exception_1_Chk,
     Exception_2_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Handle_Exception (ID : Test_ID) is
      begin
        Check_Exception (ID);
        exception
          when Constraint_Error =>
            Put_Line ("Constraint_Error" & "} (raised by Check_Exception) detected!");
          when E : others =>
            Put_Line (Exception_Name (E) & " (raised by Check_Exception) detected!");
        end Check_Handle_Exception;
    begin
      case TC is
        when Exception_1_Chk =>
          Check_Handle_Exception (Test_1);
        when Exception_2_Chk =>
          Check_Handle_Exception (Test_2);
      end case;
    end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));
    end Main;
```

18.11 Tasking

18.11.1 Display Service

Listing 124: display_services.ads

```ada
package Display_Services is
  task type Display_Service is
```

(continues on next page)
Listing 125: display_services.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Display_Services is
  task body Display_Service is
    begin
      loop
        select
          accept Display (S : String) do
            Put_Line (S);
          end Display;
          or
          accept Display (I : Integer) do
            Put_Line (Integer'Image (I));
          end Display;
          or
          terminate;
        end loop;
      end Display_Service;
    end loop;
  end Display_Service;
end Display_Services;
```

Listing 126: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Display_Services; use Display_Services;

procedure Main is
  type Test_Case_Index is (Display_Service_Chk);
  procedure Check (TC : Test_Case_Index) is
    Display : Display_Service;
  begin
    case TC is
      when Display_Service_Chk =>
        Display.Display ("Hello");
        delay 0.5;
        Display.Display ("Hello again");
        delay 0.5;
        Display.Display (55);
        delay 0.5;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    (continues on next page)
```
28 Put_Line ("Ignoring additional arguments..."nten;)
29 30 Check (Test_Case_Index'Value (Argument (1)));
31 end Main;

18.11.2 Event Manager

Listing 127: event_managers.ads

with Ada.Real_Time; use Ada.Real_Time;

package Event_Managers is

  task type Event_Manager is
    entry Start (ID : Natural);
    entry Event (T : Time);
  end Event_Manager;

end Event_Managers;

Listing 128: event_managers.adb

with Ada.Text_IO; use Ada.Text_IO;

package body Event_Managers is

  task body Event_Manager is
    Event_ID : Natural := 0;
    Event_Delay : Time;
  begin
    accept Start (ID : Natural) do
      Event_ID := ID;
    end Start;

    accept Event (T : Time) do
      Event_Delay := T;
    end Event;

    delay until Event_Delay;
    Put_Line ("Event #" & Natural'Image (Event_ID));
  end Event_Manager;

end Event_Managers;

Listing 129: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Event_Managers; use Event_Managers;
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
  type Test_Case_Index is (Event_Manager_Chk);
  procedure Check (TC : Test_Case_Index) is
  end Main;
Ev_Mng : array (1 .. 5) of Event_Manager;

begin TC is
  when Event_Manager_Chk =>
    for I in Ev_Mng'Range loop
      Ev_Mng (I).Start (I);
    end loop;
    Ev_Mng (1).Event (Clock + Seconds (5));
    Ev_Mng (2).Event (Clock + Seconds (3));
    Ev_Mng (3).Event (Clock + Seconds (1));
    Ev_Mng (4).Event (Clock + Seconds (2));
    Ev_Mng (5).Event (Clock + Seconds (4));
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

**18.11.3 Generic Protected Queue**

Listing 130: gen_queues.ads

generic
  type Queue_Index is mod <>;
  type T is private;
package Gen_Queues is

  type Queue_Array is array (Queue_Index) of T;

protected type Queue is
  function Empty return Boolean;
  function Full return Boolean;
  entry Push (V : T);
  entry Pop (V : out T);
private
  N : Natural := 0;
  Idx : Queue_Index := Queue_Array'First;
  A : Queue_Array;
end Queue;
end Gen_Queues;

Listing 131: gen_queues.adb

package body Gen_Queues is

protected body Queue is
  function Empty return Boolean is
    (N = 0);
(continues on next page)
function Full return Boolean is
  (N = A'Length);

entry Push (V : T) when not Full is
begin
  Idx := Idx + 1;
  N := N + 1;
  A (Idx) := V;
end Push;

entry Pop (V : out T) when not Empty is
begin
  V := A (Idx - Queue_Index (N) - 1);
end Pop;
end Queue;
end Gen_Queues;

Listing 132: queue_tests.ads

package Queue_Tests is

  procedure Simple_Test;

  procedure Concurrent_Test;

end Queue_Tests;

Listing 133: queue_tests.adb

with Ada.Text_IO; use Ada.Text_IO;

with Gen_Queues;

package body Queue_Tests is

  Max : constant := 10;
  type Queue_Mod is mod Max;

  procedure Simple_Test is
  package Queues_Float is new Gen_Queues (Queue_Mod, Float);
    Q_F : Queues_Float.Queue;
    V : Float;
  begin
    V := 10.0;
    while not Q_F.Full loop
      Q_F.Push (V);
      V := V + 1.5;
    end loop;
    while not Q_F.Empty loop
      Q_F.Pop (V);
      Put_Line ("Value from queue: " & Float'Image (V));
    end loop;
  end Simple_Test;
end Queue_Tests;

end Simple_Test;

procedure Concurrent_Test is
package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
Q_I : Queues_Integer.Queue;
task T_Producer;
task T_Consumer;
task body T_Producer is
V : Integer := 100;
begnin
  for I in 1 .. 2 * Max loop
    V := V + 1;
  end loop;
end T_Producer;
task body T_Consumer is
V : Integer;
begnin
delay 1.5;

  while not Q_I.Empty loop
    Q_I.Pop (V);
    Put_Line ("Value from queue: " & Integer'Image (V));
delay 0.2;
  end loop;
end T_Consumer;
begnin
null;
end Concurrent_Test;
end Queue_Tests;

Listing 134: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Tests; use Queue_Tests;
procedure Main is
type Test_Case_Index is (Simple_Queue_Chk, Concurrent_Queue_Chk);
procedure Check (TC : Test_Case_Index) is
begin
case TC is
  when Simple_Queue_Chk =>
    Simple_Test;
  when Concurrent_Queue_Chk =>
    Concurrent_Test;
end case;
end Check;
begnin
if Argument_Count < 1 then
  Put_Line ("ERROR: missing arguments! Exiting...");
return;
end if;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  end if;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  end if;
elsif Argument_Count > 1 then
  Put_Line ("Ignoring additional arguments...");
end if;

Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.12 Design by contracts

18.12.1 Price Range

Listing 135: prices.ads

