Esta práctica tiene como objetivo entender cómo se crean módulos en Ada usando paquetes y los conceptos fundamentales de concurrencia usando tareas (simples, sincronizadas y parametrizadas). Se aplicarán estos conceptos de forma progresiva a través de ejercicios.

1 Modularidad con paquetes en Ada

En Ada, un **paquete** es una herramienta que permite **organizar y modularizar el código**. Su propósito es dividir el programa en partes independientes, reutilizables y fáciles de mantener. Cada paquete encapsula un conjunto de elementos relacionados: pueden ser **constantes, variables, tipos, funciones, procedimientos, tareas, objetos protegidos**, etc.
Un paquete consta de dos partes:

- La **especificación (interface)**, que define lo que el paquete ofrece al exterior: nombres de funciones, tipos, procedimientos, etc.
- El **cuerpo (body)**, que contiene la implementación de los elementos declarados en la especificación.

Esto permite separar claramente la **declaración** del comportamiento, ayudando a mantener el código ordenado y permitiendo trabajar en equipo más fácilmente. Además, Ada ofrece un mecanismo de **información oculta** (`private`) para proteger detalles internos del paquete y garantizar el uso correcto del mismo.

1.1 Crear una biblioteca de funciones matemáticas

Vamos a aprender a separar código en especificación y cuerpo, y usarlo desde un programa principal. Imaginemos que estamos creando una pequeña biblioteca de funciones matemáticas que otras personas o programas podrían reutilizar. Para ello, separamos el código en tres partes:

1. **Especificación del paquete** (`operaciones.ads`)

   Aquí se declara qué funciones existen, sin decir cómo están hechas. Es como mostrar el "menú" de lo que ofrecemos.

```
package Operaciones is
   function Sumar(A, B : Integer) return Integer;
   function Multiplicar(A, B : Integer) return Integer;
   function Cuadrado(A: Integer) return Integer;
end Operaciones;
```

2. **Cuerpo del paquete** (`operaciones.adb`):

   Aquí se programa el "cómo": implementamos el código real de las funciones declaradas.

```
package body Operaciones is
   function Sumar(A, B : Integer) return Integer is
   begin
      return A + B;
   end Sumar;

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

```
   function Cuadrado(A : Integer) return Integer is
   begin
      return A * A;
   end Cuadrado;
end Operaciones;
```

3. **Programa principal** (`main.adb`)

   Aquí se prueba el paquete: lo importamos y usamos las funciones de suma, multiplicación y cuadrado.

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

procedure Main is
begin
   Put_Line("Suma 3 + 4 = " & Integer'Image(Operaciones.Sumar(3, 4)));
   Put_Line("Multiplicación 3 * 4 = " &
            Integer'Image(Operaciones.Multiplicar(3, 4)));
   Put_Line("Cuadrado de 5 = " & Integer'Image(Operaciones.Cuadrado(5)));
end Main;
```

Para compilar todos los archivos de especificación y cuerpo (`operaciones.ads` y `operaciones.adb`), simplemente hay que compilar el fichero `main`, siguiendo los siguientes pasos:

```
$gnatmake main.adb
$./main
```

1.2 Encapsulamiento con tipos privados

En Ada, la palabra clave `private` se utiliza para ocultar los detalles de implementación de un tipo o estructura. Esto significa que:

- Parte visible (`public`): En esta sección del paquete, se define lo que otros módulos o partes del programa pueden utilizar. Solo se exponen las funciones, procedimientos y tipos que deberían ser accesibles.
- Parte privada: Aquí es donde se definen los detalles internos del tipo o implementación que no deben ser accesibles desde fuera del paquete. Estos detalles no se pueden modificar directamente, lo que ayuda a proteger la integridad de los datos y asegura que solo se usen los métodos y operaciones permitidas.

Este enfoque permite seguir principios de encapsulación, que es esencial para la ocultación de información y para mantener un diseño limpio y modular.

El uso más importante de `private` es el de los tipos privados que se usan cuando se quiere ocultar la implementación de un tipo. También se usa `private` cuando se trabaja con constantes internas que no debe ser accesibles o cuando se quieren ocultar detalles de implementación más complejo.

1.2.1 Ejemplo: Simulación de un sistema reactivo

Vamos a implementar una simulación sencilla de un sistema reactivo, modelando un sensor de temperatura que se actualiza periódicamente y un controlador que revisa esa temperatura y lanza una alerta si es muy alta.

