Programming in Ada – Notes

Daniel Duggan

December 1, 2025

Contents

1 Ada Basics

Ada is strongly typed with almost no type inference (e.g. let x = 10; compiler will not know that x is an integer). Further it uses nominal typing. Therefore, even if two types have identical structure (e.g. are arrays of the same length, both are integer ranges), they are considered distinct by Ada unless an explicit conversion is done by the programmer. For example

```
type Meters  is new Float;
type Seconds is new Float;

Distance : Meters  := 5.0;
Time     : Seconds := Seconds(Distance);  -- Explicitly convert Meters to
    Seconds
                                           -- Think of it like to_int(distance)
    :
                                           -- a value must be converted to the
    target type
```

Listing 1: Conversion must be explicit

In this example, `Meters` and `Seconds` are *derived types*: that is, new types created from an existing type (`Float`). Although they inherit the same operations and have the same internal representation, they are considered distinct types and are not assignment-compatible.
The variables `Distance` and `Time` are therefore of different types, even though they are both represented using floating-point values. Ada prevents mixing such types implicitly. As a result, assigning `Distance` to `Time` requires an explicit type conversion, as shown in the code. This does not represent a meaningful physical conversion (meters to seconds) but simply demonstrates Ada's rule that values of derived types must be explicitly converted when used in contexts expecting a different type.
Ada is a imperative and procedural language at its core (as opposed to declarative/functional).

2 Chapter 2 Notes: Program Structure in Ada

Ada programs are composed of several key building blocks: **packages**, **procedures**, and **functions**. Understanding how these components interact is essential to mastering Ada's modular and strongly typed design.

2.1 Core Program Units

In Ada, all logical units of a program are defined within packages or subprograms. A package acts as a container for related declarations and definitions, while procedures and functions are subprograms that perform specific operations.

Packages serve as the primary means of modularization and encapsulation in Ada. Each package typically consists of two parts:

- **Specification (.ads)** — Defines the interface: what the package offers to the outside world.
- **Body (.adb)** — Provides the implementation of the declared entities.

A package can contain subprograms (procedures and functions), constants, types, variables, and even other nested packages. This separation of interface and implementation allows for strong information hiding and code organization.

Example:

```
-- Package specification (like a header)
package Math_Operations is
    function Add (A, B : Integer) return Integer;
    procedure Display_Result (Value : Integer);
end Math_Operations;

-- Package body (like a source file)
package body Math_Operations is
    function Add (A, B : Integer) return Integer is
    begin
        return A + B;
    end Add;

    procedure Display_Result (Value : Integer) is
    begin
        Put_Line("Result: " & Integer'Image(Value));
    end Display_Result;
end Math_Operations;
```

This structure mirrors the concept of header/source file separation in languages like C, but with stronger type checking and compiler enforcement.

2.2 Subprograms: Procedures and Functions

Within packages, Ada defines two kinds of subprograms: **procedures** and **functions**. Both encapsulate reusable logic, but their intended use differs.

Procedures execute actions and may modify data through parameters or perform I/O, but they do not return a value. They are invoked as standalone statements.

Functions, on the other hand, compute and return a result. They are typically used within expressions and are expected to be free of side effects whenever possible.

Example of a Procedure:

```
procedure Swap (A, B : in out Integer) is
    Temp : Integer;
begin
    Temp := A;
    A := B;
    B := Temp;
end Swap;

-- Called as a standalone statement
Swap(X, Y);
```

Example of a Function:

```
function Maximum (A, B : Integer) return Integer is
begin
    if A > B then
        return A;
    else
        return B;
    end if;
end Maximum;

-- Used within an expression
Largest := Maximum(X, Y) + 10;
```

2.3 Comparing the Constructs

Feature	Package	Procedure	Function
Purpose	Groups and organizes related elements	Performs an action or operation	Computes and returns a result
Return Value	None	None	Required
Call Syntax	Not directly called	Standalone statement	Used within expressions
Parameters	N/A	`in`, `out`, `in out`	Typically `in` only
Side Effects	Possible (via internal state)	Common/Expected	Should be avoided (pure computation)
Scope / Role	Acts as a container for subprograms	Defines a reusable action	Defines a reusable computation

Table 1: Comparison of Packages, Procedures, and Functions in Ada

2.4 Additional Notes

- A package acts like a module that groups related functions and procedures together. Its `.ads` file (specification) declares what is visible externally, while the `.adb` file (body) contains the implementation.
- Procedures and functions can exist within packages or as standalone subprograms in smaller programs.
- Ada enforces that each source file name matches its main unit name (e.g., `print_roots.adb` must contain `procedure Print_Roots`).
- Functions are ideal for returning computed results, while procedures are used for actions that affect program state or output.
- In Ada, the `.ads` file plays the role of a C header, and the `.adb` file corresponds to the C source file—but Ada strictly enforces the relationship between the two.

2.5 File Organization and Compilation in Practice

When developing Ada projects, the organization of files is crucial because the GNAT compiler determines dependencies based on file names and locations.

By default, GNAT searches only the **current directory** for package specifications and bodies. This means that if your main program (`print_roots.adb`) is located in:

```
Chapter_2_Exercises/
```

and your package files are stored in a subdirectory like:

```
Chapter_2_Exercises/Package_Simple_Maths/Simple_Maths.ads
Chapter_2_Exercises/Package_Simple_Maths/Simple_Maths.adb
```

GNAT will not find them automatically.

Two solutions are common:

1. **Keep all Ada source files in one directory.** For small projects or exercises, place all `.adb` and `.ads` files together:

```
Chapter_2_Exercises/
|-- print_roots.adb
|-- simple_maths.ads
|-- simple_maths.adb
|-- simple_io.ads
‘-- simple_io.adb
```

Then compile with:

```
gnatmake print_roots.adb
```

GNAT will automatically compile all required units.

2. **Use a GNAT Project File (.gpr).** For larger or more structured projects, you can specify source directories explicitly:

```
project Chapter_2_Exercises is
   for Source_Dirs use ("./", "Package_Simple_Maths");
   for Main use ("print_roots.adb");
end Chapter_2_Exercises;

```

This allows you to maintain a clean folder structure while ensuring GNAT knows where to locate each package. Build with:

```
gnatmake -P chapter_2_exercises.gpr
```

Summary: For beginner and educational projects, keeping all files in the same folder is the simplest approach. For larger applications, a project file provides flexibility and better modular organization.