```ada
package Prices is
  type Amount is delta 10.0 ** (-2) digits 12;
  -- subtype Price is Amount range 0.0 .. Amount'Last;
  subtype Price is Amount
    with Static_Predicate => Price >= 0.0;
end Prices;
```

Listing 136: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Prices; use Prices;

procedure Main is
  type Test_Case_Index is
    (Price_Range_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Range (A : Amount) is
      P : constant Price := A;
      begin
        Put_Line ("Price: " & Price'Image (P));
        end Check_Range;
    begin
      case TC is
        when Price_Range_Chk =>
          Check_Range (-2.0);
        end case;
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error detected (NOT as expected)."");
        when Assert_Failure =>
          Put_Line ("Assert_Failure detected (as expected)."");
        end Check;
```

(continues on next page)
Introduction to Ada: Laboratories

begin
if Argument_Count < 1 then
   Put_Line ("ERROR: missing arguments! Exiting...");
   return;
elsif Argument_Count > 1 then
   Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.12.2 Pythagorean Theorem: Predicate

Listing 137: triangles.ads

package Triangles is
subtype Length is Integer;
type Right_Triangle is record
   H : Length := 0; -- Hypotenuse
   C1, C2 : Length := 0; -- Catheti / legs
end record
   with Dynamic_Predicate => H * H = C1 * C1 + C2 * C2;
function Init (H, C1, C2 : Length) return Right_Triangle is
   ((H, C1, C2));
end Triangles;

Listing 138: triangles-io.ads

package Triangles.IO is
   function Image (T : Right_Triangle) return String;
end Triangles.IO;

Listing 139: triangles-io.adb

package body Triangles.IO is
   function Image (T : Right_Triangle) return String is
      "(" & Length'Image (T.H) & "", " & Length'Image (T.C1)
      & ")")
   end Triangles.IO;

Listing 140: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;

18.12. Design by contracts
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

   type Test_Case_Index is
      (Triangle_8_6_Pass_Chk,
       Triangle_8_6_Fail_Chk,
       Triangle_10_24_Pass_Chk,
       Triangle_10_24_Fail_Chk,
       Triangle_18_24_Pass_Chk,
       Triangle_18_24_Fail_Chk);

   procedure Check (TC : Test_Case_Index) is
      procedure Check_Triangle (H, C1, C2 : Length) is
         T : Right_Triangle;
      begin
         T := Init (H, C1, C2);
         Put_Line (Image (T));
         exception
            when Constraint_Error =>
               Put_Line ("Constraint_Error detected (NOT as expected)." " );
            when Assert_Failure =>
               Put_Line ("Assert Failure detected (as expected)." );
      end Check_Triangle;

      begin
         case TC is
            when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
            when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
            when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
            when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
            when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
            when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
         end case;
      end Check;

      begin
         if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
         elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
         end if;

         Check (Test_Case_Index'Value (Argument (1)));
      end Main;
18.12.3 Pythagorean Theorem: Precondition

Listing 141: triangles.ads

package Triangles is

  subtype Length is Integer;

  type Right_Triangle is record
      H : Length := 0;
      -- Hypotenuse
      C1, C2 : Length := 0;
      -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    with Pre => H * H = C1 * C1 + C2 * C2;
  end Triangles;

Listing 142: triangles-io.ads

package Triangles.IO is

  function Image (T : Right_Triangle) return String;
end Triangles.IO;

Listing 143: triangles-io.adb

package body Triangles.IO is

  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ")";
end Triangles.IO;

Listing 144: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

  type Test_Case_Index is
    Triangle_8_6_Pass_Chk,
    Triangle_8_6_Fail_Chk,
    Triangle_10_24_Pass_Chk,
    Triangle_10_24_Fail_Chk,
    Triangle_18_24_Pass_Chk,
    Triangle_18_24_Fail_Chk;

  procedure Check (TC : Test_Case_Index) is (continues on next page)
procedure Check_Triangle (H, C1, C2 : Length) is
  T : Right_Triangle;
begin
  T := Init (H, C1, C2);
  Put_Line (Image (T));
exception
  when Constraint_Error =>
    Put_Line ("Constraint_Error detected (NOT as expected)." );
  when Assert_Failure =>
    Put_Line ("Assert_Failure detected (as expected)." );
end Check_Triangle;

begin
  case TC is
    when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
    when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
    when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
    when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
    when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
    when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.12.4 Pythagorean Theorem: Postcondition

Listing 145: triangles.ads

package Triangles is
  subtype Length is Integer;

  type Right_Triangle is record
    H : Length := 0;
    -- Hypotenuse
    C1, C2 : Length := 0;
    -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    (H, C1, C2)
  with Post => (Init'Result.H * Init'Result.H
   = Init'Result.C1 * Init'Result.C1
   + Init'Result.C2 * Init'Result.C2);

end Triangles;
Listing 146: triangles-io.ads

```ada
package Triangles.IO is

  function Image (T : Right_Triangle) return String;

end Triangles.IO;
```

Listing 147: triangles-io.adb

```ada
package body Triangles.IO is

  function Image (T : Right_Triangle) return String is
    "(" & Length'Image (T.H) & ", " & Length'Image (T.C1) & ", " & Length'Image (T.C2) & ")";

end Triangles.IO;
```

