Introduction to Ada

Raphaël Amiard
Gustavo A. Hoffmann

LEARN.
ADACORE.COM

Введення в Ada

Release 2024-09

Raphaël Amiard
та Gustavo A. Hoffmann

вер. 30, 2024

Зміст:

1 Вступ **3**
- 1.1 Історія . . . 3
- 1.2 Ada сьогодні . . . 3
- 1.3 Філософія . . . 4
- 1.4 SPARK . . . 5

2 Імперативна мова **7**
- 2.1 Hello world . . . 7
- 2.2 Імперативна мова - If/Then/Else . . . 8
- 2.3 Імперативна мова - Цикли . . . 10
  - 2.3.1 Цикли For . . . 11
  - 2.3.2 Безумовний цикл . . . 12
  - 2.3.3 Цикл While . . . 13
- 2.4 Імперативна мова - Case . . . 13
- 2.5 Імперативна мова - Декларативні блоки . . . 15
- 2.6 Імперативна мова - умовні вирази . . . 16
  - 2.6.1 If вираз . . . 16
  - 2.6.2 Case вираз . . . 17

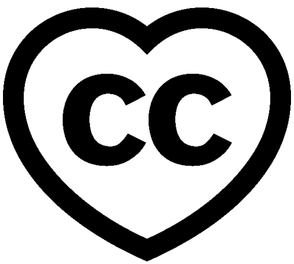
3 Підпрограми **19**
- 3.1 Підпрограми . . . 19
  - 3.1.1 Виклик підпрограм . . . 20
  - 3.1.2 Вкладені підпрограми . . . 21
  - 3.1.3 Виклик функцій . . . 22
- 3.2 Режими доступу до параметрів . . . 23
- 3.3 Виклики підпрограм . . . 24
  - 3.3.1 Параметри In . . . 24
  - 3.3.2 Параметри In out . . . 24
  - 3.3.3 Параметри Out . . . 25
  - 3.3.4 Завчасна декларація підпрограм . . . 26
- 3.4 Перенайменування . . . 27

4 Модульне програмування **29**
- 4.1 Пакети . . . 29
- 4.2 Використання пакетів . . . 31
- 4.3 Реалізація пакету . . . 31
- 4.4 Дочірні пакети . . . 33
  - 4.4.1 Дочірній пакет дочірнього пакета . . . 34
  - 4.4.2 Багато дочірніх пакетів . . . 35
  - 4.4.3 Видимість . . . 36
- 4.5 Перенайменування . . . 38

5 Суворо типізована мова **39**
5.1 Що таке тип? 39
5.2 Цілочислені типи 39
5.2.1 Семартика операцій 41
5.3 Беззнакові типи 42
5.4 Перечислення 42
5.5 Типи з плаваючою комою 43
5.5.1 Базові властивості 43
5.5.2 Точність типів з плаваючою комою 44
5.5.3 Межі типів з плаваючою комою 45
5.6 Сувора типізація 46
5.7 Похідні типи 48
5.8 Підтипи 49
5.8.1 Підтипи як псевдоніми типів 51

Copyright © 2018 – 2022, AdaCore

Цю книгу опубліковано за ліцензією CC BY-SA, що означає, що ви можете копіювати, розповсюджувати, форматувати, трансформувати та доповнювати для будь-яких цілей, навіть комерційних, за умови, що ви вказуєте належне авторство, надаєте посилання на ліцензії та вказуєте, чи були внесені зміни. Якщо ви доповнюєте, трансформуєте або використовуєте матеріал, ви повинні поширювати результат за тією ж ліцензією, що й оригінал. Ви можете знайти деталі ліцензії на цій сорінці[1]



Цей курс навчить вас основам мови програмування Ada та призначений для тих, хто вже має базові знання про техніку програмування. Ви дізнаєтеся, як застосувати ці прийоми в Ada.

Цей матеріал був написаний Raphaël Amiard та Gustavo A. Hoffmann, рецензент: Richard Kenner.

Примітка

У прикладах коду в цьому курсі використовується обмеження на 50 стовпців, що значно покращує читабельність коду на пристроях із невеликим розміром екрана. Це обмеження, однак, призводить до незвичайного стилю кодування. Наприклад замість виклику Put_Line в одну строку ми ваємо наступне:

```
Put_Line
  (" is in the northeast quadrant");
```

або так:

```
Put_Line ("  (X => "
          & Integer'Image (P.X)
          & ")");
```

Зверніть увагу, що типовий код Ada використовує обмеження щонайменше в 79 стовпців. Тому, будь ласка, не сприймайте стиль кодування з цього курсу як посилання!

Примітка

Кожен приклад коду з цієї книги має пов'язані "метадані блоку коду", які містять назву "проекту" і хеш-значення MD5. Ця інформація використовується для визначення окремого прикладу коду.

Ви можете знайти всі приклади коду в zip-файлі, який Ви можете завантажити з наступного сайту[2]. Структура каталогів у файлі zip базується на метаданих блоку коду. Наприклад, якщо ви шукаєте приклад коду з цими метаданими:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

Ви знайтидете його в цій директорії:

[1] http://creativecommons.org/licenses/by-sa/4.0

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/
cba89a34b87c9dfa71533d982d05e6ab/
```

Щоб використати цей приклад коду, просто виконайте наступні дії:

1. Розархівуйте zip файл;
2. Зайдіть в необхідну директорію;
3. Запустіть GNAT Studio в цій директорії;
4. Побудуйте (чи скомпілюйте) проект;
5. Запустіть застосунок (якщо в проекті є стартова (головна) процедура).

[2] https://learn.adacore.com/zip/learning-ada_code.zip

РОЗДІЛ 1

Вступ

1.1 Історія

У 1970-х роках Міністерство оборони США (DOD) страждало від вибухового зростання кількості мов програмування у різних проектах, в яких використовувались різні, нестандартні, діалекти або підмножини мов програмування. Міністерство вирішило вирішити цю проблему оголосивши запит на розробку единої, сучасної мови програмування. Перемогла пропозиуія від Jean Ichbiah з CII Honeywell-Bull.

Перший стандарт мови Ada був створений в 1983 який було переглянуто та вдосконалено в 1995, 2005 та 2012, кожний з яких привносив нові користні можливості.

Даний матеріал базується на стандарті 2012 року вцілому і не зосереджений на відмінностях попередніх версій.

1.2 Ada сьогодні

Сьогодні, Ada активно використовується у вбудованих системах реального часу, багато з яких є критичними з точки зору безпеки. Хоча Ada є, і може бути використана як мова загального призначення, вона повністю розкривається в низькорівневих застосунках як то:

- Вбудованих системах з жорсткими лімітами пам'яті (заборонено збирач сміття).
- Робота з обладнанням напряму.
- Системи реального часу.
- Низькорівневе системне програмування.

Специфічні галузі де використувається мова включають аерокосмос та захист, цівільна авіація, залізниця та багато інших. Ці застосунки вимагають висоеого рівня безпеки а дефекти призводять не тільки до незручностей але можуть мати важкі наслідки. Ada надає безпекові можливості які виявляють дефекти на ранній стадії — зазвияай на єтапі компіляції або за допомогою статичного аналізу. Ada також

може використовуватись для створення застосунків у багатьох інших областях, як наприклад:

- програмування відеоігор[3]
- аудіо в реальному часі[4]
- модулі ядра[5]

Це неповний список, який, сподіваюся, проливає світло на те, де Ada використовується.

З точки зору суяасних мов програмування найближчими за цілями та рівнем абстракції є C++[6] та Rust[7].

1.3 Філософія

Філософія Ada відрізняється від більшості інших мов. В основі дизайну Ada лежать наступні принципи:

- Читабельність важливіша за лаконічність. Синтаксично це виражається через факт, що перевага надається коючовим словам над символами, а ключові слова не є абревеатурами і т.д
- Дуже строгі тири. Створення нових типів дуже зручне та запобігає неправильному використанню даних.
  - Це схоже на багато інших функціональних мов за вийнятком того, що програміст повинен бути набагато чіткішим щодо введеня тексту коду, автоматичне перетворення типів заборонене.
- Явне краще, ніж неявне. Хоча це заповідь Python[8] однак Ada йде далі ніж будь яка інша мова програмування:
  - Здебільшого структурної типізації немає а більшість типів мають бути явно названі.
  - Як раніше було сказано немає автоматичного перетворення типів.
  - Семантика дуже добре визначена, а невизначена поведінка зведена до абсолютного мінімуму.
  - Програміст може надати *багато* інформаці щодо того що його код робить, як для компілятора, так і для колег. Це дозволяє компілятору бути надзвичайно корисним (читай: суворим) з програмістом.

Під час цього курсу ми пояснимо окремі мовні особливості, які є будівельними блоками для цієї філософії.

[3] https://github.com/AdaDoom3/AdaDoom3
[4] http://www.electronicdesign.com/embedded-revolution/assessing-ada-language-audio-applications
[5] http://www.nihamkin.com/tag/kernel.html
[6] https://en.wikipedia.org/wiki/C%2B%2B
[7] https://www.rust-lang.org/en-US/
[8] https://www.python.org

1.4 SPARK

Хоча цей матеріал присвячений виключно мові Ada, варто згадати, що існує інша мова, надзвичайно близька до Ada та сумісна з нею: мова SPARK.

SPARK це підмножина Ada, розроблена таким чином, щоб код, написаний у SPARK, піддавався автоматичній перевірці. Це забезпечує рівень впевненості щодо правильності коду, який набагато вищий, ніж у звичайної мови програмування.

Є відповідний курс присвячений SPARK але майте на увазі, що кожного разу, коли ми говоримо про специфікаційну потужність Ada протягом цього курсу, це потужність, яку ви можете використати в SPARK, щоб допомогти підтвердити правильність властивостей програми, починаючи від відсутності помилок під час виконання до відповідності формально визначеним функціональним вимогам.

РОЗДІЛ 2

Імперативна мова

Ada це багатопарадигмальна мова з підтримкою об'єктно-орієнтованого програмування та деяких елементів функціонального програмування, але її ядром є проста, узгоджена процедурна/імперативна мова, схожа на C або Pascal.

В інших мовах

Одна важлива відмінність між Ada та такою мовою, як C, полягає в тому, що оператори та вирази дуже чітко розрізняються. В Ada, якщо ви спробуєте використати вираз, де потрібен оператор, ваша програма не зможе скомпілюватися. Це правило підтримує корисний стилістичний принцип: вирази призначенні для отримання значень а не побічних ефектів. Це також може запобігти деяким помилкам, таким як помилкове використання оператора рівності = замість операції присвоювання := де потрібне саме присвоєння.

2.1 Hello world

Ось дуже проста імперативна програма Ada:

Listing 1: greet.adb

