From M. B. Feldman, *Software Construction and Data Structures,* Addison-Wesley, 1996.

CHAPTER 15

Introduction to Concurrent Programming

15.1 What Is Concurrent Programming?
15.2 Ada Structures: Task Types and Task Objects
15.3 Ada Structures: Protected Types and Protected Objects
15.4 Data Structures: The Task as a Data Structure
15.5 Application: Simulation of a Bank
15.6 Application: The Dining Philosophers

Each program we have seen so far has been a *sequential*, or *single-threaded,* one; that is, it has consisted of a series of steps that are executed in sequence, one after the other. In this chapter, we introduce the idea of a *concurrent*, or *multithreaded*, program, one in which several things can happen—or at least appear to happen—simultaneously.

Concurrent actions are really part of most interesting programs. For example, a time-shared operating system must deal with a number of human users working simultaneously at their terminals. Further, many real-time applications, especially those controlling physical processes, are composed of concurrent program segments, each responsible for its own physical subsystem. Finally, the world is concurrent, filled with people doing different things all at the same time, and a program that would model that world is best seen as comprising concurrent program segments.

This chapter introduces you to the fascinating field of *concurrent programming,* which is the writing of concurrent programs. Ada provides an especially rich and interesting set of structures for concurrent programming; this chapter presents some of these structures. In particular, we introduce Ada *task types* and *protected types.* A task object is an active program, carrying on its activities independently of other tasks and interacting with others only when necessary. A protected object is passive; its purpose is to encapsulate a data structure and provide services to tasks on request, allowing many tasks to view the structure simultaneously but authorizing only one task at a time to modify the structure.

15.1 What Is Concurrent Programming?

Much of the programming world involves concurrent applications. Here are some examples from operating systems, real-time systems, and simulation.

copyright © 1996, Michael B. Feldman

Operating Systems

When you and your colleagues all log in at terminals connected to the same time-sharing system, each of you works separately, but you are all using the same computer. Each of you has the feeling that the computer is working only on your task, yet many of you are working simultaneously. How is this seeming paradox possible?

The illusion that you are alone on the time-shared computer is caused by a combination of fast computers and clever programming. Suppose you are using the computer to edit a program or read electronic mail. You read and type at human speed. A very fast typist can enter 100 words per minute, or—at an average of six characters per word—about 10 characters per second. In the tenth of a second between two of your keystrokes, a modern computer can execute hundreds of thousands of machine instructions. If those "extra" machine instructions could be put to productive use, the computer would have plenty of time between your keystrokes to service other human users. It is not unusual for a modern time-shared computer to handle 100 or more simultaneous users, each working at human speed.

Managing all those instructions and users is part of the responsibility of a modern operating system. An operating system is, as you know by now, just a sophisticated program; in fact, it is a *concurrent* program, capable of managing many devices and human users to give the illusion of simultaneity.

Some time-shared computers consist of a single CPU; others consist of a set of identical CPUs. With more than one CPU, programs can be executed *in parallel*—that is, literally at the same time. With a single CPU, no real parallel execution is possible, but that one CPU can be shared in such a way that many programs *seem* to be executing in parallel. Concurrent programming is the creation of programs that consist of segments that have the potential for parallel execution; depending upon the actual number of CPUs available, execution of a concurrent program may be literally parallel, entirely time-shared, or some combination of the two.

Real-Time Systems

Many computer systems exist to control physical systems of one kind or another. Examples abound in medical technology, manufacturing and robotics, and transportation. In the latter domain, real-time computer programs control modern automotive fuel systems, aircraft such as the Boeing 777, and railroads such as the Channel Tunnel between France and England and the subway system in Washington, DC. These are, of necessity, concurrent programs: They must manage a number of electronic devices simultaneously; these devices, in turn, are connected to physical machines such as an automobile's fuel injection system or a railroad's "turnout" (a movable section of track that allows a train to enter one or the other of two rail lines).

Modeling and Simulation

Concurrent programming is useful in modeling or simulating physical systems, even if those systems are not directly controlled by a computer. For example, the waiting and service times in a bank, supermarket, or other service organization can be studied by writing a program in which each customer and each server—bank teller, supermarket checker, airline reservation clerk—is represented by its own program segment, which interacts with the other segments.

Similarly, a subway system can be modeled by a program in which each train, station, turnout, and block (section of track that is permitted to hold at most one train) is represented by a program segment. The flow of simulated customers in the bank, or of simulated trains in the subway, can be controlled or varied at will.