Listing 148: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is

  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk, Triangle_8_6_Fail_Chk, Triangle_10_24_Pass_Chk, Triangle_10_24_Fail_Chk, Triangle_18_24_Pass_Chk, Triangle_18_24_Fail_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error detected (NOT as expected)."");
          Assert_Failure =>
          Put_Line ("Assert_Failure detected (as expected)."");
      end Check_Triangle;

    begin
      case TC is
        when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
        when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
        when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
        when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
        when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
        when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
      end case;
    end Check;
```

(continues on next page)
end case;
end Check;

begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elsif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.12.5 Pythagorean Theorem: Type Invariant

Listing 149: triangles.ads

package Triangles is

  subtype Length is Integer;

  type Right_Triangle is private
    with Type_Invariant => Check (Right_Triangle);

  function Check (T : Right_Triangle) return Boolean;

  function Init (H, C1, C2 : Length) return Right_Triangle;

private

  type Right_Triangle is record
    H : Length := 0;
    -- Hypotenuse
    C1, C2 : Length := 0;
    -- Catheti / legs
  end record;

  function Init (H, C1, C2 : Length) return Right_Triangle is
    ((H, C1, C2));

  function Check (T : Right_Triangle) return Boolean is
    (T.H * T.H = T.C1 * T.C1 + T.C2 * T.C2);

end Triangles;

Listing 150: triangles-io.ads

package Triangles.IO is

  function Image (T : Right_Triangle) return String;

end Triangles.IO;

Listing 151: triangles-io.adb

package body Triangles.IO is

  (continues on next page)
function Image (T : Right_Triangle) return String is
  ("("
    & Length'Image (T.H)
  & ", "
    & Length'Image (T.C1)
  & ", "
    & Length'Image (T.C2)
  & ")");
end Triangles.IO;

Listing 152: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with System.Assertions; use System.Assertions;
with Triangles; use Triangles;
with Triangles.IO; use Triangles.IO;

procedure Main is
  type Test_Case_Index is
    (Triangle_8_6_Pass_Chk,
     Triangle_8_6_Fail_Chk,
     Triangle_10_24_Pass_Chk,
     Triangle_10_24_Fail_Chk,
     Triangle_18_24_Pass_Chk,
     Triangle_18_24_Fail_Chk);
  procedure Check (TC : Test_Case_Index) is
    procedure Check_Triangle (H, C1, C2 : Length) is
      T : Right_Triangle;
      begin
        T := Init (H, C1, C2);
        Put_Line (Image (T));
      exception
        when Constraint_Error =>
          Put_Line ("Constraint_Error detected (NOT as expected).");
        when Assert_Failure =>
          Put_Line ("Assert_Failure detected (as expected).");
        end Check_Triangle;
      begin
        case TC is
          when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
          when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
          when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
          when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
          when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
          when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
          end case;
        end Check;
    begin
      if Argument_Count < 1 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 1 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)));

18.12. Design by contracts 231
18.12.6 Primary Colors

Listing 153: color_types.ads

package Color_Types is

  type HTML_Color is
  (Salmon,
   Firebrick,
   Red,
   Darkred,
   Lime,
   Forestgreen,
   Green,
   Darkgreen,
   Blue,
   Mediumblue,
   Darkblue);

  subtype Int_Color is Integer range 0 .. 255;

  function Image (I : Int_Color) return String;

  type RGB is record
  Red : Int_Color;
  Green : Int_Color;
  Blue : Int_Color;
  end record;

  function To_RGB (C : HTML_Color) return RGB;

  function Image (C : RGB) return String;

  type HTML_Color_RGB_Array is array (HTML_Color) of RGB;

  To_RGB.Lookup_Table : constant HTML_Color_RGB_Array
  := (Salmon => (16#FA#, 16#80#, 16#72#),
       Firebrick => (16#B2#, 16#22#, 16#22#),
       Red => (16#FF#, 16#00#, 16#00#),
       Darkred => (16#8B#, 16#00#, 16#00#),
       Lime => (16#00#, 16#FF#, 16#00#),
       Forestgreen => (16#22#, 16#8B#, 16#22#),
       Green => (16#00#, 16#80#, 16#00#),
       Darkgreen => (16#00#, 16#00#, 16#00#),
       Blue => (16#00#, 16#00#, 16#FF#),
       Mediumblue => (16#00#, 16#00#, 16#CD#),
       Darkblue => (16#00#, 16#00#, 16#8B#));

  subtype HTML_RGB_Color is HTML_Color
  with Static_Predicate => HTML_RGB_Color in Red | Green | Blue;

  function To_Int_Color (C : HTML_Color;
                          S : HTML_RGB_Color) return Int_Color;

   -- Convert to hexadecimal value for the selected RGB component S
end Color_Types;
Listing 154: color_types.adb

```ada
with Ada.Integer_Text_IO;

package body Color_Types is

  function To_RGB (C : HTML_Color) return RGB is
    begin
      return To_RGB_Lookup_Table (C);
    end To_RGB;

  function To_Int_Color (C : HTML_Color; S : HTML_RGB_Color) return Int_Color is
    C_RGB : constant RGB := To_RGB (C);
    begin
      case S is
        when Red => return C_RGB.Red;
        when Green => return C_RGB.Green;
        when Blue => return C_RGB.Blue;
      end case;
    end To_Int_Color;

  function Image (I : Int_Color) return String is
    subtype Str_Range is Integer range 1 .. 10;
    S : String (Str_Range);
    begin
      Ada.Integer_Text_IO.Put (To => S, Item => I, Base => 16);
      return S;
    end Image;

  function Image (C : RGB) return String is
    begin
      return "(Red => " & Image (C.Red) & ", Green => " & Image (C.Green) & ", Blue => " & Image (C.Blue) & ")";
    end Image;

end Color_Types;
```

Listing 155: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Color_Types; use Color_Types;

procedure Main is
  type Test_Case_Index is
    (HTML_Color_Red_Chk, HTML_Color_Green_Chk, HTML_Color_Blue_Chk);

  procedure Check (TC : Test_Case_Index) is
    procedure Check_HTML_Colors (S : HTML_RGB_Color) is
      begin
        Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
        for I in HTML_Color'Range loop
```

