Concurrency in Ada – Basic Concepts

Ben Brosgol and Martyn Pike

university.adacore.com

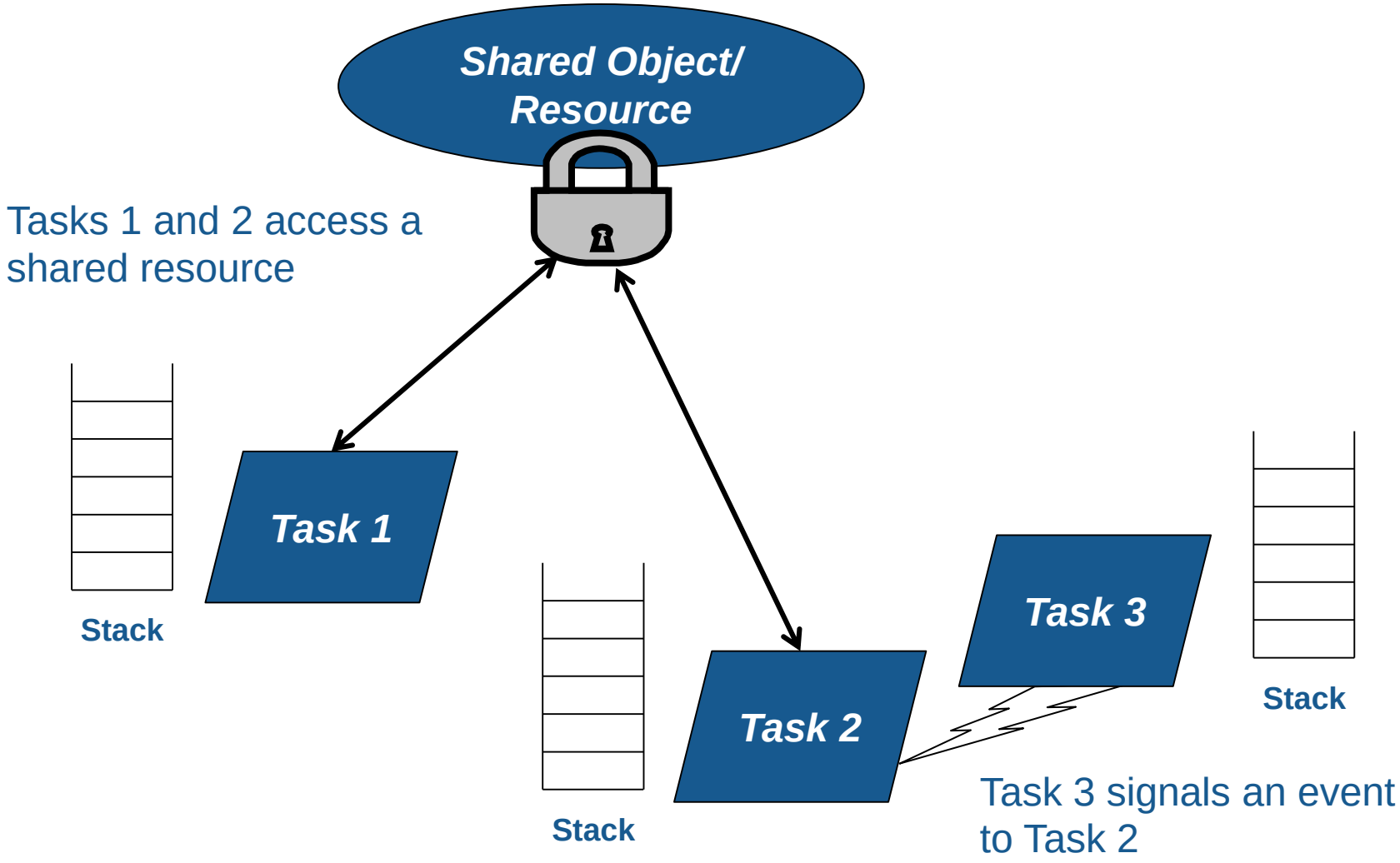
Introduction

- **What is a concurrent program?**
  - A set of *active* program entities (tasks) that interact *cooperatively* in the use of *passive* entities (data structures, shared resources)
  - Active entity has a thread of control and a stack
  - To avoid corruption, passive entity needs mutually exclusive access
    - May be implemented by hardware (atomic access), software (locks), or program logic
  - Cooperation = synchronization/coordination between tasks
    - For example, an “event” signaled by one task and awaited by another