Simulation is an important tool in optimizing physical systems—for example, choosing the most effective number of open checkout lines in a supermarket or the frequency and maximum speed of trains in a subway. Studying the computer model provides information and insight into the behavior of the physical system if the former is a faithful representation of the latter; concurrent programming provides a natural way of assigning program segments to represent physical objects and therefore aids greatly in developing good simulations.

copyright © 1996, Michael B. Feldman

Ada is one of only a few programming languages—and the only popular one—to provide built-in structures for concurrent programming. In this chapter, we use a series of examples to present a few of the basic Ada structures and end with two simulations: one of a bank and the other of a group of philosophers in a Chinese restaurant.

Ada Structures for Concurrent Programming

In concurrent programming, an execution of a program segment is called a *process.* For example, when, logged into a time-sharing system, you invoke the electronic mail program, a process is created. The mail program itself is just a file on disk; when it is loaded into memory and executed, that execution is a process. If you and several friends all log in at the same time and invoke the e-mail program, several copies of that program are executing simultaneously on the same computer. One *program* has given rise to multiple simultaneous *processes.* Ada's term for *process* is *task;* Ada provides *task types* to allow the creation of multiple processes, which Ada calls *task objects*, resulting from a single program declaration.

Generally, your incoming e-mail is stored in a system file called the electronic mailbox, or just the mailbox. Suppose you are reading your mail when a friend sends you a message. The new message must be added to your mailbox file; your reading must be momentarily suspended while the file is modified (you may not notice the temporary suspension, but it happens anyway). Now suppose that two incoming messages arrive at the same time. Not only must your reading be suspended, but something in the mail software must update your mailbox one message at a time. If this protection were not provided—if two messages could update the mailbox literally at the same time—the mailbox would become hopelessly garbled and therefore useless.

The e-mail situation is an example of a *readers-writers problem,* a category of computing problems in which multiple readers of, and multiple writers to, a data structure must be prevented from interfering with one another. The prevention technique is called *mutual exclusion;* update actions on the data structure are handled one at a time while other actions are excluded. Ada's *protected types* provide mutual exclusion; we can declare a protected type, and variables of that type, with read operations (called *protected functions)* and update operations (*protected procedures),* which Ada guarantees it will execute correctly. Specifically, multiple calls to a protected procedure are executed one at a time.

Section 15.2 introduces task types and task objects; Section 15.3 introduces protected types and protected objects.

15.2 Ada Structures: Task Types and Task Objects

An Ada task is an interesting structure. It has aspects of a package, of a procedure, and of a data structure, but is really none of these; it is something different altogether:

- Like a package, a task has a specification and a body. Unlike a package, it must be declared in an enclosing structure, not put in a separate file and compiled separately.
- Like a procedure, a task has a declaration section and a sequence of executable statements. However, it is not called like a procedure; rather, it starts executing implicitly, automatically, as part of its enclosing block.
- Like a data structure, it has a type and is brought into existence by declaring a variable of the type. Indeed, like a variant record, it can have one or more discriminants.

Program 15.1 illustrates these aspects of tasks.

Program 15.1
A Task within a Main Program

```
WITH Ada.Text_IO;
PROCEDURE One_Task IS
------------------------------------------------------------------------
```

```
--| Show the declaration of a simple task type and one
--| variable of that type.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  -- A task type has a specification
  TASK TYPE SimpleTask (Message: Character);

  -- A task type has a body
  TASK BODY SimpleTask IS

  BEGIN -- SimpleTask

    FOR Count IN 1..10 LOOP
      Ada.Text_IO.Put(Item => "Hello from Task " & Message);
      Ada.Text_IO.New_Line;
    END LOOP;

  END SimpleTask;

  Task_A: SimpleTask(Message => 'A');

BEGIN -- One_Task

-- Unlike procedures, tasks are not "called" but are activated
-- automatically.

-- Task_A will start executing as soon as control reaches this
-- point, just after the BEGIN but before any of the main program's
-- statements are executed.

  NULL;

END One_Task;
```

A sample run of this program would look like this:

```
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
```

First note the overall structure of the program. A task type, `SimpleTask`, is declared with a discriminant, `Message`. This task specification is followed by a task body in which the message is displayed 10 times. Next, `Task_A` is declared as a task variable, usually called a task object, with a discriminant value of `'A'`.

Reaching the main `BEGIN` of this program, we discover that the program has no executable statements, just a `NULL` statement to satisfy the rule that a procedure must have at least one statement. Yet the sample run shows the task actually displaying `Hello from Task A` 10 times. The task was never called from the main program, but it executed anyway.