(continues on next page)
Put_Line (HTML_Color'Image (I) & " => 
    & Image (To_Int_Color (I, S)) & ").
end loop;
end Check_HTML_Colors;

begin
  case TC is
    when HTML_Color_Red_CHK =>
      Check_HTML_Colors (Red);
    when HTML_Color_Green_CHK =>
      Check_HTML_Colors (Green);
    when HTML_Color_Blue_CHK =>
      Check_HTML_Colors (Blue);
  end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

---

### 18.13 Object-oriented programming

#### 18.13.1 Simple type extension

Listing 156: type_extensions.ads

```ada
package Type_Extensions is

  type T_Float is tagged record
    F : Float;
  end record;

  function Init (F : Float) return T_Float;
  function Init (I : Integer) return T_Float;
  function Image (T : T_Float) return String;

  type T_Mixed is new T_Float with record
    I : Integer;
  end record;

  function Init (F : Float) return T_Mixed;
  function Init (I : Integer) return T_Mixed;
  function Image (T : T_Mixed) return String;

end Type_Extensions;
```
package body Type_Extensions is

  function Init (F : Float) return T_Float is
    begin
    return ((F => F));
    end Init;

  function Init (I : Integer) return T_Float is
    begin
    return ((F => Float (I)));
    end Init;

  function Init (F : Float) return T_Mixed is
    begin
    return ((F => F,
     I => Integer (F)));
    end Init;

  function Init (I : Integer) return T_Mixed is
    begin
    return ((F => Float (I),
     I => I));
    end Init;

  function Image (T : T_Float) return String is
    begin
    return "{ F => " & Float'Image (T.F) & " }";
    end Image;

  function Image (T : T_Mixed) return String is
    begin
    return "{ F => " & Float'Image (T.F)
     & ", I => " & Integer'Image (T.I) & " }";
    end Image;

end Type_Extensions;

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Type_Extensions; use Type_Extensions;

procedure Main is

  type Test_Case_Index is
    (Type_Extension_Chk);

  procedure Check (TC : Test_Case_Index) is
    F1, F2 : T_Float;
    M1, M2 : T_Mixed;
    begin
    case TC is
    when Type_Extension_Chk =>
    F1 := Init (2.0);
    F2 := Init (3);
    M1 := Init (4.0);
    M2 := Init (5);

    (continues on next page)
if M2 in T_Float'Class then
  Put_Line ("T_Mixed is in T_Float'Class as expected");
end if;

Put_Line ("F1: " & Image (F1));
Put_Line ("F2: " & Image (F2));
Put_Line ("M1: " & Image (M1));
Put_Line ("M2: " & Image (M2));
end case;
end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.13.2 Online Store

Listing 159: online_store.ads

with Ada.Calendar; use Ada.Calendar;

package Online_Store is

  type Amount is delta 10.0**(2) digits 10;
  subtype Percentage is Amount range 0.0 .. 1.0;

  type Member is tagged record
    Start : Year_Number;
  end record;

  type Member_Access is access Member'Class;

  function Get_Status (M : Member) return String;
  function Get_Price (M : Member; P : Amount) return Amount;

  type Full_Member is new Member with record
    Discount : Percentage;
  end record;

  function Get_Status (M : Full_Member) return String;
  function Get_Price (M : Full_Member; P : Amount) return Amount;

end Online_Store;
package body Online_Store is

  function Get_Status (M : Member) return String is
    "Associate Member";

  function Get_Status (M : Full_Member) return String is
    "Full Member";

  function Get_Price (M : Member; P : Amount) return Amount is (P);

  function Get_Price (M : Full_Member; P : Amount) return Amount is
    (P * (1.0 - M.Discount));
end Online_Store;

package Online_Store.Tests is

  procedure Simple_Test;

end Online_Store.Tests;

with Ada.Text_IO; use Ada.Text_IO;

package body Online_Store.Tests is

  type Member_Due_Amount is record
    Member : Member_Access;
    Due_Amount : Amount;
  end record;

  function Get_Price (MA : Member_Due_Amount) return Amount is begin
    return MA.Member.Get_Price (MA.Due_Amount);
  end Get_Price;

  type Member_Due_Amounts is array (Positive range <>) of Member_Due_Amount;

  DB : constant Member_Due_Amounts (1 .. 4) := (
    (Member => new Member'(Start => 2010),
    Due_Amount => 250.0),
    (Member => new Full_Member'(Start => 1998,
      Discount => 0.1),
    Due_Amount => 160.0),
    (Member => new Full_Member'(Start => 1987,
      Discount => 0.2),
    Due_Amount => 400.0),
    (Member => new Member'(Start => 2013),
    Due_Amount => 110.0));

begin
  for I in DB'Range loop
    Put_Line ("Member #" & Positive'Image (I));
  end loop;
end Simple_Test;

(continues on next page)
Listing 163: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Online_Store; use Online_Store;
with Online_Store.Tests; use Online_Store.Tests;

procedure Main is

   type Test_Case_Index is
       (Type_Chk,
        Unit_Test_Chk);

   procedure Check (TC : Test_Case_Index) is

      function Result_Image (Result : Boolean) return String is
         (if Result then "OK" else "not OK");

      begin
         case TC is
            when Type_Chk =>
               declare
                  AM : constant Member := (Start => 2002);
                  FM : constant Full_Member := (Start => 1990,
                                               Discount => 0.2);
               begin
                  Put_Line ("Testing Status of Associate Member Type => "
                           & Result_Image (AM.Get_Status = "Associate Member");
                  Put_Line ("Testing Status of Full Member Type => "
                           & Result_Image (FM.Get_Status = "Full Member");
                  Put_Line ("Testing Discount of Associate Member Type => "
                           & Result_Image (AM.Get_Price (100.0) = 100.0));
                  Put_Line ("Testing Discount of Full Member Type => "
                           & Result_Image (FM.Get_Price (100.0) = 80.0));
               end;
            when Unit_Test_Chk =>
               Simple_Test;
         end case;
      end Check;

      begin
         if Argument_Count < 1 then
            Put_Line ("ERROR: missing arguments! Exiting...");
            return;
         elsif Argument_Count > 1 then
            Put_Line ("Ignoring additional arguments...");
         end if;
      Check (Test_Case_INDEX'Value (Argument (1)));
   end Main;
### Introduction to Ada: Laboratories