Introduction

- **Why concurrent programming?**
  - Improve performance on multiple processors / cores
  - Exploit OS services on a single processor
  - Model intrinsic parallelism in the problem space

Concurrent Program Structure



Concurrent Program Execution

- **Actual concurrency**
  - Parallel execution on separate processors/cores
- **Virtual concurrency**
  - Multiplexed execution on a single processor
  - Task dispatcher controls when tasks run
    - Priority can be used to establish execution preference among multiple tasks that are ready to execute
    - Dispatching policies (to be covered in a later lesson) include “run until blocked or preempted”, time-slicing
    - Dispatching policy unspecified unless defined by program
- **Shared data**
  - Common address space assumed

A Simple Ada Tasking Program

- **This program displays the string “Hello” 60 times, with at least a one-second delay between each output**

```
with Ada.Text_IO;

procedure Simple_Tasking_Program is
   N : Integer := 60;
   task Simple_Task;           -- Task specification

   task body Simple_Task is    -- Task body
   begin
      for I in 1..N loop
         Ada.Text_IO.Put_Line ("Hello");
         delay 1.0;            -- Suspend execution for at least 1 second
      end loop;
   end Simple_Task;
begin                          -- Activate Simple_Task here
   null;
end Simple_Tasking_Program ;   -- Wait for Simple_Task to complete before returning
```

- **There are two threads of control**
  - The “environment task”, which calls the main procedure
  - The task Simple_Task declared in the main procedure

Task Structure

- **A task is a declared unit containing a specification and a body**
  - The specification is the interface to the rest of the program and may be empty

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

procedure Simple_Tasking_Program is
   N : Integer := 60;
   task Simple_Task;              -- Task specification

   task body Simple_Task is       -- Task body
   begin
      for I in 1..N loop
         Put_Line ("Hello");
         delay 1.0;               -- Suspend execution for at least 1 second
      end loop;
   end Simple_Task;
begin                             -- Activate Simple_Task here
   null;
end Simple_Tasking_Program ;      -- Wait for Simple_Task to complete before returning
```

  - The body consists of (optional) declarations and a sequence of statements that form the algorithm performed by the task

Task Lifetime: Activation

- **A task is activated (its declarations are elaborated) when control reaches the "begin" of its enclosing scope**

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

procedure Simple_Tasking_Program is
   N : Integer := 60;
   task Simple_Task;           -- Task specification

   task body Simple_Task is    -- Task body
   begin
      for I in 1..N loop
         Put_Line ("Hello");
         delay 1.0;            -- Suspend execution for at least 1 second
      end loop;
   end Simple_Task;
begin                          -- Activate Simple_Task here
   null;
end Simple_Tasking_Program ;   -- Wait for Simple_Task to complete before returning
```

  - At this point the execution of the enclosing program unit is suspended
- **When the activated task reaches its "begin", both the task and its enclosing unit are eligible for execution**
  - The choice depends on the task dispatching policy and execution environment

Task Lifetime: Termination

- **A task completes when control reaches the "end" of its body**

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

procedure Simple_Tasking_Program is
   N : Integer := 60;
   task Simple_Task;           -- Task specification

   task body Simple_Task is    -- Task body
   begin
      for I in 1..N loop
         Put_Line ("Hello");
         delay 1.0;            -- Suspend execution for at least 1 second
      end loop;
   end Simple_Task;
begin                          -- Activate Simple_Task here
   null;
end Simple_Tasking_Program ;   -- Wait for Simple_Task to complete before returning
```

  - It terminates if it does not contain any nested tasks
- **The enclosing scope cannot complete until its nested tasks (its "dependents") have all terminated**
  - This prevents "dangling references" from a nested task to local variables in the enclosing scope

Top-Level Tasks

- A task may be declared in a library-level package