In fact, the task began its execution just after the main `BEGIN` was reached. In Ada, this is called “task activation”: All tasks declared in a given block are activated just after the `BEGIN` of that block. Here, there is only one task, `Task_A`.

copyright © 1996, Michael B. Feldman

Multiple Task Objects of the Same Type

Program 15.2 shows the declaration of two task objects, `Task_A` and `Task_B`. Further, the task type is modified to allow two discriminants, the message and the number of times the message is to be displayed. Here, a discriminant acts like a parameter of the task, but it is not a fully general parameter; like a variant-record discriminant, it must be of a discrete—integer or enumeration—type. A string, for example, cannot be used as a task discriminant.

Program 15.2
Two Tasks within a Main Program

```
WITH Ada.Text_IO;
PROCEDURE Two_Tasks IS
------------------------------------------------------------------------
--| Show the declaration of a simple task type and two
--| variables of that type.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  -- A task type has a specification
  TASK TYPE SimpleTask (Message: Character; HowMany: Positive);

  -- A task type has a body
  TASK BODY SimpleTask IS

  BEGIN -- SimpleTask

    FOR Count IN 1..HowMany LOOP
      Ada.Text_IO.Put(Item => "Hello from Task " & Message);
      Ada.Text_IO.New_Line;
    END LOOP;

  END SimpleTask;

  -- Now we declare two variables of the type
  Task_A: SimpleTask(Message => 'A', HowMany => 5);
  Task_B: SimpleTask(Message => 'B', HowMany => 7);

BEGIN -- Two_Tasks

-- Task_A and Task_B will both start executing as soon as control
-- reaches this point, again before any of the main program's
-- statements are executed. The Ada standard does not specify
-- which task will start first.

  NULL;

END Two_Tasks;
```

This time, the program execution might look like this:

```
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task A
Hello from Task A
Hello from Task A
```

```
Hello from Task A
Hello from Task A
```

As in Program 15.1, `Task_A` and `Task_B` are activated just after the main `BEGIN`. Now there are two tasks. In which order are they activated? The Ada standard does not specify this, leaving it up to the compiler implementer instead. In a short while, we will see how to control the order in which tasks start their work.

Looking at the sample run from Program 15.2, we see that `Task_B` evidently started—and completed—its work before `Task_A` even started its own work. This tells us first that the compiler we used activated `Task_B` first, and also that, once scheduled for the CPU, `Task_B` was allowed to continue executing until it completed its run. This seems odd: The tasks do not really execute as though they were running in parallel; there is, apparently, no time-sharing. If there were, we would expect `Task_A` and `Task_B` output to be interleaved in some fashion.

In fact, the Ada standard allows, but does not require, *time-slicing*. Time-slicing, implemented in the run-time support software, supports "parallel" execution by giving each task a slice, usually called a *quantum,* which is a certain amount of time on the CPU. At the end of the quantum, the run-time system steps in and gives the CPU to another task, allowing it a quantum, and so on, in "round-robin" fashion.

Cooperating Tasks

If Program 15.2 were compiled for a computer with several processors, in theory `Task_A` and `Task_B` could have been executed—truly in parallel—on separate CPUs, and no time-slicing would be needed. With a single CPU, we'd like to emulate the parallel operation, ensuring concurrent execution of a set of tasks even if the Ada run-time system does not time-slice.

To get "parallel" behavior portably, using one CPU or many, with or without time-slicing, we code the tasks in a style called *cooperative multitasking;* that is, we design each task so that it periodically "goes to sleep," giving up its turn on the CPU so that another task can execute for a while.

Program 15.3 shows how this is done, using a `DELAY` statement in each iteration of the task body's `FOR` loop. The `DELAY` causes the task to suspend its execution, or "sleep." Now while `Task_A` is "sleeping," `Task_B` can be executing, and so on. The cooperating nature of the two tasks is easily seen in the sample output.

Program 15.3
Using `DELAY` to Achieve Cooperation

```
WITH Ada.Text_IO;
PROCEDURE Two_Cooperating_Tasks IS
------------------------------------------------------------------------
--| Show the declaration of a simple task type and two
--| variables of that type. The tasks use DELAYs to cooperate.
--| The DELAY causes another task to get a turn in the CPU.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  -- A task type has a specification
  TASK TYPE SimpleTask (Message: Character; HowMany: Positive);

  -- A task type has a body
  TASK BODY SimpleTask IS

  BEGIN -- SimpleTask

    FOR Count IN 1..HowMany LOOP
      Ada.Text_IO.Put(Item => "Hello from Task " & Message);
      Ada.Text_IO.New_Line;
      DELAY 0.1;            -- lets another task have the CPU
    END LOOP;
```

```
  END SimpleTask;

  -- Now we declare two variables of the type
  Task_A: SimpleTask(Message => 'A', HowMany => 5);
  Task_B: SimpleTask(Message => 'B', HowMany => 7);

BEGIN -- Two_Cooperating_Tasks

-- Task_A and Task_B will both start executing as soon as control
-- reaches this point, again before any of the main program's
-- statements are executed. The Ada standard does not specify
-- which task will start first.

  NULL;

END Two_Cooperating_Tasks;
```