#### 18.14 Standard library: Containers

#### 18.14.1 Simple todo list

Listing 164: `todo_lists.ads`

```ada
with Ada.Containers.Vectors;

package Todo_Lists is
  type Todo_Item is access String;

  package Todo_List_Pkg is new Ada.Containers.Vectors
    (Index_Type => Natural,
     Element_Type => Todo_Item);

  subtype Todo_List is Todo_List_Pkg.Vector;

  procedure Add (Todos : in out Todo_List;
                 Item : String);

  procedure Display (Todos : Todo_List);
end Todo_Lists;
```

Listing 165: `todo_lists.adb`

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Todo_Lists is

  procedure Add (Todos : in out Todo_List;
                 Item : String) is
  begin
    Todos.Append (new String'(|Item|));
  end Add;

  procedure Display (Todos : Todo_List) is
  begin
    Put_Line ("TO-DO LIST");
    for T of Todos loop
      Put_Line (T.all);
    end loop;
  end Display;
end Todo_Lists;
```

Listing 166: `main.adb`

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Todo_Lists; use Todo_Lists;

procedure Main is
  type Test_Case_Index is
    (Todo_List_Chk);

  procedure Check (TC : Test_Case_Index) is
    T : Todo_List;
(continues on next page)```
begin
  case TC is
  when Todo_List_Chk =>
    Add (T, "Buy milk");
    Add (T, "Buy tea");
    Add (T, "Buy present");
    Add (T, "Buy tickets");
    Add (T, "Pay electricity bill");
    Add (T, "Schedule dentist appointment");
    Add (T, "Call sister");
    Add (T, "Revise spreadsheet");
    Add (T, "Edit entry page");
    Add (T, "Select new design");
    Add (T, "Create upgrade plan");
    Display (T);
  end case;
end Check;
begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.14.2 List of unique integers

Listing 167: ops.ads

with Ada.Containers.Ordered_Sets;
package Ops is
  type Int_Array is array (Positive range <>) of Integer;
  package Integer_Sets is new Ada.Containers.Ordered_Sets (Element_Type => Integer);
  subtype Int_Set is Integer_Sets.Set;
  function Get_Unique (A : Int_Array) return Int_Set;
  function Get_Unique (A : Int_Array) return Int_Array;
end Ops;

Listing 168: ops.adb

package body Ops is
  function Get_Unique (A : Int_Array) return Int_Set is
    S : Int_Set;
    begin
      for E of A loop
        (continues on next page)
function Get_Unique (A : Int_Array) return Int_Array is
  S : constant Int_Set := Get_Unique (A);
  AR : Int_Array (1 .. Positive (S.Length));
  I : Positive := 1;
  begin
    for E of S loop
      AR (I) := E;
      I := I + 1;
    end loop;
    return AR;
  end Get_Unique;
end Get_Unique;
end Ops;

Listing 169: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ops; use Ops;

procedure Main is
  type Test_Case_Index is
    (Get_Unique_Set_Chk,
     Get_Unique_Array_Chk);
  procedure Check (TC : Test_Case_Index;
                   A : Int_Array) is
    procedure Display_Unique_Set (A : Int_Array) is
      S : constant Int_Set := Get_Unique (A);
      begin
        for E of S loop
          Put_Line (Integer'Image (E));
        end loop;
        Display_Unique_Set;
      end Display_Unique_Set;
    procedure Display_Unique_Array (A : Int_Array) is
      AU : constant Int_Array := Get_Unique (A);
      begin
        for E of AU loop
          Put_Line (Integer'Image (E));
        end loop;
        Display_Unique_Array;
      end Display_Unique_Array;
      begin
        case TC is
          when Get_Unique_Set_Chk => Display_Unique_Set (A);
          when Get_Unique_Array_Chk => Display_Unique_Array (A);
        end case;
      end Check;
      if Argument_Count < 3 then
        begin
          (continues on next page)
18.15 Standard library: Dates & Times

18.15.1 Holocene calendar

Listing 170: to_holocene_year.adb

```ada
with Ada.Calendar; use Ada.Calendar;

function To_Holocene_Year (T : Time) return Integer is
begin
  return Year (T) + 10_000;
end To_Holocene_Year;
```

Listing 171: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;

with To_Holocene_Year;

procedure Main is
  type Test_Case_Index is
    (Holocene_Chk);

  procedure Display_Holocene_Year (Y : Year_Number) is
    HY : Integer;
  begin
    HY := To_Holocene_Year (Time_Of (Y, 1, 1));
    Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
    Put_Line ("Year (Holocene): " & Integer'Image (HY));
    end Display_Holocene_Year;

  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Holocene_Chk =>
        Display_Holocene_Year (2012);
        Display_Holocene_Year (2020);
    end case;
    end Check;
  begin
```

(continues on next page)
if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

### 18.15.2 List of events

Listing 172: events.ads

```ada
with Ada.Containers.Vectors;
package Events is
    type Event_Item is access String;
    package Event_Item_Containers is new Ada.Containers.Vectors
        (Index_Type => Positive,
         Element_Type => Event_Item);
    subtype Event_Items is Event_Item_Containers.Vector;
end Events;
```

Listing 173: events-lists.ads

```ada
with Ada.Calendar;    use Ada.Calendar;
with Ada.Containers.Ordered_Maps;
package Events.Lists is
    type Event_List is tagged private;
    procedure Add (Events : in out Event_List;
                   Event_Time : Time;
                   Event : String);
    procedure Display (Events : Event_List);
private
    package Event_Time_Item_Containers is new Ada.Containers.Ordered_Maps
        (Key_Type => Time,
         Element_Type => Event_Items,
         "=" => Event_Item_Containers."=");
    type Event_List is new Event_Time_Item_Containers.Map with null record;
end Events.Lists;
```