```
package Global_Pkg is
   … -- Declarations
end Global_Pkg;

package body Global_Pkg is
   …
   task T;
   task body T is
      …     -- Declarations local to T
   begin
      loop
         … -- Task's processing
      end loop;
   end T;
end Global_Pkg;
```

```
with Global_Pkg;
procedure Main_Proc is
   …
begin
   …
end Main_Proc;
```

- Often such a task is an infinite loop, never terminating
- A task in a library-level package is activated before the main procedure is invoked
  - It can continue execution after the main procedure returns

Simple Task Synchronization

- **Task nesting may be used for simple synchronization**

```
procedure Simple_Synchronization is
   Data_1 : Some_Type_1 := …;
   Data_2 : Some_Type_2 := …;
begin
   declare
      task T1;
      task body T1 is
      begin
         … -- Process Data_1
      end T1;

      task T2;
      task body T2 is
      begin
         … -- Process Data_2
      end T2;
   begin    -- Activate T1 and T2 here
      null;
   end;     -- Wait for T1 and T2 to terminate
   …        -- Use Data_1 and Data_2
end Simple_Synchronization;
```

  - It terminates if it does not contain any nested tasks
- **Execution of the declare block is suspended at the "end" until its nested tasks have all terminated**

Exceptions in Tasks

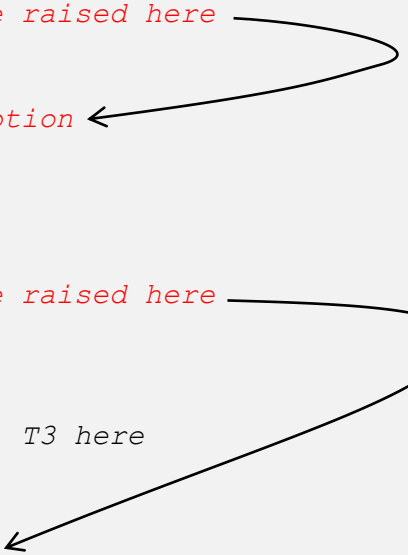
- **An exception raised in a task's statement part and not handled by the task is not propagated to the enclosing scope**
  - The surrounding statements are being executed concurrently
  - Propagation of an asynchronous exception into this context would raise serious semantic, implementation, and stylistic issues
- **If the exception is not handled by the task, the task will terminate silently**
  - To be covered later: providing a termination-related action
- **An exception raised during the elaboration of a task's declarative part is not handled by the task**
  - The elaboration occurs as part of the task's activation, with the enclosing scope suspended just after its "begin"
  - Tasking_Error is propagated (synchronously) to the enclosing scope

Exceptions in Tasks

```
procedure P is
   task T1;
   task body T1 is
   begin
      … -- Exception may be raised here
      … -- T1 terminates silently
   end T1;

   task T2;
   task body T2 is
   begin
      … -- Exception may be raised here
   exception
      when others =>
         … -- Handle exception
   end T2;

   task T3;
   task body T3 is
      … -- Exception may be raised here
   begin
      …
   end T3;
begin  -- Activate T1, T2, T3 here
   …
exception
   when Tasking_Error =>
      … -- Handle exception propagated by failure of T3 activation
end P;
```



"Catch-all" handler is common style. Task can log information to a file for inspection after the program terminates

Review

- **This lesson covered some basic principles of Ada tasking**
  - Task structure (specification and body)
  - Task lifetime properties (activation, termination)
- **Main points**
  - Tasks are declared much like other program units
  - A task can reference entities in outer scopes (normal block structure)
  - A task can execute concurrently with the statements in the scope where it is declared
  - A task depends on its enclosing scope, which cannot terminate until all dependent tasks have terminated
  - A task declared in a library-level package does not depend on the main procedure
  - An unhandled exception in a task's statements is not propagated

Quiz

What does the program do? (1/10)