This time, the execution output is interleaved.

```
Hello from Task B
Hello from Task A
Hello from Task B
Hello from Task A
Hello from Task B
Hello from Task A
Hello from Task B
Hello from Task A
Hello from Task B
Hello from Task A
Hello from Task B
Hello from Task B
```

Controlling the Starting Order of Tasks

We know that the Ada standard does not specify an order of activation for multiple tasks in the same program. Each compiler can use a different order; indeed, a compiler is—theoretically—free to use a different starting order each time the program is run, though practical compilers rarely, if ever, take advantage of this freedom.

Although we cannot control the actual activation order of tasks, we can gain control of the order in which these tasks start to do their work by using a so-called “start button.” This is a special case of a *task entry,* which is a point in a task at which it can *synchronize* with other tasks. This is illustrated in Program 15.4.

Program 15.4
Using “Start Buttons” to Control Tasks’ Starting Order

```
WITH Ada.Text_IO;
PROCEDURE Start_Buttons IS
------------------------------------------------------------------------
--| Show the declaration of a simple task type and three
--| variables of that type. The tasks use DELAYs to cooperate.
--| "Start button" entries are used to to control starting order.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  TASK TYPE SimpleTask (Message: Character; HowMany: Positive) IS

    -- This specification provides a "start button" entry.
```

```
    ENTRY StartRunning;

  END SimpleTask;

  TASK BODY SimpleTask IS

  BEGIN -- SimpleTask

    -- The task will "block" at the ACCEPT, waiting for the "button"
    -- to be "pushed" (called from another task, Main in this case).
    ACCEPT StartRunning;

    FOR Count IN 1..HowMany LOOP
      Ada.Text_IO.Put(Item => "Hello from Task " & Message);
      Ada.Text_IO.New_Line;
      DELAY 0.1;            -- lets another task have the CPU
    END LOOP;

  END SimpleTask;

  -- Now we declare three variables of the type
  Task_A: SimpleTask(Message => 'A', HowMany => 5);
  Task_B: SimpleTask(Message => 'B', HowMany => 7);
  Task_C: SimpleTask(Message => 'C', HowMany => 4);

BEGIN -- Start_Buttons

-- Tasks will all start executing as soon as control
-- reaches this point, but each will block on its ACCEPT
-- until the entry is called. In this way we control the starting
-- order of the tasks.

  Task_B.StartRunning;
  Task_A.StartRunning;
  Task_C.StartRunning;

END Start_Buttons;
```

The execution output is

```
Hello from Task B
Hello from Task A
Hello from Task C
Hello from Task B
Hello from Task A
Hello from Task C
Hello from Task B
Hello from Task A
Hello from Task C
Hello from Task B
Hello from Task A
Hello from Task C
Hello from Task B
Hello from Task A
Hello from Task B
Hello from Task B
```

In this program, the task specification is expanded to include an entry specification:

```
ENTRY StartRunning;
```

This is, syntactically, similar to the subprogram specifications that usually appear in package specifications. The task body includes, immediately after its BEGIN, the corresponding line

```
ACCEPT StartRunning;
```

According to the rules of Ada, a SimpleTask object, upon reaching an ACCEPT statement, must *wait* at that statement until the corresponding entry is called by another task. In Program 15.4, then, each task—Task_A, Task_B, and Task_C—is activated just after the main program's BEGIN, but—before it starts any work—each reaches its respective ACCEPT and must wait there (in this simple case, possibly forever) until the entry is called.

How is the entry called? In our first three examples, the main program had nothing to do. In this case, its job is to "press the start buttons" of the three tasks, with the entry call statements

```
Task_B.StartRunning;
Task_A.StartRunning;
Task_C.StartRunning;
```

These statements are syntactically similar to procedure calls. The first statement "presses the start button" of Task_B. Since Task_B was waiting for the button to be pressed, it accepts the call and proceeds with its work.

The main program is apparently executing—in this case, pressing the start buttons—"in parallel" with the three tasks. In fact, this is true. In a program with multiple tasks, the Ada run-time system treats the main program as a task as well.

A task body can contain code that is much more interesting than what we have seen. Ada provides the SELECT statement to give a programmer much flexibility in coding task bodies. For example, using the SELECT,

- The ACCEPT statement can be written to "time out" if a call is not received within a given time interval.
- The task can be made to terminate—end its execution—if the call is never received.
- The task specification can provide a number of entries and its body can be made to respond to whichever of a set of different entry calls occurs first, then loop around and respond again.