18.15. Standard library: Dates & Times 243
Introduction to Ada: Laboratories

Listing 174: events-lists.adb

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

package body Events.Lists is

procedure Add (Events : in out Event_List;
    Event_Time : Time;
    Event : String) is
  use Event_Item_Containers;
  E : constant Event_Item := new String'(Event);
begin
  if not Events.Contains (Event_Time) then
    Events.Include (Event_Time, Empty_Vector);
  end if;
  Events (Event_Time).Append (E);
end Add;

function Date_Image (T : Time) return String is
  Date_Img : constant String := Image (T);
begin
  return Date_Img (1 .. 10);
end;

procedure Display (Events : Event_List) is
  use Event_Time_Item_Containers;
  T : Time;
begin
  Put_Line ("EVENTS LIST");
  for C in Events.Iterate loop
    T := Key (C);
    Put_Line ("- " & Date_Image (T));
    for I of Events (C) loop
      Put_Line (" - " & I.all);
    end loop;
  end loop;
end Display;
end Events.Lists;
```

Listing 175: main.adb

```ada
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Events.Lists; use Events.Lists;

procedure Main is
  type Test_Case_Index is (Event_List_Chk);

  procedure Check (TC : Test_Case_Index) is
    EL : Event_List;
  begin
    case TC is
      when Event_List_Chk =>
        EL.Add (Time_Of (2018, 2, 16), "Final check");
    end case;
  end Check;
begin
  Check (Event_List_Chk);
end Main;
```

(continues on next page)
EL.Add (Time_Of (2018, 2, 16),
"Release");
EL.Add (Time_Of (2018, 12, 3),
"Brother's birthday");
EL.Add (Time_Of (2018, 1, 1),
"New Year's Day");
EL.Display;
end case;
end Check;

begin
if Argument_Count < 1 then
Put_Line ("ERROR: missing arguments! Exiting...");
return;
elseif Argument_Count > 1 then
Put_Line ("Ignoring additional arguments...");
end if;
Check (Test_Case_Index'Value (Argument (1)));
end Main;

18.16 Standard library: Strings

18.16.1 Concatenation

Listing 176: str_concat.ads


package Str_Concat is

  type Unbounded_Strings is array (Positive range <>) of Unbounded_String;

  function Concat (USA : Unbounded_Strings;
                   Trim_Str : Boolean;
                   Add_Whitespace : Boolean) return Unbounded_String;

  function Concat (USA : Unbounded_Strings;
                   Trim_Str : Boolean;
                   Add_Whitespace : Boolean) return String;

end Str_Concat;

Listing 177: str_concat.adb

with Ada.Strings; use Ada.Strings;

package body Str_Concat is

  function Concat (USA : Unbounded_Strings;
                   Trim_Str : Boolean;
                   Add_Whitespace : Boolean) return Unbounded_String is

    function Retrieve (USA : Unbounded_Strings;
                       Trim_Str : Boolean;
                       Index : Positive) return Unbounded_String is

    US_Internal : Unbounded_String := USA (Index);

    return US_Internal;

end Retri

(continues on next page)
begin
  if Trim_Str then
    US_Internal := Trim (US_Internal, Both);
  end if;
  return US_Internal;
end Retrieve;

US : Unbounded_String := To_Unbounded_String ("");
begin
  for I in USA'First .. USA'Last - 1 loop
    US := US & Retrieve (USA, Trim_Str, I);
    if Add_Whitespace then
      US := US & " ";
    end if;
  end loop;
  US := US & Retrieve (USA, Trim_Str, USA'Last);
  return US;
end Concat;

function Concat (USA : Unbounded_Strings;
                 Trim_Str : Boolean;
                 Add_Whitespace : Boolean) return String is
begin
  return To_String (Concat (USA, Trim_Str, Add_Whitespace));
end Concat;

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Command_Line; use Ada.Command_Line;
with Str_Concat; use Str_Concat;

procedure Main is
  type Test_Case_Index is
    (Unbounded_Concat_No_Trim_No_WS_Chk, Unbounded_Concat_Trim_No_WS_Chk, String_Concat_Trim_WS_Chk, Concat_Single_Element);
  procedure Check (TC : Test_Case_Index) is
  begin
    case TC is
      when Unbounded_Concat_No_Trim_No_WS_Chk =>
        declare
          S : constant Unbounded_Strings := (To_Unbounded_String ("Hello"), To_Unbounded_String (" World"), To_Unbounded_String ("!"));
        begin
          Put_Line (To_String (Concat (S, False, False))); end;
      when Unbounded_Concat_Trim_No_WS_Chk =>
        declare
          S : constant Unbounded_Strings := (To_Unbounded_String (" This "), To_Unbounded_String (" _is_ "),
To_Unbounded_String (" a "),
To_Unbounded_String (" _check ");
begin
Put_Line (To_String (Concat (S, True, False)));
end;
when String_Concat_Trim_WS_Chk =>
declare
  S : constant Unbounded_Strings := (
    To_Unbounded_String (" This "),
    To_Unbounded_String (" is a "),
    To_Unbounded_String (" test. "));
begin
  Put_Line (Concat (S, True, True));
end;
when Concat_Single_Element =>
declare
  S : constant Unbounded_Strings := (
    1 => To_Unbounded_String (" Hi "));
begin
  Put_Line (Concat (S, True, True));
end;
end case;
end Check;
end Main;

18.16.2 List of events

Listing 179: events.ads

with Ada.Containers.Vectors;

package Events is
  subtype Event_Item is Unbounded_String;

  package Event_Item_Containers is new
    Ada.Containers.Vectors
      (Index_Type => Positive,
       Element_Type => Event_Item);

  subtype Event_Items is Event_Item_Containers.Vector;
end Events;

Listing 180: events-lists.ads

with Ada.Calendar; use Ada.Calendar;
with Ada.Containers.Ordered_Maps;

package Events.Lists is

  type Event_List is tagged private;

  procedure Add (Events : in out Event_List;
                 Event_Time :   Time;
                 Event      :   String);

  procedure Display (Events : Event_List);

private

  package Event_Time_Item_Containers is new
    Ada.Containers.Ordered_Maps
      (Key_Type => Time,
       Element_Type => Event_Items,
       "=" => Event_Item_Containers."=");

  type Event_List is new Event_Time_Item_Containers.Map with null record;

end Events.Lists;

Listing 181: events-lists.adb

with Ada.Text_IO;       use Ada.Text_IO;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

package body Events.Lists is

  procedure Add (Events : in out Event_List;
                 Event_Time :   Time;
                 Event      :   String) is
    use Event_Item_Containers;
    E : constant Event_Item := To_Unbounded_String (Event);
  begin
    if not Events.Contains (Event_Time) then
      Events.Include (Event_Time, Empty_Vector);
    end if;
    Events (Event_Time).Append (E);
    Add;
  end Add;

  function Date_Image (T : Time) return String is
    Date_Img : constant String := Image (T);
  begin
    return Date_Img (1 .. 10);
  end;

  procedure Display (Events : Event_List) is
    use Event_Time_Item_Containers;
    T : Time;
  begin
    Put_Line ("EVENTS LIST");
    for C in Events.Iterate loop
      T := Key (C);
      Put_Line ("- " & Date_Image (T));
      for I of Events (C) loop
        Put_Line (" - " & To_String (I));
      end loop;
    end loop;
  end Display;

(continues on next page)
end Display;
end Events.Lists;

Listing 182: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

with Events;
with Events.Lists; use Events.Lists;

procedure Main is

  type Test_Case_Index is
    (Unbounded_String_Chk,
     Event_List_Chk);

  procedure Check (TC : Test_Case_Index) is
    EL : Event_List;
  begin
    case TC is
      when Unbounded_String_Chk =>
        declare
          S : constant Events.Event_Item := To_Unbounded_String ("Checked");
        begin
          Put_Line (To_String (S));
        end;
      when Event_List_Chk =>
        EL.Add (Time_Of (2018, 2, 16),
          "Final check");
        EL.Add (Time_Of (2018, 2, 16),
          "Release");
        EL.Add (Time_Of (2018, 12, 3),
          "Brother's birthday");
        EL.Add (Time_Of (2018, 1, 1),
          "New Year's Day");
        EL.Display;
    end case;
  end Check;

begin
  if Argument_Count < 1 then
    Put_Line ("ERROR: missing arguments! Exiting...");
    return;
  elsif Argument_Count > 1 then
    Put_Line ("Ignoring additional arguments...");
  end if;
  Check (Test_Case_Index'Value (Argument (1)));
end Main;
18.17 Standard library: Numerics

18.17.1 Decibel Factor

Listing 183: decibels.ads