```
package Global_Pkg is
end Global_Pkg;

with Ada.Text_IO;
package body Global_Pkg is
   task T;
   task body T is
   begin
      loop
         Ada.Text_Io.Put_Line("Running");
         delay 1.0;
      end loop;
   end T;
end Global_Pkg;
```

```
with Global_Pkg;
procedure Main_Proc is
begin
  null;
end Main_Proc;
```

Answer (1/10)

```
package Global_Pkg is
end Global_Pkg;

with Ada.Text_IO;
package body Global_Pkg is
   task T;
   task body T is
   begin
      loop
         Ada.Text_Io.Put_Line("Running");
         delay 1.0;
      end loop;
   end T;
end Global_Pkg;
```

```
with Global_Pkg;
procedure Main_Proc is
begin
  null;
end Main_Proc;
```

Main_Proc can return even though T is still executing

A library-level task does not depend on the main procedure, so the main procedure can return while the task is still running

What does the program do? (2/10)

```
with Ada.Text_IO;
procedure Main_Proc is
   task T;
   task body T is
   begin
      loop
         Ada.Text_Io.Put_Line("Running");
         delay 1.0;
      end loop;
   end T;
begin
  null;
end Main_Proc;
```

Answer (2/10)

```
with Ada.Text_IO;
procedure Main_Proc is
   task T;
   task body T is
   begin
      loop
         Ada.Text_Io.Put_Line("Running");
         delay 1.0;
      end loop;
   end T;
begin
  null;
end Main_Proc;
```

T loops forever, preventing Main_Proc from returning

Task T depends on its enclosing unit (procedure Main_Proc). Since Main_Proc cannot return until T terminates, and T is in an infinite loop, Main_Proc will be suspended at its “end” and will not be able to return

What's the output of this program? (3/10)

```
with Ada.Text_IO;
procedure Proc is
   Arr : array (1..10) of Integer := …; -- Initialization
   Max : Integer := Integer'First;
begin
  declare
    task Find_Max;

    task body Find_Max is
    begin
      for I in Arr'Range loop
         if Max < Arr( I) then Max := Arr (I); end if;
      end loop;
    end Find_Max;
  begin
     null;
  end;
  Ada.Text_IO.Put_Line (Integer'Image (Max));
end Proc;
```

Answer (3/10)

```
with Ada.Text_IO;
procedure Proc is
   Arr : array (1..10) of Integer := …; -- Initialization
   Max : Integer := Integer'First;
begin
  declare
    task Find_Max;

    task body Find_Max is
    begin
      for I in Arr'Range loop
         if Max < Arr( I) then Max := Arr (I); end if;
      end loop;
    end Find_Max;
  begin
     null;
  end;
  Ada.Text_IO.Put_Line (Integer'Image (Max));
end Proc;
```

The maximum value in the array

Declaring Find_Max in an inner block means that the reference to Max in the statement in Proc will not occur until Find_Max has terminated.

Task Termination (4/10)

Please consider the following statement on Task Termination

The enclosing scope cannot complete until its nested tasks (its "dependents") have all terminated

Answer (4/10)

The enclosing scope cannot complete until its nested tasks (its “dependents”) have all terminated

TRUE

What's the output of this program? (5/10)

```
with Ada.Text_IO;
procedure Proc is
begin
  declare
    task T;
    task body T is
      N : Positive;
    begin
      N := -1; -- Raises Constraint_Error
     end T;
  begin
     null;
  end;
  Ada.Text_IO.Put_Line ("Normal Return");
exception
  when others =>
    Ada.Text_IO.Put_Line ("Exceptional return");
end Proc;
```

Answer (5/10)