The SELECT construct is one of the most interesting in all of programming. We will return to it a bit later, in Section 15.4, when we introduce a bank simulation.

In this section, we have seen the basics of task types and objects. We will now introduce protected types and objects.

15.3 Ada Structures: Protected Types and Protected Objects

In Section 15.1, we discussed mutual exclusion, using the example of an e-mail reader. Here we look at an analogous, but simpler, situation. Suppose we have a three-task program like Program 15.4, but we want each task to write its output in its own area of the screen. The desired output is

```
Hello from Task A        Hello from Task B        Hello from Task C
Hello from Task A        Hello from Task B        Hello from Task C
Hello from Task A        Hello from Task B        Hello from Task C
Hello from Task A        Hello from Task B        Hello from Task C
Hello from Task A        Hello from Task B
                         Hello from Task B
                         Hello from Task B
```

copyright © 1996, Michael B. Feldman January 26, 2003

This simple example is representative of multiwindow programs. We modify the task specification to read

```
TASK TYPE SimpleTask (Message: Character;
                      HowMany: Screen.Depth;
                      Column:  Screen.Width) IS . . .
```

adding a third discriminant, `Column`, to indicate which screen column each task should use for the first character of its repeated message. Further, we modify the main loop of the task body as follows:

```
FOR Count IN 1..HowMany LOOP
  Screen.MoveCursor(Row => Count, Column => Column);
  Ada.Text_IO.Put(Item => "Hello from Task " & Message);
  DELAY 0.5;            -- lets another task have the CPU
END LOOP;
```

That is, the task positions the screen cursor in the proper column before writing the message. Program 15.5 shows the full program.

Program 15.5
Several Tasks Using the Screen

```
WITH Ada.Text_IO;
WITH Screen;
PROCEDURE Columns IS
------------------------------------------------------------------------
--| Shows tasks writing into their respective columns on the
--| screen. This will not always work correctly, because if the
--| tasks are time-sliced, one task may lose the CPU before
--| sending its entire "message" to the screen. This may result
--| in strange "garbage" on the screen.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  TASK TYPE SimpleTask (Message: Character;
                        HowMany: Screen.Depth;
                        Column:  Screen.Width) IS

    -- This specification provides a "start button" entry.
    ENTRY StartRunning;

  END SimpleTask;

  TASK BODY SimpleTask IS

  BEGIN -- SimpleTask

    -- Each task will write its message in its own column
    ACCEPT StartRunning;

    FOR Count IN 1..HowMany LOOP
      Screen.MoveCursor(Row => Count, Column => Column);
      Ada.Text_IO.Put(Item => "Hello from Task " & Message);
      DELAY 0.5;            -- lets another task have the CPU
    END LOOP;

  END SimpleTask;

  -- Now we declare three variables of the type
  Task_A: SimpleTask(Message => 'A', HowMany => 5, Column => 1);
```

```
  Task_B: SimpleTask(Message => 'B', HowMany => 7, Column => 26);
  Task_C: SimpleTask(Message => 'C', HowMany => 4, Column => 51);

BEGIN -- Columns

  Screen.ClearScreen;
  Task_B.StartRunning;
  Task_A.StartRunning;
  Task_C.StartRunning;

END Columns;
```

Here is the execution output:

```
Hello from Task A        Hello from Task B        Hello from Task C
                                                  2Hello from Task C;26f[2;1fHello
from Task AHello from Task B                   [[3;1fHello from Task A3;26fHello
from Task BHello from Task C4;4;1fHello from Task A51fHello from Task C26fHello
from Task B5;526;f1fHello from Task BHello from Task A
                         Hello from Task B
                         Hello from Task B
```

The output from running this program is not exactly what we intended! Instead of the desired neat columns, we got messages displayed in seemingly random locations, interspersed with apparent "garbage" like

```
C;26f[2;1f
```

What happened here? To understand this, recall the body of `Screen.MoveCursor` (Program 2.16):

```
PROCEDURE MoveCursor (Column : Width; Row : Depth) IS
BEGIN
  Ada.Text_IO.Put (Item => ASCII.ESC);
  Ada.Text_IO.Put ("[");
  Ada.Integer_Text_IO.Put (Item => Row, Width => 1);
  Ada.Text_IO.Put (Item => ';');
  Ada.Integer_Text_IO.Put (Item => Column, Width => 1);
  Ada.Text_IO.Put (Item => 'f');
END MoveCursor;
```

Positioning the cursor requires an instruction, up to eight characters in length, to the ANSI terminal software: the `ESC` character, then `'['`, followed by a possibly two-digit `Row`, then `';'`, then a possibly two-digit `Column` value, and finally `'F'`. Once it receives the entire instruction, the terminal actually moves the cursor on the screen.