```
with Ada.Text_IO;

procedure Greet is
begin
   --  Вивести "Hello, World!" на екран
   Ada.Text_IO.Put_Line ("Hello, World!");
end Greet;
```

яка може знаходитися в файлі `greet.adb`.

У наведеній вище програмі є кілька цікавих речей:

- Підпрограма в Ada може бути або процедурою, або функцією. Процедура, як показано вище, не повертає значення під час виклику.

- **with** використовується для посилання на зовнішні модулі, які потрібні процедурі. Це схоже на import у інших мовах або приблизно схоже на *#include* у C та C++. Пізніше ми побачимо, як вони працюють у деталях. Тут нам потрібен модуль стандартної бібліотеки, пакет Ada.Text_IO, який містить процедуру друку тексту на екрані: Put_Line.
- Greet це процедура та основна точка входу для нашої першої програми. На відміну від C або C++, її можна назвати як завгодно. Точку входу визначить побудовник. У нашому простому прикладі **gprbuild**, побудовник GNAT, використовуватиме файл, який Ви передали як параметр.
- Put_Line це процедура, як і Greet, за винятком того, що вона оголошена в модулі Ada.Text_IO. Це еквівалент C printf.
- Коментарі починаються з -- і йдуть до кінця рядка. Синтаксису багаторядкового коментаря немає, тобто неможливо почати коментар в одному рядку з продовженням в наступному рядку. Єдиний спосіб створити кілька рядків коментарів в Ada, це використовувати -- для кожного рядка. Наприклад:

```
--  Ми починаємо коментар в цьому рядку
--  і продовжуємо в наступному
```

В інших мовах

Процедури подібні до функцій у C або C++, які повертають **void**. Пізніше ми побачимо, як оголошувати функції в Ada.

Ось мінімалістичний варіант прикладу "Hello, World":

Listing 2: greet.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4  begin
5     --  Вивести "Hello, World!" на екран
6     Put_Line ("Hello, World!");
7  end Greet;
```

Ця версія використовує ключове слово **use**, яке має форму **use** *ім'я-пакету*. Як видно з виклику Put_Line, ефект полягає в тому, що на сутності з названого пакета можна посилатися безпосередньо, без необхідності вказувати *ім'я-пакету.* попереду.

2.2 Імперативна мова - If/Then/Else

У цьому розділі описується оператор **if** і представлено кілька інших фундаментальних можливостей мови, включаючи цілочисельний ввід/вивід, оголошення змінних і режими доступу до параметрів підпрограм.

Оператор **if** не дивує за формою та функціями:

Listing 3: check_positive.adb

```
1  with Ada.Text_IO;         use Ada.Text_IO;
2  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4  procedure Check_Positive is
```

(continues on next page)

(continued from previous page)

```ada
   N : Integer;
begin
   --  Виводимо запит на екран
   Put ("Enter an integer value: ");

   --  Зчитуємо цілочисленне значення
   Get (N);

   if N > 0 then
      --  Виводимо число
      Put (N);
      Put_Line (" is a positive number");
   end if;
end Check_Positive;
```

Оператор **if** складається як мінімум із зарезервованого слова **if**, умови (яка має бути логічним значенням), зарезервованого слова **then** і непорожньої послідовності операторів (частина **then**) яка виконується, якщо умова оцінюється як Істина і завершальне **end if**.

У цьому прикладі оголошується цілочисленна змінна N, запитується у користувача ціле число, перевіряється, чи значення є додатним, і, якщо так, відображається ціле значення, за яким слідує рядок "це додатне число". Якщо значення відємне, процедура не відображає жодних результатів.

Тип Integer є попередньо визначеним типом зі знаком, і його діапазон залежить від архітектури комп'ютера. На типових сучасних процесорах ціле число має 32-розрядний знак.

Приклад ілюструє деякі базові функції для цілочисельного введення-виведення. Відповідні підпрограми знаходяться у пакеті `Ada.Integer_Text_IO` і включають процедуру `Get` (яка читає число з клавіатури) і процедуру `Put` (яка відображає ціле число).

Далі невеликі модифікації прикладу, які ілюструють оператор **if** із частиною **else**:

Listing 4: check_positive.adb

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

procedure Check_Positive is
   N : Integer;
begin
   --  Виводимо запит на екран
   Put ("Enter an integer value: ");

   --  Зчитуємо цілочисленне значення
   Get (N);

   --  Виводимо число
   Put (N);

   if N > 0 then
      Put_Line (" is a positive number");
   else
      Put_Line (" is not a positive number");
   end if;
end Check_Positive;
```

У цьому прикладі, якщо вхідне значення є відємним, програма відображає значення, за яким слідує рядок "не є додатним числом".

Останній варіант ілюструє оператор **if** з декількома **elsif**:

Listing 5: check_direction.adb

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

procedure Check_Direction is
   N : Integer;
begin
   Put ("Enter an integer value: ");
   Get (N);
   Put (N);

   if N = 0 or N = 360 then
      Put_Line (" is due north");
   elsif N in 1 .. 89 then
      Put_Line (" is in the northeast quadrant");
   elsif N = 90 then
      Put_Line (" is due east");
   elsif N in 91 .. 179 then
      Put_Line (" is in the southeast quadrant");
   elsif N = 180 then
      Put_Line (" is due south");
   elsif N in 181 .. 269 then
      Put_Line (" is in the southwest quadrant");
   elsif N = 270 then
      Put_Line (" is due west");
   elsif N in 271 .. 359 then
      Put_Line (" is in the northwest quadrant");
   else
      Put_Line (" is not in the range 0..360");
   end if;
end Check_Direction;
```

У цьому прикладі очікується, що користувач введе ціле число від 0 до 360 включно, і відображається, якому квадранту чи осі відповідає значення. Оператор **in** перевіряє, чи знаходиться скалярне значення в заданому діапазоні, і повертає логічний результат. Ефект від програми має бути зрозумілим; пізніше ми побачимо альтернативний і ефективніший стиль для досягнення того самого ефекту за допомогою оператора **case**.

Ключове слово **elsif** відрізняється від C або C++, де замість нього використовуються вкладені блоки **else .. if**. І ще одна відмінність полягає в наявності **end if**, що дозволяє уникнути проблеми, відомої як «висячий else».

2.3 Імперативна мова - Цикли

Ada має три способи визначення циклів. Вони відрізняються від циклів C / Java / Javascript, проте більш простим синтаксисом і семантикою відповідно до філософії Ada.

2.3.1 Цикли For

Першим типом циклу є цикл **for**, який дозволяє виконувати ітерацію в дискретному діапазоні.

Listing 6: greet_5a.adb

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

procedure Greet_5a is
begin
   for I in 1 .. 5 loop
      --  Виклик процедури Put_Line
      Put_Line ("Hello, World!"
                & Integer'Image (I));
      --        ^ Procedure parameter
   end loop;
end Greet_5a;
```

Кілька речей, на які варто звернути увагу:

- `1 .. 5` це дискретний діапазон від `1` до `5` включно.
- Параметр циклу `I` (назва довільна) у тілі циклу має значення в цьому діапазоні.
- `I` є локальним для циклу, тому ви не можете посилатися на `I` поза циклом.
- Хоча значення `I` збільшується на кожній ітерації, з точки зору коду всередині циклу воно є постійним. Змінювати його неможна. Спроба змінити його значення призведе до помилки компіляції.
- `Integer'Image` це функція, яка приймає ціле число та перетворює його на `String`. Це приклад мовної конструкції, відомої як *атрибут*, позначений синтаксисом `'`, який буде розглянуто більш детально пізніше.
- Символ `&` є оператором конкатенації для строкових значень
- `end loop` позначає кінець циклу

«Крок» циклу обмежений 1 (напрямок вперед) і -1 (назад). Щоб виконати ітерацію в діапазоні назад, використовуйте ключове слово **reverse**:

Listing 7: greet_5a_reverse.adb

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

procedure Greet_5a_Reverse is
begin
   for I in reverse 1 .. 5 loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));
   end loop;
end Greet_5a_Reverse;
```

Межі циклу **for** можуть бути обчислені під час виконання; вони обчислюються один раз перед виконанням тіла циклу. Якщо значення верхньої межі менше значення нижньої, то цикл не виконується взагалі. Це також стосується циклів **reverse**. Таким чином, у наступному прикладі виводу на екран не буде:

Listing 8: greet_no_op.adb

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

procedure Greet_No_Op is
```

(continues on next page)

(continued from previous page)

```
begin
   for I in reverse 5 .. 1 loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));
   end loop;
end Greet_No_Op;
```

Докладніше про цикл **for** буде підніше.

2.3.2 Безумовний цикл

Найпростішим циклом в Ada є безумовний цикл, який є основою для інших типів циклів Ada.

Listing 9: greet_5b.adb

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

procedure Greet_5b is
   --  Оголошуємо змінну:
   I : Integer := 1;
   --  ^ Тип
   --             ^ Начальне значення
begin
   loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));

      --  Вихід із циклу:
      exit when I = 5;
      --         ^ Умова

      --  Присвоєння:
      I := I + 1;
      --  Конструкції як в C 'I++' немає
   end loop;
end Greet_5b;
```

Цей приклад дає той самий ефект, що й `Greet_5a`, показаний раніше.

Він ілюструє кілька концепцій:

- Ми оголосили змінну з назвою `I` між **is** і **begin**. Це являє собою *декларативний блок*. Ada чітко відокремлює декларативну область від суто коду підпрограми. Оголошення може відбуватися в декларативній області, але не допускається посеред коду.
- Цикл починається ключовим словом **loop** і, як і будь-який тип циклу, завершується комбінацією ключових слів **end loop**. Сам по собі це нескінченний цикл. Ви можете вийти з цього за допомогою оператора **exit**
- Синтаксис присвоєння: `:=`, а рівності — `=`. Немає способу сплутати їх, оскільки, як зазначалося раніше, в Ada твердження та вирази є різними, а вирази не є дійсними твердженнями.

2.3.3 Цикл While

Останній вид циклу в Ada — це цикл **while**.