```
with Ada.Text_IO;
procedure Proc is
begin
  declare
    task T;
    task body T is
      N : Positive;
    begin
      N := -1; -- Raises Constraint_Error
     end T;
  begin
     null;
  end;
  Ada.Text_IO.Put_Line ("Normal Return");
exception
  when others =>
    Ada.Text_IO.Put_Line ("Exceptional return");
end Proc;
```

The string “Normal Return”

Task T “dies silently” when the exception is raised; no exception is propagated back to Proc

What's the output of this program? (6/10)

```
with Ada.Text_IO;
procedure Proc is
begin
  declare
    task T;
    task body T is
      N : Positive := -1; -- Raises Constraint_Error
    begin
      null;
     end T;
  begin
     null;
  end;
  Ada.Text_IO.Put_Line ("Normal Return");
exception
  when others =>
    Ada.Text_IO.Put_Line ("Exceptional return");
end Proc;
```

Answer (6/10)

```
with Ada.Text_IO;
procedure Proc is
begin
  declare
    task T;
    task body T is
      N : Positive := -1; -- Raises Constraint_Error
    begin
      null;
     end T;
  begin
     null;
  end;
  Ada.Text_IO.Put_Line ("Normal Return");
exception
  when others =>
    Ada.Text_IO.Put_Line ("Exceptional return");
end Proc;
```

The string “Exceptional Return”

The declarative part of T is elaborated as part of the activation of T, when execution is suspended just after the “begin” of the declare block. The exception Tasking_Error is raised (synchronously) at that point, and is handled by Proc’s exception handler.

Why Concurrent Programming ? (7/10)

Which of the following statements is NOT a reason for Concurrent Programming introduced in this Lesson ?

Answer (7/10)

Concurrent Programs make Object Orientation Easier

Is NOT a reason for Concurrent Programming introduced in this Lesson

The correct reasons are :

Improve performance on multiple processors / cores

Exploit OS services on a single processor

Model intrinsic parallelism in the problem space

Is this correct? (8/10)

YES
(click on the check icon)

NO
(click on the error location(s))

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

procedure Simple_Tasking_Program is

   N : Integer := 60;

   task Simple_Task;           -- Task specification

   task body Simple_Task is    -- Task body
   begin

      for I in 1..N loop

         Put_Line ("Hello");
         sleep 1.0;
         -- Suspend execution for at least 1 second

      end loop;

   end Simple_Task;

begin
   null;
end Simple_Tasking_Program;
```

Is this correct? (8/10)

NO

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

procedure Simple_Tasking_Program is

   N : Integer := 60;

   task Simple_Task;           -- Task specification

   task body Simple_Task is    -- Task body
   begin

      for I in 1..N loop

         Put_Line ("Hello");
         sleep 1.0;
         -- Suspend execution for at least 1 second

      end loop;

   end Simple_Task;

begin
   null;
end Simple_Tasking_Program;
```

```
-- Correct code should have
-- used a delay statement

delay 1.0;
```

Please pick an answer (9/10)

Which of the following correctly identifies the entity that starts the main procedure ?

Answer (9/10)

The Correct Answer is

The Environment Task

Is this correct? (10/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
with Ada.Text_IO;

procedure Simple_Tasking_Program is

   N : Integer := 60;

   task Simple_Task;

begin

   task body Simple_Task is
   begin
      for I in 1..N loop
         Ada.Text_IO.Put_Line ("Hello");
         delay 1.0;
      end loop;
   end Simple_Task;

end Simple_Tasking_Program ;
```

Is this correct? (10/10)

NO

```
with Ada.Text_IO;

procedure Simple_Tasking_Program is

   N : Integer := 60;

   task Simple_Task;

begin

   task body Simple_Task is
   begin
      for I in 1..N loop
         Ada.Text_IO.Put_Line ("Hello");
         delay 1.0;
      end loop;
   end Simple_Task;

end Simple_Tasking_Program ;
```

```
task body Simple_Task is
begin
[…]
end Simple_Tasking_Program ;

-- Must appear in declarative section

begin
```

university.adacore.com