Suppose the `MoveCursor` call is issued from within a task, as in the present example. Suppose further that in this case the Ada run-time system *does* provide time-slicing to produce "parallel" behavior by multiple tasks. It is quite possible that the task's quantum will expire after only some of the eight characters have been sent to the terminal, and then another task will attempt to write something to the terminal. In this case, the terminal never recognized the first instruction, because it received only part of it, so instead of obeying the instruction, it just displays the characters. The "garbage" string above, `C;26f[2;1f`, consists of pieces from several different intended instructions.

This problem arose because a task was interrupted in mid-instruction and then another task was allowed to begin its own screen instruction. This is called a *race condition,* because two tasks are, effectively, in a race to write to the screen, with unpredictable results. It is actually a readers-writers problem: Multiple tasks are interfering with each other's attempts to write to the screen.

To prevent this problem from ruining our columnar output, we must *protect* the screen so that—whether or not we have time-slicing—a task is allowed to finish an entire display operation before another task can begin one. We do this in Ada with a protected type, as shown in Program 15.6.

Program 15.6
Using a Protected Type to Ensure Completion of a Screen Action

```
WITH Ada.Text_IO;
WITH Screen;
PROCEDURE Protect_Screen IS
------------------------------------------------------------------------
--| Shows tasks writing into their respective columns on the
--| screen. This time we use a protected type, whose procedure
--| can be executed by only one task at a time.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  PROTECTED TYPE ScreenManagerType IS

  -- If multiple calls of Write are made simultaneously, each is
  -- executed in its entirety before the next is begun.
  -- The Ada standard does not specify an order of execution.

    PROCEDURE Write (Item:   IN String;
                     Row:    IN Screen.Depth;
                     Column: IN Screen.Width);

  END ScreenManagerType;

  PROTECTED BODY ScreenManagerType IS

    PROCEDURE Write (Item:   IN String;
                     Row:    IN Screen.Depth;
                     Column: IN Screen.Width) IS
    BEGIN -- Write

      Screen.MoveCursor(Row => Row, Column => Column);
      Ada.Text_IO.Put(Item => Item);

    END Write;

  END ScreenManagerType;

  Manager: ScreenManagerType;

  TASK TYPE SimpleTask (Message: Character;
                        HowMany: Screen.Depth;
                        Column:  Screen.Width) IS

    -- This specification provides a "start button" entry.
    ENTRY StartRunning;

  END SimpleTask;

  TASK BODY SimpleTask IS

  BEGIN -- SimpleTask

    -- Each task will write its message in its own column
    -- Now the task locks the screen before moving the cursor,
    -- unlocking it when writing is completed.

    ACCEPT StartRunning;
```

copyright © 1996, Michael B. Feldman

```
    FOR Count IN 1..HowMany LOOP

      -- No need to lock the screen explicitly; just call the
      -- protected procedure.
      Manager.Write(Row => Count, Column => Column,
                    Item => "Hello from Task " & Message);

      DELAY 0.5;            -- lets another task have the CPU
    END LOOP;

  END SimpleTask;

  -- Now we declare three variables of the type
  Task_A: SimpleTask(Message => 'A', HowMany => 5, Column => 1);
  Task_B: SimpleTask(Message => 'B', HowMany => 7, Column => 26);
  Task_C: SimpleTask(Message => 'C', HowMany => 4, Column => 51);

BEGIN -- Protect_Screen

  Screen.ClearScreen;
  Task_B.StartRunning;
  Task_A.StartRunning;
  Task_C.StartRunning;

END Protect_Screen;
```

In this program, we declare a type

```
PROTECTED TYPE ScreenManagerType IS

  PROCEDURE Write (Item:   IN String;
                   Row:    IN Screen.Depth;
                   Column: IN Screen.Width);

END ScreenManagerType;

Manager: ScreenManagerType;
```

An object of this type—in this case, `Manager`—provides a procedure `Write` to which all the parameters of the desired screen operation are passed: the string to be written, the row, and the column. Any task wishing to write to the screen must do so by passing these parameters to the screen manager. The `SimpleTask` body now contains the call

```
Manager.Write(Row => Count, Column => Column,
              Item => "Hello from Task " & Message);
```

as required. The body of the protected type is

```
PROTECTED BODY ScreenManagerType IS

  PROCEDURE Write (Item:   IN String;
                   Row:    IN Screen.Depth;
                   Column: IN Screen.Width) IS
  BEGIN -- Write

    Screen.MoveCursor(Row => Row, Column => Column);
    Ada.Text_IO.Put(Item => Item);

  END Write;
```

```
END ScreenManagerType;
```

and the `Write` procedure encapsulates the `MoveCursor` and `Put` operations. `Write` is a *protected procedure.*

What is the difference between this protected write procedure and an ordinary procedure? Ada guarantees that each call of a protected procedure will complete before another call can be started. Even if several tasks are running, trading control of the CPU among them, a task will not be allowed to start a protected procedure call if another call of the same procedure, or any other procedure of the same protected object, is still incomplete. In our case, this provides the necessary mutual exclusion for the screen.