Listing 10: greet_5c.adb

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

procedure Greet_5c is
   I : Integer := 1;
begin
   --  Умова має бути логічним виразом
   --  (не численним).
   --  Оператор "<=" вертає результат
   --  порівняння
   while I <= 5 loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));

      I := I + 1;
   end loop;
end Greet_5c;
```

Умова оцінюється перед кожною ітерацією. Якщо результат хибний, то цикл припиняється.

Ця програма робить то самє, що й попередні.

В інших мовах

Зауважте, що Ada має іншу семантику, ніж мови на основі C щодо умови в циклі while. В Ada умова має бути логічним значенням, інакше компілятор відхилить програму; умова не є цілим числом, яке розглядається як **True** або **False** залежно від того, відмінне воно від нуля чи ні.

2.4 Імперативна мова - Case

Оператор Ada **case** схожий на оператор C і C++ **switch**, але з деякими важливими відмінностями.

Ось приклад, варіація програми, яка була показана раніше з оператором **if**:

Listing 11: check_direction.adb

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

procedure Check_Direction is
   N : Integer;
begin
   loop
      Put ("Enter an integer value: ");
      Get (N);
      Put (N);

      case N is
         when 0 | 360 =>
            Put_Line
```

(continues on next page)

(continued from previous page)

```
15              (" is due north");
16        when 1 .. 89 =>
17           Put_Line
18              (" is in the northeast quadrant");
19        when 90 =>
20           Put_Line
21              (" is due east");
22        when 91 .. 179 =>
23           Put_Line
24              (" is in the southeast quadrant");
25        when 180 =>
26           Put_Line
27              (" is due south");
28        when 181 .. 269 =>
29           Put_Line
30              (" is in the southwest quadrant");
31        when 270 =>
32           Put_Line
33              (" is due west");
34        when 271 .. 359 =>
35           Put_Line
36              (" is in the northwest quadrant");
37        when others =>
38           Put_Line
39              (" Au revoir");
40           exit;
41      end case;
42   end loop;
43 end Check_Direction;
```

Ця програма постійно запитує ціле число, а потім, якщо значення знаходиться в діапазоні `0 .. 360`, відображає відповідний квадрант або вісь. Якщо значення виходить за межі цього діапазону, цикл (і програма) припиняється після виведення прощального повідомлення.

Ефект оператора case подібний до оператора if у попередньому прикладі, але оператор case може бути ефективнішим, оскільки він не передбачає багаторазових перевірок діапазону.

Важливі моменти щодо конструкції case:

- Вираз для case (тут змінна N) має бути дискретного типу, тобто або цілого типу, або типу перерахування. Дискретні типи будуть розглянуті більш детально пізніше *дискретні типи* (page 39).
- Кожне можливе значення виразу case має бути охоплено одною з гілок оператора case. Це буде перевірено під час компіляції.
- Гілка може охоплювати одне значення, наприклад `0`; діапазон значень, наприклад `1 .. 89`; або будь-яка комбінація значень (розділених символом |).
- Як особливий випадок, необов'язкова кінцева гілка може вказати **others**, яка охоплює всі інші значення, не включені в попередні гілки.
- Виконання складається з оцінки виразу case, а потім передачі керування коду в унікальній гілці, яка охоплює це значення.
- Коли виконання коду у вибраній гілці завершено, керування передається коду після **end case**. На відміну від C, виконання не переходить до наступної гілки. Отже, Ada не потребує (і не має) оператора **break**.

2.5 Імперативна мова - Декларативні блоки

Як згадувалося раніше, Ada проводить чіткий синтаксичний розподіл між деклараціями, які вводять імена для сутностей, які використовуватимуться в програмі, та операторами, які виконують обробку. Області в програмі, де можуть з'являтися декларації, називаються декларативними блоками.

У будь-якій підпрограмі розділ між **is** і **begin** є декларативним блоком. Ви можете оголосити там змінні, константи, типи, внутрішні підпрограми та інші сутності.

Ми коротко згадували про оголошення змінних у попередньому підрозділі. Давайте розглянемо простий приклад, де ми оголошуємо цілочисельну змінну X у декларативному блоці та виконуємо її ініціалізацію та модифікацію:

Listing 12: main.adb

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

procedure Main is
   X : Integer;
begin
   X := 0;
   Put_Line ("The initial value of X is "
             & Integer'Image (X));

   Put_Line ("Performing operation on X...");
   X := X + 1;

   Put_Line ("The value of X now is "
             & Integer'Image (X));
end Main;
```

Давайте розглянемо ще один приклад:

Listing 13: main.adb

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

procedure Main is
   procedure Nested is
   begin
      Put_Line ("Hello World");
   end Nested;
begin
   Nested;
   --  Викликали процедуру Nested
end Main;
```

Оголошення неможливі посеред операторів. Якщо вам потрібно оголосити локальну змінну серед операторів, ви можете оголосити новий декларативний блок за допомогою оператора **declare**:

Listing 14: greet.adb

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

procedure Greet is
begin
   loop
      Put_Line ("Please enter your name: ");

```

(continues on next page)

(continued from previous page)

```
      declare
         Name : String := Get_Line;
         --               ^ Виклик функції
         --                 Get_Line
      begin
         exit when Name = "";
         Put_Line ("Hi " & Name & "!");
      end;

      --  Змінної Name більше не існує
   end loop;

  Put_Line ("Bye!");
end Greet;
```

Увага

Функція Get_Line дозволяє вам отримувати дані від користувача та отримати строку як результат. Це більш-менш еквівалентно функції C:c:*scanf*.

Вона повертає **String**, який, як ми побачимо пізніше, є Необмеженим масивом[9]. Наразі ми просто зауважимо, що якщо ви бажаєте оголосити змінну **String** і не знаєте її розмір заздалегідь, вам потрібно ініціалізувати змінну під час її оголошення.

2.6 Імперативна мова - умовні вирази

Ada 2012 додала аналог виразу для умовних операторів (**if** і **case**).

2.6.1 If вираз

Ось альтернативна версія прикладу, який ми бачили раніше; оператор **if** було замінено виразом **if**:

Listing 15: check_positive.adb

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

procedure Check_Positive is
   N : Integer;
begin
   Put ("Enter an integer value: ");
   Get (N);
   Put (N);

   declare
      S : constant String :=
        (if N > 0
           then " is a positive number"
           else " is not a positive number");
   begin
      Put_Line (S);
```

(continues on next page)

[9] https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html#intro-ada-unconstrained-array-types

(continued from previous page)

```
   end;
end Check_Positive;
```

Вираз **if** використовує один із двох строк залежно від N і присвоює це значення локальній змінній S.

Вирази Ada **if** подібні до операторів **if**. Однак є кілька відмінностей, які пов'язані з тим, що це вираз:

- Обидва варіанти мають бути одного типу
- Він *має* бути в дужках, якщо навколишній вираз їх ще не містить
- Гілка **else** є обов'язковою, якщо вираз після **then** не є логічним типом. У цьому випадку гілка **else** є необов'язковою і, якщо її немає, за замовчуванням використовується **else True**.

Ось інший приклад:

Listing 16: main.adb

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

procedure Main is
begin
   for I in 1 .. 10 loop
      Put_Line (if I mod 2 = 0
                then "Even"
                else "Odd");
   end loop;
end Main;
```

Ця програма виводить 10 рядків, чергуючи «Непарні» та «Парні».

2.6.2 Case вираз

Подібно до виразів **if**, Ada також має вирази **case**. Вони працюють так, як Ви очікуєте.

Listing 17: main.adb

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

procedure Main is
begin
   for I in 1 .. 10 loop
      Put_Line
        (case I is
         when 1 | 3 | 5 | 7 | 9  => "Odd",
         when 2 | 4 | 6 | 8 | 10 => "Even");
   end loop;
end Main;
```

Ця програма робить то саме, що й попередній приклад.

Синтаксис відрізняється від оператора **case**, в даному випадку варіанти розділені комами.

РОЗДІЛ 3

Підпрограми

3.1 Підпрограми

Досі ми використовували процедури, здебільшого, щоб мати головну підпрограму для виконання. Процедури є одним із видів *підпрограм*.

В Ada є два типи підпрограм: *функції* та *процедури*. Різниця між ними полягає в тому, що функція повертає результат, а процедура – ні.

У наступному прикладі показано специфікацію та реалізацію функції:

Listing 1: increment.ads

```
function Increment (I : Integer) return Integer;
```

Listing 2: increment.adb

```
--  Оголошення (не реалізація) функції з
--  одним параметром, яка повертає цілочисленне
--  значення

function Increment (I : Integer) return Integer is
   --  Тут функція реалізовується
begin
    return I + 1;
end Increment;
```

Підпрограми в Ada, зазвичай, мають параметри. Одне важливе синтаксичне зауваження полягає в тому, що підпрограма, яка не має параметрів, взагалі не має розділу для параметрів, наприклад:

```
procedure Proc;

function Func return Integer;
```

Зверніть увагу на відсутність дужок.

Ось ще один приклад підпрограми:

Listing 3: increment_by.ads

```
function Increment_By
  (I    : Integer := 0;
   Incr : Integer := 1) return Integer;
--                ^ Значення параметрів
--                  за замовчуванням
```

У цьому прикладі ми бачимо, що параметри можуть мати значення за замовчуванням. Під час виклику підпрограми ви можете не вказувати параметри, якщо вони мають такі значення. На відміну від C/C++, виклик підпрограми без параметрів не містить дужок.

Це реалізація цієї функції:

Listing 4: increment_by.adb

```
function Increment_By
  (I    : Integer := 0;
   Incr : Integer := 1) return Integer is
begin
   return I + Incr;
end Increment_By;
```

В інструментах GNAT

Стандарт Ada не визначає, у якому файлі має зберігатися специфікація або реалізація підпрограми. Іншими словами, стандарт не вимагає певної структури файлів або певних розширень імен файлів. Наприклад, ми могли б зберегти і специфікацію, і реалізацію функції `Increment` у файлі під назвою `increment.txt` (ми навіть могли б зберігати весь код застосунку в одному файлі). З точки зору стандарту, це було б цілком прийнятно.

Проте для інструментів GNAT потрібна така схема іменування файлів:

- файли з розширенням *.ads* містять специфікацію, а
- файли з розширенням *.adb* містять реалізацію.

Таким чином, для інструментів GNAT специфікація функції `Increment` має зберігатися у файлі `increment.ads`, тоді як її реалізація має зберігатися у файлі `increment.adb`. Це правило завжди стосується пакетів, які ми обговоримо *пізніше* (page 29) (однак зауважте, що це правило можна обійти). Для отримання додаткової інформації ви можете звернутися до курсу Вступ до інструментів GNAT або до Посібник користувача GPRbuild[10].