Para ello, se define un paquete llamado `Temperatura`, que encapsula el tipo de dato `Sensor` como un tipo privado, asegurando así que su implementación interna no sea accesible directamente desde fuera del paquete. El tipo `Sensor` almacena internamente un valor de temperatura (`Valor_Temp`), inicializado a cero. El paquete ofrece tres operaciones públicas: `Crear`, que devuelve un nuevo sensor; `Actualizar`, que permite modificar el valor de temperatura del sensor; y `Leer`, que permite acceder al valor almacenado.

1. **Especificación del paquete** (`temperatura.ads`)

```
package Temperatura is
   type Sensor is private;
   function Crear return Sensor;
   procedure Actualizar(S : in out Sensor; Valor : Float);
   function Leer(S : Sensor) return Float;

private
   type Sensor is record
      Valor_Temp : Float := 0.0;
   end record;
end Temperatura;
```

2. **Cuerpo del paquete** (`temperatura.adb`):

```
package body Temperatura is
   function Crear return Sensor is S : Sensor;
   begin
      return S;
   end Crear;

   procedure Actualizar(S : in out Sensor; Valor : Float) is
   begin
      S.Valor_Temp := Valor;
   end Actualizar;

   function Leer(S : Sensor) return Float is
   begin
      return S.Valor_Temp;
   end Leer;
end Temperatura;
```

3. **Programa principal** (`main.adb`)

   La lógica principal del sistema se desarrolla en el procedimiento `Main`. En él se crea un sensor utilizando la función `Crear`, y a partir de ese momento comienza un bucle infinito que simula el comportamiento del sistema en tiempo real.

```
with Ada.Text_IO, Temperatura, Ada.Real_Time;
use Ada.Text_IO, Temperatura, Ada.Real_Time;

procedure Main is
   -- Crear el sensor
   S : Sensor := Crear;
   Temp_Actual : Float;

   -- Guarda la hora de inicio del programa. Clock devuelve la hora actual
   -- según el reloj del sistema.
   Inicio : Time := Ada.Real_Time.Clock;

begin
   loop
      declare
         -- Obtener el número de segundos desde el inicio del programa
         Ahora : Time := Ada.Real_Time.Clock;
         -- (Ahora - Inicio) da un Time_Span (tiempo pasado desde que
         -- empezó el programa)
         -- To_Duration convierte el Time_Span a segundos (Duration), y se
         -- convierte a Float.
         Time_Seconds : Float := Float(To_Duration(Ahora - Inicio));

         -- Simulación de temperatura
         Valor : Float := 20.0 + Float'Remainder(Time_Seconds, 15.0);
      begin
         Actualizar(S, Valor);
         Put_Line("Sensor: Temperatura actualizada a" & Float'Image(Valor));
      end;

      -- Leer la temperatura del sensor y controlar
      Temp_Actual := Leer(S);
      Put_Line("Controlador: Temperatura leída =" & Float'Image(Temp_Actual));

      if Temp_Actual > 25.0 then
         Put_Line("¡Alerta! Temperatura alta.");
      end if;

      -- Espera de 1 segundo antes de continuar
      delay 1.0;
   end loop;
end Main;
```

En cada iteración del bucle, se calcula un valor de temperatura simulado, que varía en función del tiempo transcurrido desde el inicio del programa. Este valor se utiliza para actualizar el sensor mediante la operación `Actualizar`, y se imprime un mensaje indicando que la temperatura ha sido actualizada.

A continuación, el sistema lee el valor actual del sensor con la operación `Leer`, e imprime el valor leído. Si la temperatura supera un umbral predefinido (en este caso, 25.0 grados), el sistema lanza una alerta indicando que la temperatura es alta.

Finalmente, el sistema espera durante un segundo antes de volver a iniciar el ciclo, simulando así una lectura periódica y continua del entorno. De este modo, se implementa un comportamiento reactivo básico, donde el sistema responde en tiempo real a los cambios en la temperatura simulada.

2 Concurrencia en Ada

Una **tarea** en Ada representa una **unidad concurrente de ejecución**. Es como un hilo o proceso ligero que puede ejecutarse en paralelo con otras tareas.

Cuando el programa comienza, **todas las tareas se activan automáticamente**. No necesitas invocarlas: simplemente, el entorno ejecutará su código concurrentemente.

2.1 Concurrencia con tareas simples

Vamos a estudiar cómo Ada implementa concurrencia mediante tareas, y lo haremos a partir de un ejemplo simple pero representativo: una tarea que escribe mensajes de forma repetida en paralelo al programa principal. La intención es comprender no solo la sintaxis, sino también el modelo de ejecución y el control temporal mediante `delay`.