Protected types can provide functions and entries in addition to procedures. Protected functions allow multiple tasks to examine a data structure simultaneously but not to modify the data structure. Protected entries have some of the properties of both task entries and protected procedures. A detailed discussion of these is beyond our scope here.

15.4 Data Structures: The Task as a Data Structure

We mentioned earlier in this chapter that a task has characteristics resembling those of a procedure, of a package, and of a data structure. We have seen examples of the first two; we will now consider the third.

So far, we have declared task types and task variables. In Program 15.7, we declare an array of tasks, with the declaration

```
Family: ARRAY (1..3) OF SimpleTask;
```

Program 15.7
Creating an Array of Tasks

```
WITH Ada.Text_IO;
WITH Screen;
PROCEDURE Task_Array IS
------------------------------------------------------------------------
--| Shows tasks writing into their respective columns on the
--| screen. This time we use a protected type, whose procedure
--| can be executed by only one task at a time.
--| The task objects are stored in an array, and receive their
--| configuration parameters through "start buttons" rather than
--| discriminants.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: December 1995
------------------------------------------------------------------------

  PROTECTED TYPE ScreenManagerType IS

  -- If multiple calls of Write are made simultaneously, each is
  -- executed in its entirety before the next is begun.
  -- The Ada standard does not specify an order of execution.

    PROCEDURE Write (Item:   IN String;
                     Row:    IN Screen.Depth;
                     Column: IN Screen.Width);

  END ScreenManagerType;

  PROTECTED BODY ScreenManagerType IS

    PROCEDURE Write (Item:   IN String;
```

```
                     Row:    IN Screen.Depth;
                     Column: IN Screen.Width) IS
    BEGIN -- Write

      Screen.MoveCursor(Row => Row, Column => Column);
      Ada.Text_IO.Put(Item => Item);

    END Write;

  END ScreenManagerType;

  Manager: ScreenManagerType;

  TASK TYPE SimpleTask IS

    -- Task receives its parameters through a start-button entry
    -- instead of discriminants This is more flexible..
    ENTRY StartRunning (Message: Character;
                        HowMany: Screen.Depth;
                        Column:  Screen.Width);

  END SimpleTask;

  TASK BODY SimpleTask IS

    MyMessage: Character;
    MyCount  : Screen.Depth;
    MyColumn : Screen.Width;

  BEGIN -- SimpleTask

    -- Each task will write its message in its own column
    -- Now the task locks the screen before moving the cursor,
    -- unlocking it when writing is completed.

    ACCEPT StartRunning (Message: Character;
                        HowMany: Screen.Depth;
                        Column:  Screen.Width) DO
      MyMessage := Message;
      MyCount   := HowMany;
      MyColumn  := Column;

    END StartRunning;

    FOR Count IN 1..MyCount LOOP

      -- No need to lock the screen explicitly; just call the
      -- protected procedure.
      Manager.Write(Row => Count, Column => MyColumn,
                    Item => "Hello from Task " & MyMessage);

      DELAY 0.5;            -- lets another task have the CPU
    END LOOP;

  END SimpleTask;

  Family: ARRAY (1..3) OF SimpleTask;
  Char  : CONSTANT Character := 'A';

BEGIN -- Task_Array;

  Screen.ClearScreen;
  FOR Which IN Family'Range LOOP
    Family(Which).StartRunning
```

```
      (Message => Character'Val(Character'Pos(Char) + Which),
       HowMany => 3 * Which,
       Column  => 3 + (24 * (Which - 1)));
  END LOOP;

END Task_Array;
```

This program creates three task objects, just as declaring an array of three integers would create three integer objects. We refer to the task objects with array subscripts, as in an ordinary array. In this case, each task has a "start button" entry

```
ENTRY StartRunning (Message: Character;
                    HowMany: Screen.Depth;
                    Column:  Screen.Width);
```

and we call each task's respective entry in the loop

```
FOR Which IN Family'Range LOOP
  Family(Which).StartRunning
    (Message => Character'Val(Character'Pos(Char) + Which),
     HowMany => 3 * Which,
     Column  => 3 + (24 * (Which - 1)));
END LOOP;
```

In this program, we have passed each task's parameters in the "start button" instead of using the discriminants of earlier examples.