```adacode
package Decibels is
    subtype Decibel is Float;
    subtype Factor is Float;

    function To_Decibel (F : Factor) return Decibel;
    function To_Factor (D : Decibel) return Factor;
end Decibels;
```

Listing 184: decibels.adb

```adacode

package body Decibels is
    function To_Decibel (F : Factor) return Decibel is
        begin
            return 20.0 * Log (F, 10.0);
        end To_Decibel;

    function To_Factor (D : Decibel) return Factor is
        begin
            return 10.0 ** (D / 20.0);
        end To_Factor;
end Decibels;
```

Listing 185: main.adb

```adacode
with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Decibels; use Decibels;

procedure Main is
    type Test_Case_Index is
        (Db_Chk, Factor_Chk);

    procedure Check (TC : Test_Case_Index; V : Float) is
        package F_IO is new Ada.Text_IO.Float_IO (Factor);
        package D_IO is new Ada.Text_IO.Float_IO (Decibel);

        procedure Put_Decibel_Cnvt (D : Decibel) is
            F : constant Factor := To_Factor (D);
            begin
                D_IO.Put (D, 0, 2, 0);
                F_IO.Put (F, 0, 2, 0);
                New_Line;
            end;
```

(continues on next page)
procedure Put_Factor_Cnvtn (F : Factor) is
    D : constant Decibel := To_Decibel (F);
begin
    Put ("Factor of ");
    F_IO.Put (F, 0, 2, 0);
    Put (" => ");
    D_IO.Put (D, 0, 2, 0);
    Put_Line (" dB");
end;

begin
    case TC is
        when Db_Chk =>
            Put_Decibel_Cnvtn (Decibel (V));
        when Factor_Chk =>
            Put_Factor_Cnvtn (Factor (V));
    end case;
end Check;

begin
    if Argument_Count < 2 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
    elsif Argument_Count > 2 then
        Put_Line ("Ignoring additional arguments...");
    end if;

    Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
end Main;

18.17.2 Root-Mean-Square

Listing 186: signals.ads

package Signals is
    subtype Sig_Value is Float;
    type Signal is array (Natural range <>) of Sig_Value;
    function Rms (S : Signal) return Sig_Value;
end Signals;

Listing 187: signals.adb


package body Signals is
    function Rms (S : Signal) return Sig_Value is
        Acc : Float := 0.0;
    begin
        for V of S loop
            Acc := Acc + V * V;
        end loop;
        return Sqrt (Acc / Float (S'Length));
end;
Introduction to Ada: Laboratories

(continued from previous page)

end;
end Signals;

Listing 188: signals-std.ads

package Signals.Std is
  Sample_Rate : Float := 8000.0;
  function Generate_Sine (N : Positive; Freq : Float) return Signal;
  function Generate_Square (N : Positive) return Signal;
  function Generate_Triangular (N : Positive) return Signal;
end Signals.Std;

Listing 189: signals-std.adb

with Ada.Numerics; use Ada.Numerics;

package body Signals.Std is
  function Generate_Sine (N : Positive; Freq : Float) return Signal is
    S : Signal (0 .. N - 1);
    begin
      for I in S'First .. S'Last loop
        S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
      end loop;
      return S;
    end;
  end;

  function Generate_Square (N : Positive) return Signal is
    S : constant Signal (0 .. N - 1) := (others => 1.0);
    begin
      return S;
    end;

  function Generate_Triangular (N : Positive) return Signal is
    S : Signal (0 .. N - 1);
    S_Half : constant Natural := S'Last / 2;
    begin
      for I in S'First .. S_Half loop
        S (I) := 1.0 * (Float (I) / Float (S_Half));
      end loop;
      for I in S_Half .. S'Last loop
        S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
      end loop;
      return S;
    end;
end Signals.Std;
with Ada.Command_Line; use Ada.Command_Line;  
with Ada.Text_IO; use Ada.Text_IO;  
with Signals; use Signals;  
with Signals.Std; use Signals.Std;  

procedure Main is  
  type Test_Case_Index is  
    (Sine_Signal_Chk,  
     Square_Signal_Chk,  
     Triangular_Signal_Chk);  
  procedure Check (TC : Test_Case_Index) is  
    package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);  
    N : constant Positive := 1024;  
    S_Si : constant Signal := Generate_Sine (N, 440.0);  
    S_Sq : constant Signal := Generate_Square (N);  
    S_Tr : constant Signal := Generate_Triangular (N + 1);  
    begin  
      case TC is  
        when Sine_Signal_Chk =>  
          Put ("RMS of Sine Signal: ");  
          Sig_IO.Put (Rms (S_Si), 0, 2, 0);  
          New_Line;  