Comenzamos declarando un procedimiento principal llamado `Ejemplo_tarea`, que será el punto de entrada de la aplicación. Dentro de este procedimiento, introducimos la declaración de una tarea, `Escritor`. Como ya dijimos, en Ada una tarea se declara con la palabra clave `task` y puede contener su propia lógica de control en un bloque `task body`. La declaración de la tarea implica que su activación es automática al comenzar el bloque principal del procedimiento que la contiene, es decir, no es necesario invocarla explícitamente.

```
task Escritor;
task body Escritor is
begin
   -- Código concurrente
end Escritor;
```

Después definimos el comportamiento de la tarea. La tarea `Escritor` va a ejecutar un bucle de cinco iteraciones en las que imprime un mensaje y realiza una pausa entre cada impresión utilizando la instrucción `delay`. Esta instrucción es una forma muy directa y precisa de introducir esperas en programas que requieren sincronización temporal o ejecución periódica. El programa se detiene el tiempo indicado en segundos antes de continuar con la siguiente instrucción. Esto es útil en programación concurrente, donde puede ser necesario esperar entre lecturas de sensores, limitar la frecuencia de ciertas operaciones, o simplemente coordinar la ejecución de varias tareas que deben mantenerse en ritmo.

En este sentido, `delay` está pensado para ser usado en sistemas de tiempo real, por lo que su comportamiento es muy predecible y está estrechamente ligado al reloj del sistema. También permite trabajar con valores temporales más precisos usando tipos específicos como `Duration` o con el paquete `Ada.Real_Time` para trabajar con tiempos absolutos o relativos más exactos.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Ejemplo_Tarea is
   task Escritor;
   task body Escritor is
   begin
      for I in 1 .. 5 loop
         Put_Line("Hola desde la tarea");
         delay 1.5;
      end loop;
   end Escritor;
begin
   Put_Line("Hola desde el programa principal");
   delay 3.0;  -- Esperamos a que la tarea termine
end Ejemplo_Tarea;
```

2.2 Tareas con sincronización

Ahora, vamos a introducir la sincronización activa mediante **entradas** (`entry`) y **aceptaciones** (`accept`), lo que nos permite estudiar cómo las tareas interactúan de forma coordinada. Es como una llamada a **procedimiento bloqueante** que solo se ejecuta cuando la tarea lo permite.

1. **Especificación del paquete** (`temperatura_Entry.ads`)

   Comenzamos con la definición de la tarea llamada `Sensor_Tarea`, que incorpora dos **puntos de entrada** definido mediante `entry Actualizar` y `entry Leer`. Esto puede interpretarse como una operación expuesta por la tarea para ser invocada desde otro contexto concurrente, con una semántica muy similar a una llamada a procedimiento, pero con **sincronización obligatoria**.

```
package Temperatura_Entry is
   task type Sensor_Tarea is
      entry Actualizar(Valor : Float);
      entry Leer(Valor : out Float);
   end Sensor_Tarea;
end Temperatura_Entry;
```

2. **Cuerpo del paquete** (`temperatura_Entry.adb`):

   A continuación, definimos el cuerpo de la tarea. Lo primero que hace la tarea es mostrar un mensaje en consola indicando que está a la espera de una actualización. Esta parte se ejecuta inmediatamente al activarse la tarea, de manera concurrente con el resto del programa. Acto seguido, la tarea entra en una espera activa mediante la instrucción `accept Actualizar`, lo que significa que la tarea suspende su ejecución hasta que el programa principal (u otra tarea) invoque explícitamente la entrada `Actualizar`. En el momento en que esto ocurre, ambas partes —la tarea y el invocador— se sincronizan, y se ejecuta el bloque de código asociado al `accept`, en este caso simplemente una impresión por pantalla. Lo mismo ocurre para el caso de `Leer`.

```
with Ada.Text_IO;
use Ada.Text_IO;
package body Temperatura_Entry is
   task body Sensor_Tarea is
      Temp : Float := 0.0;
   begin
      loop
         Put_Line("Esperando actualizar...");
         accept Actualizar(Valor : Float) do
            Temp := Valor;
            Put_Line("actualizado...");
         end Actualizar;
         Put_Line("Esperando leer ...");
         accept Leer(Valor : out Float) do
            Valor := Temp;
            Put_Line("leído...");
         end Leer;
      end loop;
   end Sensor_Tarea;
end Temperatura_Entry;
```

3. **Programa principal** (`main.adb`)

En el cuerpo principal del procedimiento `Main`, antes de invocar `Actualizar`, el programa actualiza el valor de temperatura. Una vez hecho, procede a la realización de la lectura. Dependiendo de si ese valor es superior o no a 25 grados, lanzará una alerta. A continuación espera un segundo utilizando `delay 1.0`.