3.1.1 Виклик підпрограм

Ми можемо визвати нашу підпрограму таким чином:

Listing 5: show_increment.adb

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

procedure Show_Increment is
   A, B, C : Integer;
```

(continues on next page)

[10] https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html

(continued from previous page)

```
begin
   C := Increment_By;
   --              ^ Виклик без параметрів
   --                значення I є 0
   --                та Incr є 1

   Put_Line ("Using defaults for Increment_By is "
             & Integer'Image (C));

   A := 10;
   B := 3;
   C := Increment_By (A, B);
   --                   ^ Виклик з параметрами

   Put_Line ("Increment of "
             & Integer'Image (A)
             & " with "
             & Integer'Image (B)
             & " is "
             & Integer'Image (C));

   A := 20;
   B := 5;
   C := Increment_By (I    => A,
                      Incr => B);
    --                ^ Виклик з іменуванням
    --                  параметрів

   Put_Line ("Increment of "
             & Integer'Image (A)
             & " with "
             & Integer'Image (B)
             & " is "
             & Integer'Image (C));
end Show_Increment;
```

Ada дозволяє вам передавати параметри за іменами, незалежно від того, чи мають вони значення за замовчуванням чи ні. Проте є деякі правила:

- Не іменовані параметри ідуть в порядку об'явлення.
- Після іменованих параметрів не можуть ідти порядкові.

Зазвичай, параметри іменують при виклику, якщо відповідні параметри підпрограми мають значення за замовчуванням. Однак також цілком прийнятно іменувати кожен параметр, якщо це робить код зрозумілішим.

3.1.2 Вкладені підпрограми

Як коротко згадувалося раніше, Ada дозволяє вам оголошувати одну підпрограму всередині іншої.

Це корисно з двох причин:

- Це дозволяє зробити ваші програми структурованішими. Якщо вам потрібна підпрограма тільки як «помічник» для іншої підпрограми, то принцип локалізації вказує на те, що допоміжну підпрограму слід зробити вкладеною.
- Це дозволяє вам легко надавати доступ та контролювати його, оскільки вкладені підпрограми мають доступ до параметрів, а також будь-яких локальних змінних, оголошених до вкладеної підпрограми.

У попередньому прикладі ми можемо перемістити дубльований код (виклик Put_Line) до окремої процедури. Ось версія з вкладеною процедурою Display_Result:

Listing 6: show_increment.adb

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

procedure Show_Increment is
   A, B, C : Integer;

   procedure Display_Result is
   begin
      Put_Line ("Increment of "
                & Integer'Image (A)
                & " with "
                & Integer'Image (B)
                & " is "
                & Integer'Image (C));
   end Display_Result;

begin
   A := 10;
   B := 3;
   C := Increment_By (A, B);
   Display_Result;
   A := 20;
   B := 5;
   C := Increment_By (A, B);
   Display_Result;
end Show_Increment;
```

3.1.3 Виклик функцій

Важливою особливістю викликів функцій в Ada є те, що зезультат, що повертається під час виклику, не можна ігнорувати; тобто виклик функції не можна використовувати як оператор.

Якщо ви хочете викликати функцію і не потребуєте її результату, вам усе одно потрібно буде явно зберегти її в локальній змінній.

Listing 7: quadruple.adb

```
function Quadruple (I : Integer)
                    return Integer is

    function Double (I : Integer)
                     return Integer is
    begin
       return I * 2;
    end Double;

   Res : Integer := Double (Double (I));
   --                ^ Виклик Double
begin
   Double (I);
   --  ПОМИЛКА: не можна викликати функцію
   --           "Double" як оператор

```

(continues on next page)

(continued from previous page)

```
17    return Res;
18 end Quadruple;
```

В інструментах GNAT

У GNAT, коли всі попередження активовані, стає ще важче ігнорувати результат функції, оскільки невикористані змінні будуть позначені. Наприклад, цей код буде невірним:

```
function Read_Int
   (Stream :     Network_Stream;
    Result : out Integer) return Boolean;

procedure Main is
    Stream : Network_Stream := Get_Stream;
    My_Int : Integer;

    -- Попередження в рядку нижче:
    --    Значення B не використовується
    B : Boolean := Read_Int (Stream, My_Int);
begin
   null;
end Main;
```

Тоді у вас є два варіанти, щоб вимкнути це попередження:

- Або додайте до змінної **pragma** *Unreferenced*, наприклад:

```
B : Boolean := Read_Int (Stream, My_Int);
pragma Unreferenced (B);
```

- Або використайте одне з імен `discard` `dummy` `ignore` `junk` `unused` (незалежно від регістру) як ім'я змінної.

Будь який з варіанів означає що Ви свідомо і в "ручну" вимикаєте таку перевірку для цього конкретного місця (гарантуючи його корректність).

3.2 Режими доступу до параметрів

Поки що ми бачили, що Ada є мовою, орієнтованою на безпеку. Є багато способів це реалізувати. Розглянемо два важливі моменти:

- Ada змушує програміста вказати якомога більше про очікувану поведінку програми, щоб компілятор міг попередити або відхилити, якщо є невідповідність.
- Ada надає різноманітні методи для досягнення одноманітності та гнучкості посилань і динамічного керування пам’яттю, але без недоліків таких як витік пам’яті та завислі посилання.

Режими параметрів — це можливість, яка допомагає досягти двох цілей проектування, наведених вище. Параметр підпрограми можна обмежити за допомогою режиму доступу, який є одним із наведених:

in	Тільки для зчитування, зміна не можлива
out	Запис можливий, значення не існує до запису
in out	Зчитування та запис

Режимом за замовчуванням (коли нічого не вказано) для параметрів є **in**; досі більшість прикладів використовували параметри в режимі доступу **in**.

> **Історично**
>
> Функції та процедури спочатку мали більше відмінностей. До Ada 2012 функції могли мати лише параметри **in**.

3.3 Виклики підпрограм

3.3.1 Параметри In

Перший режим для параметрів - це той, який ми неявно використовували досі. Параметри, передані за допомогою цього режиму, не можна змінювати, тому наступна програма викличе помилку:

Listing 8: swap.adb

```ada
procedure Swap (A, B : Integer) is
   Tmp : Integer;
begin
   Tmp := A;

   --  ПОМИЛКА: зміна "in" параметра
   --          заборонена
   A := B;

   --  ПОМИЛКА: зміна "in" параметра
   --          заборонена
   B := Tmp;
end Swap;
```

Той факт, що **in** є режимом за замовчуванням, дуже важливий. Це означає, що параметр не буде змінено, якщо ви явно не вкажете режим, у якому це дозволено.

3.3.2 Параметри In out

Розглянемо наступний приклад з використанням **in out** режиму доступу:

Listing 9: in_out_params.adb

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

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

   A : Integer := 12;
   B : Integer := 44;
begin
```

(continues on next page)

(continued from previous page)

```
15      Swap (A, B);
16
17      --  Виведе на екран 44
18      Put_Line (Integer'Image (A));
19   end In_Out_Params;
```

Параметр **in out** дозволено читати та записувати, тому у наведеному вище прикладі ми бачимо, що значення A змінено після виклику Swap .

> **Увага**
>
> Хоча параметри **in out** виглядають дещо як посилання в C++ або звичайні параметри в Java, які передаються за посиланням, стандарт мови Ada не вимагає передачі таких параметрів «за посиланням», за винятком певних категорій типів, як буде пояснено пізніше.
>
> Загалом, краще розглядати режими ширше, ніж семантику за значенням і за посиланням. Для компілятора це означає, що масив, переданий як параметр **in**, може бути переданий за посиланням, оскільки це більш ефективно (що нічого не змінює для користувача, оскільки параметр не можна змінити). Однак параметр дискретного типу завжди передаватиметься через копію, незалежно від його режиму (який більш ефективний у більшості архітектур).

3.3.3 Параметри Out

Режим **out** застосовується, коли підпрограмі потрібно записати параметр, який може бути неініціалізованим у точці виклику. Читання значення параметра **out** дозволено, але це слід робити лише після того, як сама підпрограма призначить йому значення. Такі параметри подібні до результатів, що повертаються функціями. Коли підпрограма завершується, фактичне значення буде тим, що у параметра було в точці повернення.

> **В інших мовах**
>
> Ada не має конструкції яка дозволяє повертати кілька значень із функції (за винятком використання типу - запису/record). Отже, спосіб повернути кілька значень із підпрограми — це використовувати параметри **out**.

Наприклад, підпрограми, які читають цілі числа з мережі, може мати одну з наступних специфікацій:

```
procedure Read_Int
   (Stream  :     Network_Stream;
    Success : out Boolean;
    Result  : out Integer);

function Read_Int
   (Stream :     Network_Stream;
    Result : out Integer) return Boolean;
```

Зчитування змінної з таким режимом доступу перед записом у неї в ідеалі має викликати помилку, яка, як правило, спричинить або неефективні перевірки під час виконання, або складні правила під час компіляції. Отже, з точки зору користувача такий параметр діє як неініціалізована змінна, коли підпрограма викликається.

В інструментах GNAT

GNAT виявить прості випадки неправильного використання таких параметрів. Наприклад, компілятор видасть попередження для наступної процедури:

Listing 10: outp.adb

```
procedure Outp is
   procedure Foo (A : out Integer) is
      B : Integer := A;
      --             ^ Попередження про доступ
      --               до неініціалізованої A
   begin
      A := B;
   end Foo;
begin
   null;
end Outp;
```

3.3.4 Завчасна декларація підпрограм

Як ми бачили раніше, підпрограму можна декларувати окремо від реалізації. Це можливо загалом і може бути корисним, якщо вам потрібно, щоб підпрограми були взаємно рекурсивними (викликають одна-одну), як у прикладі нижче:

Listing 11: mutually_recursive_subprograms.adb

```
procedure Mutually_Recursive_Subprograms is
    procedure Compute_A (V : Natural);
    --  Декларація Compute_A

    procedure Compute_B (V : Natural) is
    begin
       if V > 5 then
          Compute_A (V - 1);
          --  Виклик Compute_A
       end if;
    end Compute_B;

    procedure Compute_A (V : Natural) is
    begin
       if V > 2 then
          Compute_B (V - 1);
          --  Виклик Compute_B
       end if;
    end Compute_A;
begin
   Compute_A (15);
end Mutually_Recursive_Subprograms;
```

3.4 Перенайменування

Підпрограми можна перейменувати за допомогою ключового слова **renames** і вказання нової назви для неї:

```
procedure New_Proc renames Original_Proc;
```

Це може бути корисно, наприклад, для покращення читабельності вашої програми, коли ви використовуєте код із зовнішніх джерел, який не можна змінити у вашій системі. Давайте розглянемо приклад:

Listing 12: a_procedure_with_very_long_name_that_cannot_be_changed.ads

```
procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
   (A_Message : String);
```

Listing 13: a_procedure_with_very_long_name_that_cannot_be_changed.adb

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

 procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
   (A_Message : String) is
 begin
    Put_Line (A_Message);
 end A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
```

Як випливає з формулювання в назві процедури вище, ми не можемо змінити її назву. Однак ми можемо перейменувати його на щось на зразок Show у нашій програмі та використати це коротше ім'я. Зауважте, що ми також повинні оголосити всі параметри оригінавльної підпрограми. Ми також можемо перейменувати їх. Наприклад:

Listing 14: show_renaming.adb

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

 procedure Show_Renaming is

    procedure Show (S : String) renames
      A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

 begin
    Show ("Hello World!");
 end Show_Renaming;
```

Зауважте, що оригінальна назва (A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed) все ще може бути використана після оголошення процедури Show.

Ми також можемо перейменувати підпрограми зі стандартної бібліотеки. Наприклад, ми можемо перейменувати **Integer**'Image на Img:

Listing 15: show_image_renaming.adb

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

procedure Show_Image_Renaming is

   function Img (I : Integer) return String
     renames Integer'Image;

```

(continues on next page)

(continued from previous page)

```
8  begin
9     Put_Line (Img (2));
10    Put_Line (Img (3));
11 end Show_Image_Renaming;
```

Перейменування також дозволяє нам вводити значення за замовчуванням для параметрів, які були недоступні в оригінальній декларації. Наприклад, ми можемо вказати `"Hello World!"` як значення за замовчуванням для параметра **String** процедури Show:

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming_Defaults is

   procedure Show (S : String := "Hello World!")
     renames
       A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

begin
   Show;
end Show_Renaming_Defaults;
```

РОЗДІЛ 4

Модульне програмування

Поки що наші приклади були простими автономними підпрограмами. Ada корисна в цьому відношенні, оскільки вона дозволяє використовувати декларативні блоки. Таким чином ми змогли оголосити наші типи та змінні в тілах основних підпрограм.

Однак легко побачити, що це не буде масштабуватися для реальних програм. Нам потрібен кращий спосіб структурувати наші програми в модулі та окремі одиниці.

Ada заохочує поділ програм на кілька пакетів і підпакетів, надаючи багато інструментів програмісту, який шукає як ідеально організовати код.

4.1 Пакети

Ось приклад оголошення пакета в Ada:

Listing 1: week.ads

```
package Week is

   Mon : constant String := "Monday";
   Tue : constant String := "Tuesday";
   Wed : constant String := "Wednesday";
   Thu : constant String := "Thursday";
   Fri : constant String := "Friday";
   Sat : constant String := "Saturday";
   Sun : constant String := "Sunday";

end Week;
```

І ось як ви його використовувати:

Listing 2: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Week;
--  Посилання на пакет Week і
--  додавання залежності Main від Week
```

(continues on next page)

(continued from previous page)

```
5
6  procedure Main is
7  begin
8     Put_Line ("First day of the week is "
9               & Week.Mon);
10 end Main;
```

Пакети дозволяють зробити Ваш код модульним, розділяючи ваші програми на семантично значущі одиниці. Крім того, відокремлення специфікації пакета від його тіла (що ми побачимо нижче) може зменшити час компіляції.

Хоча ключове слово **with** вказує на залежність, ви можете бачити в прикладі вище, що вам все одно потрібно додавати назву пакету як префікс до посилань на сутності з пакету Week. Якби ми викорастали також **use** Week, тоді такий префікс не був би потрібним.

Для доступу до сутностей із пакета використовується нотація з крапками `A.B`, яка є такою ж нотацією, як і для доступу до полів запису.

Ключове слово **with** може з'являтися *тільки* перед блоком компіляції (тобто перед зарезервованим словом, таким як **procedure** або **package**, яке позначає початок блоку). Пізніше це неможливо. Це правило потрібне лише з методологічних міркувань: людина, яка читає Ваш код, повинна мати змогу одразу бачити, від яких пакетів залежить код.

В інших мовах

Пакети виглядають схожими на файли заголовків у C/C++, але семантично сильно від них відрізняються.

- Перша і найважливіша відмінність полягає в тому, що пакети є механізмом на рівні мови. Це на відміну від файлу заголовка *#include*'d, який є функціональністю препроцесора C.
- Негайним наслідком є те, що конструкція **with** є механізмом семантичного включення, а не механізмом включення тексту коду. Отже, коли ви використовуєте **with**, ви говорите компілятору "Я залежу від цієї семантичної одиниці", а не "включаю цей текст на це місце".
- Таким чином, сам пакет не змінюється залежно від того, де він був включений. Порівняйте це з C/C++, де значення включеного коду залежить від контексту, у якому з'являється *#include*.

  Це дозволяє компіляції/повторній компіляції бути більш ефективними. Це також дозволяє таким інструментам, як IDE, мати правильну інформацію про семантику програми. У свою чергу, це дозволяє покращити інструменти в цілому та код, який краще аналізується навіть людьми.

Важливою перевагою Ada **with** порівняно з *#include* є відсутність стану. Порядок речень **with** і **use** не має значення та може бути змінений без побічних ефектів.

В інструментах GNAT

Стандарт мови Ada не вимагає жодних особливих відносин між файлами коду та пакетами; наприклад, теоретично ви можете помістити весь свій код в один файл або використовувати власні угоди про іменування файлів. Однак на практиці реалізація матиме певні правила. З GNAT кожна одиниця компіляції верхнього рівня повинна мати окремий файл. У наведеному вище прикладі пакет `Week` буде у файлі `.ads` (для специфікації Ada), а процедура `Main` буде у файлі `.adb`. (для тіла

Ada).

4.2 Використання пакетів

Як ми бачили вище, ключове слово **with** вказує на залежність від іншого пакета. Однак кожне посилання на сутність, що надходить із пакета Week, має мати префікс повної назви пакета. Можна зробити кожну сутність пакета видимою безпосередньо в поточній області за допомогою ключового слова **use**.

Фактично, ми використовували **use** майже з початку цього курсу.

Listing 3: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
--                ^ Робимо кожну сутність
--                  з пакету Ada.Text_IO
--                  безпосередньо видимою.
with Week;

procedure Main is
   use Week;
   --  Робимо кожну сутність з пакету Week
   --  Week безпосередньо видимою.
begin
   Put_Line ("First day of the week is " & Mon);
end Main;
```

Як ви можете бачити в прикладі вище:

- Put_Line — це підпрограма, яка походить з пакету Ada.Text_IO. Ми можемо посилатися на неї безпосередньо, тому що у нас є **use**d у верхній частині блоку Main.
- На відміну від **with**, **use** можна розмістити або перед блоком компіляції, або в будь-якому декларативному блоці. В останньому випадку **use** матиме ефект у межах цього блоку.

4.3 Реалізація пакету

У наведеному вище простому прикладі пакет Week містить лише декларації, а не реалізації. Це не помилка: у специфікації пакета, яка проілюстрована вище, ви не можете розміщати код реалізації. Вони мають бути в реалізації пакету.

Listing 4: operations.ads

```
package Operations is

   --  Декларація
   function Increment_By
     (I    : Integer;
      Incr : Integer := 0) return Integer;

   function Get_Increment_Value return Integer;

end Operations;
```

Listing 5: operations.adb

```ada
package body Operations is

   Last_Increment : Integer := 1;

   --  Реалізація
   function Increment_By
     (I    : Integer;
      Incr : Integer := 0) return Integer is
   begin
      if Incr /= 0 then
         Last_Increment := Incr;
      end if;

      return I + Last_Increment;
   end Increment_By;

   function Get_Increment_Value return Integer is
   begin
      return Last_Increment;
   end Get_Increment_Value;

end Operations;
```

Тут ми бачимо, що реалізація функції `Increment_By` має бути також декларована і в реалізації пакету (тілі пакету). За збігом обставин, це дозволяє нам помістити змінну `Last_Increment` в тіло, і зробити їх недоступною для користувача пакета `Operations`, забезпечуючи першу форму інкапсуляції.

Це працює, оскільки сутності, оголошені в тілі, видимі *лише* в тілі.

Наступний приклад показує, як `Last_Increment` використовується:

Listing 6: main.adb

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

procedure Main is
   use Operations;

   I : Integer := 0;
   R : Integer;

   procedure Display_Update_Values is
      Incr : constant Integer :=
               Get_Increment_Value;
   begin
      Put_Line (Integer'Image (I)
                & " incremented by "
                & Integer'Image (Incr)
                & " is "
                & Integer'Image (R));
      I := R;
   end Display_Update_Values;
begin
   R := Increment_By (I);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;

```

(continues on next page)

(continued from previous page)

```
   R := Increment_By (I, 5);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;

   R := Increment_By (I, 10);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;
end Main;
```

4.4 Дочірні пакети

З пакетів можна створити ієрархію. Ми досягаємо цього за допомогою дочірніх пакетів, які розширюють функціональні можливості батьківського пакета. Одним із прикладів дочірнього пакету, який ми використовували досі, є пакет Ada.Text_IO. Тут батьківський пакет називається Ada, а дочірній — Text_IO. У попередніх прикладах ми використовували процедуру Put_Line з дочірнього пакету Text_IO.

Важливо

Ada також підтримує вкладені пакети. Однак, оскільки вони можуть бути складнішими у використанні, рекомендується замість них використовувати дочірні пакети. Вкладені пакети будуть розглянуті в поглибленому курсі.

Давайте почнемо обговорення дочірніх пакетів, взявши наш попередній пакет Week:

Listing 7: week.ads

```
package Week is

   Mon : constant String := "Monday";
   Tue : constant String := "Tuesday";
   Wed : constant String := "Wednesday";
   Thu : constant String := "Thursday";
   Fri : constant String := "Friday";
   Sat : constant String := "Saturday";
   Sun : constant String := "Sunday";

end Week;
```

Якщо ми хочемо створити дочірній пакет для Week, ми можемо написати:

Listing 8: week-child.ads

```
package Week.Child is

   function Get_First_Of_Week return String;

end Week.Child;
```

Тут Week — батьківський пакет, а Child — дочірній пакет. Це відповідна реалізація пакета Week.Child:

Listing 9: week-child.adb

```
package body Week.Child is

   function Get_First_Of_Week return String is
   begin
      return Mon;
   end Get_First_Of_Week;

end Week.Child;
```

У реалізації функції Get_First_Of_Week ми можемо використовувати Mon безпосередньо, навіть якщо він був оголошений у батьківському пакеті Week. Ми не пишемо тут **with** Week, оскільки всі елементи зі специфікації пакета Week — наприклад Mon, Tue тощо — видимі в дочірньому пакеті Week.Child.