        when Square_Signal_Chk =>  
          Put ("RMS of Square Signal: ");  
          Sig_IO.Put (Rms (S_Sq), 0, 2, 0);  
          New_Line;  
        when Triangular_Signal_Chk =>  
          Put ("RMS of Triangular Signal: ");  
          Sig_IO.Put (Rms (S_Tr), 0, 2, 0);  
          New_Line;  
      end case;  
    end Check;  
  begin  
    if Argument_Count < 1 then  
      Put_Line ("ERROR: missing arguments! Exiting...");  
      return;  
    elsif Argument_Count > 1 then  
      Put_Line ("Ignoring additional arguments...");  
    end if;  
    Check (Test_Case_Index'Value (Argument (1)));  
  end Main;  

18.17.3 Rotation

Listing 191: rotation.ads

package Rotation is  
  type Complex_Points is array (Positive range <>) of Complex;  
  (continues on next page)
function Rotation (N : Positive) return Complex_Points;
end Rotation;

Listing 192: rotation.adb

with Ada.Numerics; use Ada.Numerics;

package body Rotation is

    function Rotation (N : Positive) return Complex_Points is
        C_Angle : constant Complex :=
            Compose_From_Polar (1.0, 2.0 * Pi / Float (N));
    begin
        return C : Complex_Points (1 .. N + 1) do
            C (1) := Compose_From_Cartesian (1.0, 0.0);
            for I in C'First + 1 .. C'Last loop
                C (I) := C (I - 1) * C_Angle;
            end loop;
        end return;
    end Rotation;
end Rotation;

Listing 193: angles.ads

with Rotation; use Rotation;

package Angles is

    subtype Angle is Float;

    type Angles is array (Positive range <>) of Angle;

    function To_Angles (C : Complex_Points) return Angles;
end Angles;

Listing 194: angles.adb

with Ada.Numerics; use Ada.Numerics;

package body Angles is

    function To_Angles (C : Complex_Points) return Angles is
    begin
        return A : Angles (C'Range) do
            for I in A'Range loop
                A (I) := Argument (C (I)) / Pi * 180.0;
            end loop;
        end return;
    end To_Angles;
end Angles;
package Rotation.Tests is

    procedure Test_Rotation (N : Positive);

    procedure Test_Angles (N : Positive);

end Rotation.Tests;

package body Rotation.Tests is

    package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);

    package F_IO is new Ada.Text_IO.Float_IO (Float);

    -- Adapt value due to floating-point inaccuracies
    --
    function Adapt (C : Complex) return Complex is
        function Check_Zero (F : Float) return Float is
            (if F <= 0.0 and F >= -0.01 then 0.0 else F);
        begin
            return C_Out : Complex := C do
                C_Out.Re := Check_Zero (C_Out.Re);
                C_Out.Im := Check_Zero (C_Out.Im);
            end return;
        end Adapt;

    function Adapt (A : Angle) return Angle is
        (if A <= -179.99 and A >= -180.01 then 180.0 else A);

    procedure Test_Rotation (N : Positive) is
        C : constant Complex_Points := Rotation (N);
        begin
            Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
            for V of C loop
                Put ('Point: ');
                C_IO.Put (Adapt (V), 0, 1, 0);
                New_Line;
            end loop;
        end Test_Rotation;

    procedure Test_Angles (N : Positive) is
        C : constant Complex_Points := Rotation (N);
        A : constant Angles.Angles := To_Angles (C);
        begin
            Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
            for V of A loop
                Put ('Angle: ');
                F_IO.Put (Adapt (V), 0, 2, 0);
                Put_Line (" degrees");
            end loop;
        end Test_Angles;

    end Rotation.Tests;

18.17. Standard library: Numerics
end Test_Angles;
end Rotation.Tests;

Listing 197: main.adb

with Ada.Command_Line; use Ada.Command_Line;
with Ada.Text_IO; use Ada.Text_IO;
with Rotation.Tests; use Rotation.Tests;

procedure Main is
  type Test_Case_Index is
    (Rotation_Chk, Angles_Chk);

  procedure Check (TC : Test_Case_Index; N : Positive) is
    begin
      case TC is
        when Rotation_Chk =>
          Test_Rotation (N);
        when Angles_Chk =>
          Test_Angles (N);
      end case;
    end Check;

    begin
      if Argument_Count < 2 then
        Put_Line ("ERROR: missing arguments! Exiting...");
        return;
      elsif Argument_Count > 2 then
        Put_Line ("Ignoring additional arguments...");
      end if;
      Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
    end Main;