```
with Ada.Text_IO, Temperatura_Entry, Ada.Real_Time;
use Ada.Text_IO, Temperatura_Entry, Ada.Real_Time;

procedure Main is
   S : Sensor_Tarea;
   Temp_Actual : Float;
   Inicio : Time := Clock;
begin
   loop
      declare
         Ahora : Time := Clock;
         Time_Seconds : Float := Float(To_Duration(Ahora - Inicio));
         Valor : Float := 20.0 + Float'Remainder(Time_Seconds, 10.0);
      begin
         S.Actualizar(Valor);
         Put_Line("Sensor: Temperatura actualizada a" & Float'Image(Valor));
      end;

      S.Leer(Temp_Actual);
      Put_Line("Controlador: Temperatura leída =" & Float'Image(Temp_Actual));

      if Temp_Actual > 25.0 then
         Put_Line("¡Alerta! Temperatura alta.");
      end if;

      delay 1.0;
   end loop;
end Main;
```

En ejecución, la secuencia sería la siguiente:

1. El programa principal y la tarea se inician de forma concurrente.
2. La tarea `Sensor_Tarea` imprime "*Esperando actualizar ...*" y entra en estado de espera en el `accept` de actualizar.
3. El programa principal realiza una espera hasta que se realice `S.Actualizar(Valor)`, lo que provoca que la tarea despierte y ejecute el cuerpo del `accept`.
4. La tarea `Sensor_Tarea` imprime "*Esperando leer ...*" y entra en estado de espera en el `accept` de leer.
5. El programa principal realiza una espera hasta que se realice `S.Leer(Temp_Actual)`, lo que provoca que la tarea despierte y ejecute el cuerpo del `accept`.
6. Una vez pasado el `delay`, el programa principal vuelve a iterar en el bucle.

Este ejemplo sirve para comprender cómo **Ada permite una sincronización determinista entre tareas**. A diferencia de los modelos de paso de mensajes asincrónicos o las primitivas tradicionales de exclusión mutua, Ada proporciona una abstracción integrada para sincronización estructurada, donde el control de la ejecución es compartido explícitamente entre los participantes.

2.3 Tareas concurrentes parametrizadas

Vamos ahora con otro ejemplo. Este amplía lo visto anteriormente introduciendo tareas parametrizadas y múltiples instancias en ejecución concurrente.

Este procedimiento, llamado `Multitareas`, está diseñado para ilustrar el concepto de tareas concurrentes parametrizadas, donde se crean varias instancias de una misma plantilla de tarea (`task type`), cada una con un comportamiento similar pero identificado por un parámetro individual. Para lograrlo es necesario seguir los siguientes pasos.

2.3.1 Definición de la plantilla de tarea

Lo primero es declarar un tipo de tarea parametrizada, en este caso `Mensajeador`, que recibe un parámetro `Id`. Esta construcción permite crear varias tareas con el mismo cuerpo pero diferente identidad o comportamiento asociado al valor de `Id`.

```
task type Mensajeador(Id : Integer);
```

2.3.2 Cuerpo de la tarea

Luego, se implementa el cuerpo de la tarea. En este caso se podrían tener implementaciones diferentes en base al `Id` proporcionado. En este caso, el cuerpo de la tarea ejecuta un bucle de tres iteraciones. En cada iteración, imprime un mensaje con el identificador de la tarea y el número de la iteración actual. El `delay` introduce una pausa proporcional al valor de `Id`, simulando una carga de trabajo o un ritmo de ejecución diferente para cada tarea.

```
task body Mensajeador is
begin
   for I in 1 .. 3 loop
      Put_Line("Tarea " & Integer'Image(Id) & " -> " & Integer'Image(I));
      delay 0.3 * Id;
   end loop;
end Mensajeador;
```

Esta diferencia en los retardos permite observar cómo se intercalan las salidas de las distintas tareas, demostrando la no determinación del orden de ejecución y la interferencia entre tareas concurrentes, dos aspectos clave al trabajar con concurrencia.

2.3.3 Instanciación de las tareas

Aquí se crean tres instancias de la tarea `Mensajeador`, cada una con un identificador diferente. Estas tareas se inician automáticamente y comienzan a ejecutarse de forma paralela al cuerpo principal del programa.

```
T1 : Mensajeador(1);
T2 : Mensajeador(2);
T3 : Mensajeador(3);
```

2.3.4 Cuerpo principal del programa

La línea inicial imprime un mensaje indicando que ha comenzado la ejecución del programa principal. A continuación, se introduce un delay 2.0, que actúa como una espera para que las tareas concurrentes puedan completarse antes de que finalice el programa principal. Este patrón es común cuando se quiere permitir que tareas hijas finalicen su ejecución sin forzar mecanismos de sincronización explícita.