Тепер, коли ми завершили реалізацію пакета Week.Child, ми можемо використовувати елементи цього дочірнього пакета в підпрограмі, просто написавши **with** Week.Child. Так само, якщо ми хочемо використовувати ці елементи напряму, ми додатково пишемо **use** Week.Child. Наприклад:

Listing 10: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Week.Child;  use Week.Child;

procedure Main is
begin
   Put_Line ("First day of the week is "
             & Get_First_Of_Week);
end Main;
```

4.4.1 Дочірній пакет дочірнього пакета

Поки що ми бачили дворівневу ієрархію пакетів. Але ієрархія, яку ми потенційно можемо створити, не обмежується цим. Наприклад, ми могли б розширити ієрархію попереднього прикладу, оголосивши пакет Week.Child.Grandchild. У цьому випадку Week.Child буде батьківським пакетом для Grandchild. Розглянемо цю реалізацію:

Listing 11: week-child-grandchild.ads

```
package Week.Child.Grandchild is

   function Get_Second_Of_Week return String;

end Week.Child.Grandchild;
```

Listing 12: week-child-grandchild.adb

```
package body Week.Child.Grandchild is

   function Get_Second_Of_Week return String is
   begin
      return Tue;
   end Get_Second_Of_Week;

end Week.Child.Grandchild;
```

Ми можемо використовувати цей новий пакет Grandchild у нашій тестовій програмі

так само, як і раніше: адаптувати **with** та **use**, а також виклик функції. Ось оновлений код:

Listing 13: main.adb

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

with Week.Child.Grandchild;
use  Week.Child.Grandchild;

procedure Main is
begin
   Put_Line ("Second day of the week is "
             & Get_Second_Of_Week);
end Main;
```

Знову ж таки, це не межа для ієрархії пакетів. Ми могли б продовжити розширення ієрархії попереднього прикладу, реалізувавши пакет Week.Child.Grandchild.Grand_grandchild.

4.4.2 Багато дочірніх пакетів

Поки що ми бачили один дочірній пакет у батьківського пакета. Однак батьківський пакет також може мати кілька дочірніх елементів. Ми могли б розширити наведений вище приклад і реалізувати пакет Week.Child_2. Наприклад:

Listing 14: week-child_2.ads

```
package Week.Child_2 is

   function Get_Last_Of_Week return String;

end Week.Child_2;
```

Тут Week все ще є батьківським пакетом пакета Child, але він також є батьківським пакетом Child_2. Таким же чином Child_2, очевидно, є одним із дочірніх пакетів Week.

Це відповідне тіло пакета Week.Child_2:

Listing 15: week-child_2.adb

```
package body Week.Child_2 is

   function Get_Last_Of_Week return String is
   begin
      return Sun;
   end Get_Last_Of_Week;

end Week.Child_2;
```

Тепер ми можемо посилатися на обидва дочірні пакети в нашому тестовому прикладі:

Listing 16: main.adb

```
with Ada.Text_IO;  use Ada.Text_IO;
with Week.Child;   use Week.Child;
with Week.Child_2; use Week.Child_2;

procedure Main is
begin
   Put_Line ("First day of the week is "
```

(continues on next page)

(continued from previous page)

```
              & Get_First_Of_Week);
   Put_Line ("Last day of the week is "
              & Get_Last_Of_Week);
end Main;
```

4.4.3 Видимість

У попередньому розділі ми бачили, що елементи, оголошені в специфікації батьківського пакета, видимі в дочірньому пакеті. Однак це не стосується елементів, оголошених у реалізації батьківського пакета.

Давайте розглянемо пакет Book і його дочірній - Additional_Operations:

Listing 17: book.ads

```
package Book is

   Title : constant String :=
     "Visible for my children";

   function Get_Title return String;

   function Get_Author return String;

end Book;
```

Listing 18: book-additional_operations.ads

```
package Book.Additional_Operations is

   function Get_Extended_Title return String;

   function Get_Extended_Author return String;

end Book.Additional_Operations;
```

Це реалізація обох пакетів:

Listing 19: book.adb

```
package body Book is

   Author : constant String :=
     "Author not visible for my children";

   function Get_Title return String is
   begin
      return Title;
   end Get_Title;

   function Get_Author return String is
   begin
      return Author;
   end Get_Author;

end Book;
```

Listing 20: book-additional_operations.adb

```
package body Book.Additional_Operations is

   function Get_Extended_Title return String is
   begin
      return "Book Title: " & Title;
   end Get_Extended_Title;

   function Get_Extended_Author return String is
   begin
      --  Строка "Author" декларована в
      --  реалізації пакету Book не видима
      --  тут. Тому ми не можемо написати:
      --
      --  return "Book Author: " & Author;

      return "Book Author: Unknown";
   end Get_Extended_Author;

end Book.Additional_Operations;
```

У реалізації Get_Extended_Title ми використовуємо константу Title з батьківського пакета Book. Однак, як зазначено в коментарях до функції Get_Extended_Author, строка Author — яку ми оголосили в реалізації пакету Book — не видимий в пакеті Book.Additional_Operations. Тому ми не можемо використовувати його для реалізації функції Get_Extended_Author.

Однак ми можемо використати функцію Get_Author з Book у реалізації функції Get_Extended_Author, щоб отримати цей рядок. Так само ми можемо використати цю стратегію для реалізації функції Get_Extended_Title. Це адаптований код:

Listing 21: book-additional_operations.adb

```
package body Book.Additional_Operations is

   function Get_Extended_Title return String is
   begin
      return "Book Title: " & Get_Title;
   end Get_Extended_Title;

   function Get_Extended_Author return String is
   begin
      return "Book Author: " & Get_Author;
   end Get_Extended_Author;

end Book.Additional_Operations;
```

Це простий тестовий код для пакетів вище:

Listing 22: main.adb

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

with Book.Additional_Operations;
use  Book.Additional_Operations;

procedure Main is
begin
   Put_Line (Get_Extended_Title);
   Put_Line (Get_Extended_Author);
end Main;
```

Декларуючи елементи в реалізації пакету, ми можемо реалізувати інкапсуляцію в Ada. Ці елементи будуть видимі лише в самій реалізації пакету, але ніде більше. Однак це не єдиний спосіб досягти інкапсуляції в Ada: ми обговоримо інші підходи в розділі ./privacy.

4.5 Перенайменування

Раніше ми згадували, що *підпрограми можна перейменувати* (page 27). Ми також можемо перейменовувати і пакети. Знову ж таки, для цього ми використовуємо ключове слово **renames**. У наступному прикладі пакет Ada.Text_IO змінюється на TIO:

Listing 23: main.adb

```
with Ada.Text_IO;

procedure Main is
   package TIO renames Ada.Text_IO;
begin
   TIO.Put_Line ("Hello");
end Main;
```

Ми можемо використовувати перейменування, щоб покращити читабельність нашого коду, використовуючи коротші імена пакетів. У прикладі вище ми пишемо TIO.Put_Line замість довшої версії (Ada.Text_IO.Put_Line). Цей підхід особливо корисний, коли ми не використовуємо **use** для пакетів і хочемо уникнути того, щоб код став надто багатослівним.

Зауважте, що ми також можемо перейменовувати підпрограми та об'єкти всередині пакетів. Наприклад, ми могли просто перейменувати процедуру Put_Line як у прикладі нижче:

Listing 24: main.adb

```
with Ada.Text_IO;

procedure Main is
   procedure Say (Something : String)
     renames Ada.Text_IO.Put_Line;
begin
   Say ("Hello");
end Main;
```

В цьому прикладі ми переіменували процедуру Put_Line в Say.

РОЗДІЛ 5

Суворо типізована мова

Ada — це суворо типізована мова. Цікаво, що це відповідає сучасним трендам: сильна статична типізація стає все більш популярною в розробці мов програмування завдяки таким факторам, як розвиток статично типізованого функціонального програмування, великий поштовх дослідницького співтовариства в області типізації та багато практичних мов з системами суворого типізування.

5.1 Що таке тип?

У статично типізованих мовах тип — це переважно (але не тільки) конструкція *часу компіляції*. Це конструкція для забезпечення інваріантів щодо поведінки програми. Інваріанти — це незмінні властивості, які зберігаються для всіх змінних даного типу. Їх застосування гарантує, наприклад, що змінні типу даних ніколи не матимуть недійсних значень.

Тип використовується для міркування про *об'єкти*, якими керує програма (об'єкт — це змінна чи константа). Мета полягає в тому, щоб класифікувати об'єкти за тим, що ви можете виконати з ними (тобто за дозволеними операціями), і таким чином ви можете міркувати про правильність значень об'єктів.

5.2 Цілочислені типи

Приємною особливістю Ada є те, що ви можете визначати власні цілочислені типи на основі вимог вашої програми (тобто діапазону значень, який має сенс). Насправді механізм визначення, який надає Ada, формує семантичну основу для попередньо визначених цілочисельних типів. У цьому відношенні немає «магічного» вбудованого типу, який відрізняється від більшості мов і, можливо, дуже елегантний.

Listing 1: integer_type_example.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Integer_Type_Example is
```

(continues on next page)

(continued from previous page)

```
4      --  Декларуємо знаковий цілочислений
5      --  тип і зазначаємо межі
6      type My_Int is range -1 .. 20;
7      --                         ^ Верхня межа
8      --                   ^ Нижня межа
9
10     --  Як і змінні, типи декларуються
11     --  лише в декларативній частині.
12  begin
13     for I in My_Int loop
14        Put_Line (My_Int'Image (I));
15        --             ^ Атрибут 'Image
16        --               перетворює значення
17        --               на строку.
18     end loop;
19  end Integer_Type_Example;
```

Цей приклад ілюструє декларацію цілочисельного типу зі знаком і кілька речей, які ми можемо з ними робити.

Кожна декларація типу в Ada починається з ключового слова **type** (за винятком task types[11]). Після назви ми можемо побачити діапазон, який дуже схожий на діапазони, які ми використовуємо в циклах **for**, який визначає нижню та верхню межу типу. Кожне ціле число у цьому діапазоні є дійсним значенням для типу.

> **цілочислені типи Ada**
>
> В Ada цілочисельний тип визначається не в термінах його машинного представлення, а радше в його діапазоні. Потім компілятор вибере найбільш відповідне представлення.

У наведеному вище прикладі слід звернути увагу на вираз `My_Int'Image (I)`. Нотація `Name'Attribute (необов'язкові параметри)` використовується для того, що в Ada називається атрибутом. Атрибут — це вбудована операція над типом, значенням або іншою сутністю програми. Доступ до нього здійснюється за допомогою символу `'` (апостроф ASCII).

Ada має декілька типів, доступних як «вбудовані»; :ada: *Integer* є одним із них. Ось як можна визначити **Integer** для типового процесора:

```
type Integer is
  range -(2 ** 31) .. +(2 ** 31 - 1);
```

`**` — оператор експоненти, що означає, що перше дійсне значення для **Integer** дорівнює -2^{31}, а останнє дійсне значення — 2:sup:`31 ` - 1.

Ada не вимагає діапазону для вбудованого типу **Integer**. Реалізація для 16-бітної платформи, ймовірно, вибере діапазон від -2^{15} до 2^{15} - 1.