```
Put_Line("Inicio del programa principal");
delay 2.0;
```

El resultado es el siguiente:

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

procedure Multitareas is
   task type Mensajeador(Id : Integer);
   task body Mensajeador is
   begin
      for I in 1 .. 3 loop
        Put_Line("Tarea " & Integer'Image(Id) & " -> " & Integer'Image(I));
        delay 0.3 * Id;
      end loop;
   end Mensajeador;

   T1 : Mensajeador(1);
   T2 : Mensajeador(2);
   T3 : Mensajeador(3);

begin
   Put_Line("Inicio del programa principal");
   delay 2.0;
end Multitareas;
```

3 Entregables

Tarea 1: Sistema de gestión de cuentas bancarias con paquetes
Desarrollar un sistema de gestión de cuentas bancarias que permita realizar las siguientes operaciones:

1. Creación de cuentas bancarias: Cada cuenta debe tener un titular y un saldo inicial.
2. Depósito y retiro de dinero: Los usuarios deben poder depositar y retirar dinero de sus cuentas.
3. Consulta de saldo: Los usuarios pueden consultar el saldo de su cuenta individualmente.
4. Transferencia entre cuentas: Se podrá realizar transferencias de dinero entre cuentas dentro de un banco.
5. Consulta de saldo total del banco: El sistema debe ser capaz de consultar el saldo total de todas las cuentas en el banco.

Se requiere implementar las siguientes unidades:

- El paquete Cuenta contendrá:
  - Tipo Cuenta_Bancaria: Define la estructura de una cuenta bancaria, que debe almacenar el titular (nombre) y el saldo de la cuenta.
  - Operaciones:
    * Crear: Inicializa una cuenta bancaria con un titular y un saldo inicial.
    * Depositar: Permite agregar una cantidad de dinero al saldo de la cuenta.
    * Retirar: Permite retirar una cantidad de dinero de la cuenta, verificando que haya suficiente saldo.
    * Consultar_Saldo: Devuelve el saldo actual de la cuenta.
    * Obtener_Titular: Devuelve el nombre del titular de la cuenta.
- El paquete Banco contendrá:
  - Tipo Banco: Define la estructura de un banco, que puede almacenar hasta 10 cuentas bancarias.
  - Operaciones:
    * Crear_Banco: Inicializa un banco vacío.
    * Agregar_Cuenta: Permite agregar una cuenta bancaria al banco.
    * Transferir: Realiza una transferencia de dinero entre dos cuentas dentro del banco.
    * Consultar_Cuenta: Permite consultar una cuenta bancaria por el nombre del titular.
    * Consultar_Saldo_Banco: Devuelve el saldo total de todas las cuentas en el banco.

- Fichero principal:
  - Secuencia de operaciones a realizar es:
    * Crear un banco.
    * Crear varias cuentas bancarias con titulares y saldos iniciales.
    * Agregar las cuentas al banco.
    * Realizar transferencias entre cuentas.
    * Consultar y mostrar los saldos de las cuentas y el saldo total del banco.
    * Realizar depósitos y retiros en las cuentas.
- Requerimientos:
  - El banco puede tener hasta 10 cuentas bancarias. Si se intenta agregar más, debe lanzarse un error.
  - Las transferencias y retiros deben verificar si las cuentas tienen el saldo suficiente.
  - El sistema debe ser capaz de manejar la creación de cuentas, depósitos, retiros y transferencias correctamente, asegurando que las operaciones no generen saldos negativos (excepto en casos de error en retiro o transferencia por saldo insuficiente).

Tarea 2: Sistema de gestión de cuentas bancarias con concurencia

A partir de las operaciones de gestión de cuentas bancarias anteriores implementar tareas concurrentes para realizar transferencias, depósitos y retiros de forma simultánea.

Se requiere implementar un sistema de tareas concurrentes donde es necesario:

- Definir tareas para Transferir, Depositar, Retirar y Consultar_Saldo.
- Utilizar entradas (entry) y aceptaciones (accept) en las tareas para sincronizar las operaciones y asegurar que las consultas de saldo no se realicen mientras se están ejecutando otras operaciones.

El programa principal creará un banco con varias cuentas bancarias. Luego lanzará las tareas concurrentes para realizar transferencias, depósitos y retiros. Se tendrá que asegurar que las consultas de saldo se realicen solo cuando no hay operaciones en curso.