[11] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html#intro-ada-task-types

5.2.1 Семартика операцій

На відміну від деяких інших мов, Ada вимагає, щоб операції над цілими числами перевірялися на переповнення.

Listing 2: main.adb

```
procedure Main is
   A : Integer := Integer'Last;
   B : Integer;
begin
   B := A + 5;
   --  Ця операція призведе до переповнення, тому
   --  призведе до виникнення виключення під
   --  час виконання
end Main;
```

Є два рівні перевірки переповнення:

- Переповнення на архітектурному рівні, коли результат операції перевищує максимальне значення (або менше мінімального значення), яке може бути представлене в сховищі, зарезервованому для об'єкта типу, і
- Переповнення на рівні типу, коли результат операції виходить за межі діапазону, визначеного для типу.

Здебільшого з міркувань ефективності, хоча переповнення на архітектурному рівні завжди призводить до виключення, переповнення на рівні типу перевірятиметься лише на певних етапах, як-от присвоєння:

Listing 3: main.adb

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

procedure Main is
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;
   M : My_Int := (A + B) / 2;
   --  Виключення небуде,
   --  результат в межах
begin
   for I in 1 .. M loop
      Put_Line ("Hello, World!");
   end loop;
   --  Тіло циклу виконається 13 раз
end Main;
```

Переповнення на рівні типу перевірятиметься лише в певних точках виконання. Результатом, як ми бачимо вище, є те, що у вас може бути операція, яка переповнюється під час проміжного обчислення, але виключення не створюється, оскільки кінцевий результат не виходить за заявлені межі.

5.3 Беззнакові типи

Ada також має беззнакові цілочисленні типи. На мові Ada вони називаються *модульними* типами. Причина такої назви пов'язана з їхньою поведінкою у разі переповнення: вони просто «обертаються», ніби була застосована операція за модулем.

Для модульних типів машинного розміру, наприклад, модуль 2^{32}, імітує найпоширенішу поведінку реалізації беззнакових типів. Однак перевагою Ada є те, що модуль є більш загальним:

Listing 4: main.adb

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

procedure Main is
   type Mod_Int is mod 2 ** 5;
   --              ^ Межі 0 .. 31

   A : constant Mod_Int := 20;
   B : constant Mod_Int := 15;

   M : constant Mod_Int := A + B;
   --  Немає переповнення,
   --  M = (20 + 15) mod 32 = 3
begin
   for I in 1 .. M loop
      Put_Line ("Hello, World!");
   end loop;
end Main;
```

На відміну від C/C++, оскільки ця поведінка обертання гарантується специфікацією Ada, ви можете покластися на неї для реалізації портативного коду. Крім того, можливість використовувати обертання в довільних межах дуже корисна — модуль не обов'язково має бути ступенем 2 — для реалізації певних алгоритмів і структур даних, таких як кільцеві буфери[12].

5.4 Перечислення

Перечислимі типи є ще однією перевагою системи типів Ади. На відміну від перечислення C, вони *не* цілі числа, і кожен новий перечислимий тип несумісний з іншими такими типами. Перечислимі типи є частиною більшого сімейства дискретних типів, що робить їх придатними для використання в певних ситуаціях, які ми опишемо пізніше, але один контекст, який ми вже бачили, це оператор case.

Listing 5: enumeration_example.adb

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

procedure Enumeration_Example is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);
   --  Перечислимий тип
begin
   for I in Days loop
```

(continues on next page)

[12] https://en.wikipedia.org/wiki/Circular_buffer

(continued from previous page)

```
      case I is
         when Saturday .. Sunday =>
            Put_Line ("Week end!");

         when Monday .. Friday =>
            Put_Line ("Hello on "
                      & Days'Image (I));
            --  Атрибут 'Image працює з
            --  перечислимими типами теж
      end case;
   end loop;
end Enumeration_Example;
```

Перечислимі типи достатньо потужні, тому, на відміну від більшості мов, вони використовуються для визначення стандартного логічного типу:

```
type Boolean is (False, True);
```

Як згадувалося раніше, кожен «вбудований» тип в Ada визначається загальнодоступними для користувача засобами.

5.5 Типи з плаваючою комою

5.5.1 Базові властивості

Як і більшість мов, Ada підтримує типи з плаваючою комою. Найпоширенішим типом числа з плаваючою комою є **Float**:

Listing 6: floating_point_demo.adb

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

procedure Floating_Point_Demo is
   A : constant Float := 2.5;
begin
   Put_Line ("The value of A is "
             & Float'Image (A));
end Floating_Point_Demo;
```

Програма відображатиме 2,5 як значення A.

Мова Ada не вказує точність (кількість десяткових цифр у мантисі) для Float; на типовій 32-розрядній платформі точність буде 6.

Доступні всі загальні операції, які можна очікувати для типів із плаваючою комою, включаючи абсолютне значення та піднесення до степеня. Наприклад:

Listing 7: floating_point_operations.adb

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

procedure Floating_Point_Operations is
   A : Float := 2.5;
begin
   A := abs (A - 4.5);
   Put_Line ("The value of A is "
             & Float'Image (A));

```

(continues on next page)

(continued from previous page)

```
   A := A ** 2 + 1.0;
   Put_Line ("The value of A is "
             & Float'Image (A));
end Floating_Point_Operations;
```

Значення A становить 2.0 після першої операції та 5.0 після другої операції.

На додаток до **Float**, реалізація Ada може пропонувати типи даних з вищою точністю, такі як **Long_Float** і **Long_Long_Float**. Як і **Float**, стандарт не вказує точність цих типів: він лише гарантує, що тип **Long_Float**, наприклад, має принаймні точність **Float**. Щоб гарантувати дотримання певної вимоги до точності, ми можемо визначити власні типи з плаваючою комою, як ми побачимо в наступному розділі.

5.5.2 Точність типів з плаваючою комою

Ada дозволяє користувачеві вказати точність для типу з плаваючою комою, виражену десятковими цифрами. Тоді операції над цими користувацькими типами матимуть принаймні вказану точність. Синтаксис простого оголошення типу з плаваючою комою:

```
type T is digits <number_of_decimal_digits>;
```

Компілятор вибере представлення з плаваючою комою, яке підтримує необхідну точність. Наприклад:

Listing 8: custom_floating_types.adb

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

procedure Custom_Floating_Types is
   type T3  is digits 3;
   type T15 is digits 15;
   type T18 is digits 18;
begin
   Put_Line ("T3  requires "
             & Integer'Image (T3'Size)
             & " bits");
   Put_Line ("T15 requires "
             & Integer'Image (T15'Size)
             & " bits");
   Put_Line ("T18 requires "
             & Integer'Image (T18'Size)
             & " bits");
end Custom_Floating_Types;
```

У цьому прикладі атрибут 'Size використовується для отримання кількості бітів, які використовуються для вказаного типу даних. Як ми бачимо, запустивши цей приклад, компілятор виділяє 32 біти для T3, 64 біти для T15 і 128 бітів для T18. Це включає як мантису, так і експоненту.

Кількість цифр, указана в типі даних, також використовується у форматуванні під час відображення змінних із плаваючою комою. Наприклад:

Listing 9: display_custom_floating_types.adb

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

procedure Display_Custom_Floating_Types is
   type T3  is digits 3;
```

(continues on next page)

(continued from previous page)

```
   type T18 is digits 18;

   C1 : constant := 1.0e-4;

   A : constant T3  := 1.0 + C1;
   B : constant T18 := 1.0 + C1;
begin
   Put_Line ("The value of A is "
             & T3'Image (A));
   Put_Line ("The value of B is "
             & T18'Image (B));
end Display_Custom_Floating_Types;
```

Як і очікувалося, програма відображатиме змінні відповідно до заданої точності (1,00E+00 і 1,00010000000000000E+00).

5.5.3 Межі типів з плаваючою комою

Окрім точності, для типу з плаваючою комою також можна вказати діапазон. Синтаксис подібний до того, який використовується для цілочисельних типів даних — використовуючи ключове слово **range**. Цей простий приклад створює новий тип із плаваючою комою на основі типу **Float**, для нормалізованого діапазону між `-1.0` і `1.0`:

Listing 10: floating_point_range.adb

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

procedure Floating_Point_Range is
   type T_Norm  is new Float range -1.0 .. 1.0;
   A  : T_Norm;
begin
   A := 1.0;
   Put_Line ("The value of A is "
             & T_Norm'Image (A));
end Floating_Point_Range;
```

Програма відповідає за те, щоб змінні цього типу залишалися в цьому діапазоні; інакше генерується виключення. У наступному прикладі виключення `Constraint_Error` виникає під час призначення `2.0` змінній A:

Listing 11: floating_point_range_exception.adb

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

procedure Floating_Point_Range_Exception is
   type T_Norm  is new Float range -1.0 .. 1.0;
   A  : T_Norm;
begin
   A := 2.0;
   Put_Line ("The value of A is "
             & T_Norm'Image (A));
end Floating_Point_Range_Exception;
```

Діапазони також можна вказати для користувацьких типів з плаваючою комою. Наприклад:

Listing 12: custom_range_types.adb

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

procedure Custom_Range_Types is
   type T6_Inv_Trig  is
     digits 6 range -Pi / 2.0 .. Pi / 2.0;
begin
   null;
end Custom_Range_Types;
```

У цьому прикладі ми визначаємо тип під назвою `T6_Inv_Trig`, який має діапазон від -π / 2 до π / 2 з мінімальною точністю 6 цифр. (`Pi` визначено у стандартному пакеті `Ada.Numerics`.)

5.6 Сувора типізація

Як зазначалося раніше, Ada є суворо типізованою. В результаті різні типи однієї сім'ї несумісні один з одним; значення одного типу не може бути присвоєно змінній іншого типу. Наприклад:

Listing 13: illegal_example.adb

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

procedure Illegal_Example is
   --  Декларація двух різних типів
   --  з плаваючою комою
   type Meters is new Float;
   type Miles  is new Float;

   Dist_Imperial : Miles;

   --  Декларація константи
   Dist_Metric : constant Meters := 1000.0;
begin
   --  Не вірно: типи різні
   Dist_Imperial := Dist_Metric * 621.371e-6;
   Put_Line (Miles'Image (Dist_Imperial));
end Illegal_Example;
```

Наслідком цих правил є те, що в загальному випадку вираз «змішаного режиму», наприклад `2 * 3.0`, викличе помилку компіляції. У таких мовах, як C або Python, такі вирази стають дійсними шляхом неявних перетворень. В Ada такі перетворення мають бути явними:

Listing 14: conv.adb

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Conv is
   type Meters is new Float;
   type Miles is new Float;
   Dist_Imperial : Miles;
   Dist_Metric : constant Meters := 1000.0;
begin
   Dist_Imperial :=
     Miles (Dist_Metric) * 621.371e-6;
```

(continues on next page)

(continued from previous page)

```
   --  ^^^^^^^^^^^^^^^^^
   --    Перетворення типу з Meters до Miles
   --    Тепер код корректний

   Put_Line (Miles'Image (Dist_Imperial));
end Conv;
```

Звичайно, ми, ймовірно, не хочемо писати код перетворення кожного разу, коли ми перетворюємо метри в милі. Ідіоматичний спосіб Ada в цьому випадку полягав би в тому, щоб ввести функції перетворення разом із типами.

Listing 15: conv.adb

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

procedure Conv is
   type Meters is new Float;
   type Miles  is new Float;

   --  Декларація функції, як процедура
   --  але повертає значення.
   function To_Miles (M : Meters) return Miles is
   --                             ^ тип результату
   begin
      return Miles (M) * 621.371e-6;
   end To_Miles;

   Dist_Imperial : Miles;
   Dist_Metric   : constant Meters := 1000.0;
begin
   Dist_Imperial := To_Miles (Dist_Metric);
   Put_Line (Miles'Image (Dist_Imperial));
end Conv;
```

Якщо ви пишете багато коду з перетворюваннями, необхідність явного надання таких перетворень спочатку може здатися болючою. Однак такий підхід має переваги. Примітно, що ви можете покластися на відсутність неявних перетворень, що, у свою чергу, запобігатиме деяким неявним помилкам.

В інших мовах

У C, наприклад, правила для неявних перетворень не завжди можуть бути цілком очевидними. Однак в Ada код завжди буде робити саме те, що він має робити. Наприклад:

```
int a = 3, b = 2;
float f = a / b;
```

Цей код компілюватиметься добре, але результат `f` буде 1,0 замість 1,5, тому що компілятор згенерує цілочисельне ділення (три поділене на два), що призводить до одиниці. Розробник програмного забезпечення повинен знати про таке перетворення даних і використовувати відповідне приведення:

```
int a = 3, b = 2;
float f = (float)a / b;
```

У виправленому прикладі компілятор перетворить обидві змінні на відповідне представлення з плаваючою комою перед виконанням ділення. Це дасть очікуваний результат.

Цей приклад дуже простий, і досвідчені розробники C, ймовірно, помітять і виправлять його, перш ніж він створить більші проблеми. Однак у більш складних програмах, де оголошення типу не завжди видно — напр. при посиланні на елементи **struct** — ця ситуація не завжди може бути очевидною та швидко призвести до дефектів програмного забезпечення, які може бути важче знайти.

Компілятор Ada, навпаки, завжди відхиляє код, який поєднує змінні з плаваючою крапкою та цілі числа без явного перетворення. Наступний код Ada, заснований на помилковому прикладі в C, не компілюється:

Listing 16: main.adb

```
procedure Main is
   A : Integer := 3;
   B : Integer := 2;
   F : Float;
begin
   F := A / B;
end Main;
```

Відповідний рядок потрібно змінити на `F := Float (A) / Float (B);`, щоб його прийняв компілятор.

Ви можете використовувати сувору типізацію Ada, щоб допомогти застосувати інваріанти у вашому коді, як у прикладі вище: оскільки милі та метри є двома різними типами, ви не можете помилково перетворити екземпляр одного на екземпляр іншого.

5.7 Похідні типи

В Ada ви можете створювати нові типи на основі існуючих. Це дуже корисно: ви отримуєте тип, який має ті самі властивості, що й деякий існуючий тип, але який розглядається як окремий тип в інтересах надійної типізації.

Listing 17: main.adb

```
procedure Main is
   --  Тип: ідентифікаційний номер
   --  несумісний з Integer.
   type Social_Security_Number is new Integer
     range 0 .. 999_99_9999;
   --      ^ Оскільки номер має максимум
   --        9 цифр, та не може бути
   --        негативним, ми накладаємо
   --        такі обмеження для типу.

   SSN : Social_Security_Number :=
     555_55_5555;
   --   ^ Ви можете додавати '_' для
   --     форматування будь якого числа.

   I   : Integer;

   --  Значення -1 нижче викличе виключення
   --  під час виконання і попередження
   --  під час компіляції з GNAT.
   Invalid : Social_Security_Number := -1;
```

(continues on next page)

(continued from previous page)

```
begin
   --  Невірно, різні типи:
   I := SSN;

   --  Також невірно:
   SSN := I;

   --  Вірно, з приведенням типів:
   I := Integer (SSN);

   --  Також вірно:
   SSN := Social_Security_Number (I);
end Main;
```

Тип `Social_Security` називається *похідним типом*; його *батьківським типом* є **Integer**.

Як показано в цьому прикладі, ви можете уточнити дійсний діапазон під час визначення похідного скалярного типу (наприклад, ціле, з плаваючою комою та перерахування).

Синтаксис для перечислень використовує синтаксис **range** <**range**>:

Listing 18: greet.adb

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

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   type Weekend_Days is new
     Days range Saturday .. Sunday;
   --  Новий тип, де тільки Saturday і Sunday
   --  є дійсними.
begin
   null;
end Greet;
```

5.8 Підтипи

Як ми бачили, типи можуть використовуватися в Ada для встановлення обмежень на допустимий діапазон значень. Однак іноді ми хочемо накласти обмеження на деякі значення, залишаючись у межах одного типу. Ось тут і вступають у гру підтипи. Підтип не створює новий тип.

Listing 19: greet.adb

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

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   --  Декларація підтипу
   subtype Weekend_Days is
```

(continues on next page)

(continued from previous page)

```
      Days range Saturday .. Sunday;
   --      ^ Обмеження на підтип

   M : Days := Sunday;

   S : Weekend_Days := M;
   --  Немає помилки, Days та Weekend_Days
   --  є одного типу.
begin
   for I in Days loop
      case I is
         --  Як і тип, підтип може
         --  бути використаний як діапазон
         when Weekend_Days =>
            Put_Line ("Week end!");
         when others =>
            Put_Line ("Hello on "
                      & Days'Image (I));
      end case;
   end loop;
end Greet;
```

Кілька підтипів попередньо визначені в стандартному пакеті Ada та автоматично доступні для вас:

```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

Хоча підтипи одного типу статично сумісні один з одним, обмеження застосовуються під час виконання: якщо ви порушите обмеження підтипу, буде згенеровано виключення.

Listing 20: greet.adb

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

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   subtype Weekend_Days is
     Days range Saturday .. Sunday;

   Day     : Days := Saturday;
   Weekend : Weekend_Days;
begin
   Weekend := Day;
   --         ^ Вірно: той самий тип, підтип
   --           обмеження не порушені
   Weekend := Monday;
   --         ^ Невірне значення для підтипу
   --           виключення під час виконання
end Greet;
```

5.8.1 Підтипи як псевдоніми типів

Раніше ми бачили, що можемо створювати нові типи, оголосивши, наприклад, **type Miles is new Float**. Ми також можемо створити псевдоніми типів, які генеруватимуть альтернативні імена — *псевдоніми* — для відомих типів. Зауважте, що псевдоніми типів іноді називають *синонімами типу*.

Ми досягаємо цього в Ada, використовуючи підтипи без нових обмежень. Однак у цьому випадку ми не отримуємо всіх переваг перевірки типу Ada. Давайте перепишемо приклад, використовуючи псевдоніми типів:

Listing 21: undetected_imperial_metric_error.adb

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

procedure Undetected_Imperial_Metric_Error is
   --  Декларуємо два псевдоніми
   subtype Meters is Float;
   subtype Miles is Float;

   Dist_Imperial : Miles;

   --  Декларація константи
   Dist_Metric : constant Meters := 100.0;
begin
   --  Приведення типів не потрібні
   Dist_Imperial := (Dist_Metric * 1609.0)
                    / 1000.0;

   --  Невірно по лозиції але не виявиться
   --  під час компіляції:
   Dist_Imperial := Dist_Metric;

   Put_Line (Miles'Image (Dist_Imperial));
end Undetected_Imperial_Metric_Error;
```

У прикладі вище той факт, що і Meters, і Miles є підтипами **Float**, дозволяє нам змішувати змінні обох типів без приведення типів. Однак це може призвести до різного роду помилок у программі, яких ми хотіли б уникнути, як ми бачимо в невиявленій помилці, виділеній у коді вище. У цьому прикладі помилка в присвоєнні значення в метрах змінній, призначеній для зберігання значень у милях, залишається непоміченою, тому що і Meters, і Miles є підтипами **Float**. Тому рекомендується використовувати жорстку типізацію — через **type X is new** Y — для таких випадків, як наведений вище.

Однак є багато ситуацій, коли псевдоніми типів корисні. Наприклад, у програмі, яка використовує типи з плаваючою комою в кількох контекстах, ми можемо використовувати псевдоніми типів, щоб вказати додаткове значення типів або уникнути довгих імен змінних. Наприклад, замість того, щоб писати:

```
Paid_Amount, Due_Amount : Float;
```

Ми можемо написати:

```
subtype Amount is Float;

Paid, Due : Amount;
```

В інших мовах

У C, наприклад, ми можемо використовувати декларацію **typedef** для створення псевдоніма типу. Наприклад:

```
typedef float meters;
```

Це відповідає декларації, яку ми бачили вище з використанням підтипів. Інші мови програмування включають цю концепцію подібним чином. Наприклад:

- C++: `using meters = float;`
- Swift: `typealias Meters = Double`
- Kotlin: `typealias Meters = Double`
- Haskell: `type Meters = Float`

Зауважте, однак, що підтипи в Ada відповідають псевдонімам типів тоді і тільки тоді, коли вони не мають нових обмежень. Таким чином, якщо ми додамо нове обмеження до оголошення підтипу, у нас більше не буде псевдоніма типу. Наприклад, таке оголошення *не* можна вважати псевдонімом типу **Float**:

```
subtype Meters is Float range 0.0 .. 1_000_000.0;
```

Розглянемо інший приклад:

```
subtype Degree_Celsius is Float;

subtype Liquid_Water_Temperature is
  Degree_Celsius range 0.0 .. 100.0;

subtype Running_Water_Temperature is
  Liquid_Water_Temperature;
```

У цьому прикладі `Liquid_Water_Temperature` не є псевдонімом `Degree_Celsius`, оскільки він додає нове обмеження, яке не було частиною оголошення `Degree_Celsius`. Однак у нас є псевдоніми двох типів:

- `Degree_Celsius` є псевдонімом **Float**;
- `Running_Water_Temperature` є псевдонімом `Liquid_Water_Temperature`, навіть якщо `Liquid_Water_Temperature` сама по собі має обмежений діапазон.