It is also possible to declare a task as a field of a record. Finally, it is possible to declare an access type such as

```
TYPE TaskPointer IS ACCESS SimpleTask;
```

Then, given a variable

```
Task_1: TaskPointer;
```

we can allocate a task dynamically, like any other dynamic data structure:

```
Task_1 := NEW SimpleTask;
```

The task starts running when it is allocated; we can call its entry with a statement such as

```
Task_1.ALL.StartRunning(Message => 'Z', HowMany => 10, Column => 20);
```

or, more concisely,

```
Task_1.StartRunning(Message => 'Z', HowMany => 10, Column => 20);
```

Further examples of tasks as record fields, and of dynamically allocated tasks, are beyond the scope of this book.

Because a task type is a *type,* it makes sense to ask how it is related to the overall Ada type system. Specifically, which operations are available for task types? The answer is that task types are similar to `LIMITED PRIVATE` types—no operations at all are predefined for them. Task objects can be declared, but

the only operations are those provided by entries. In particular, assignment and equality test are *not* available.

Having introduced the basics of task types and protected types through a series of simple examples, we now proceed to two extended aplications: a bank simulation and the Dining Philosophers.

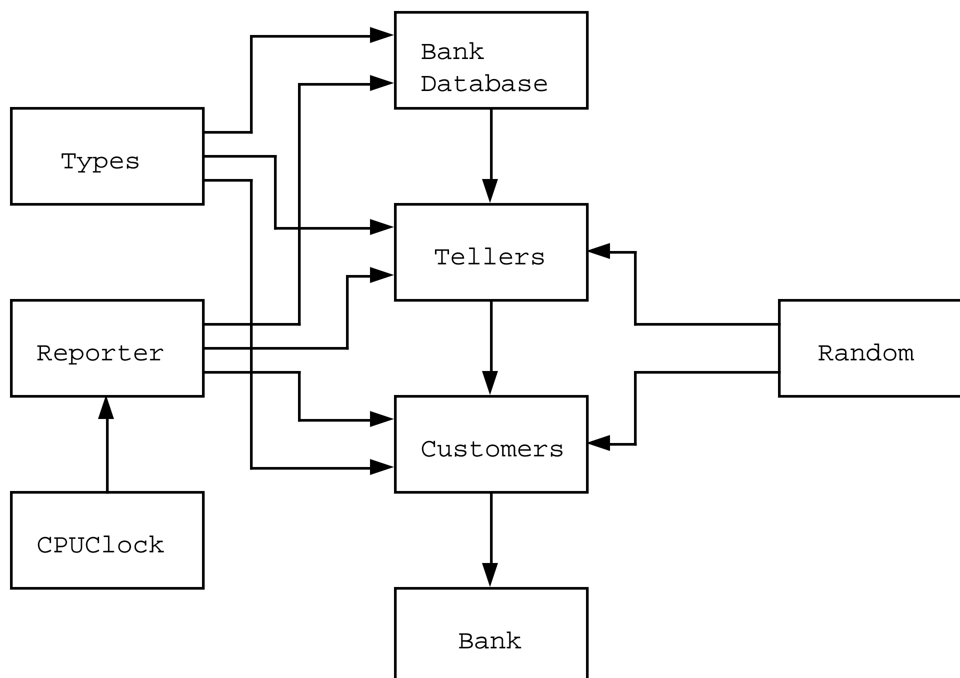
15.5 Application: Simulation of a Bank

One interesting application of concurrent programming is simulating the behavior of a physical system or a real-life situation. For example, it is relatively straightforward to simulate a customer/server environment such as a bank. The objects in our model of a bank are:

- The *bank* itself
- A set of *customers*, who visit the bank periodically to open accounts, deposit money, and withdraw money
- A set of *tellers,* the bank employees who are responsible for interacting with the customers
- The *database,* a file of account information to which the tellers have access

We will model the bank using a main program for the bank and packages for the database, tellers, and customers. The dependencies are shown in Figure 15.1; the bank and the three packages are shown in the middle column; to the right and the left are other packages that provide services to them. As in other dependency diagrams, an arrow from one unit to a second unit means that the second is a client of the first. `CPUClock` refers to the package introduced in Programs 3.17 and 3.18; `Random` refers to `Ada.Numerics.Discrete_Random` (details in Appendix F). We will shortly introduce `Types` and `Reporter`.

Figure 15.1
Package Dependencies for Bank Simulation



Executing the main program simulates a time period in the life of the bank. Here is a sample of the line-by-line output from the simulation after it has been running for a while:

```
T = 11 Account 3 depositing 398 with Teller C
T = 11   Teller B: Acct 1 - Balance is 0
T = 11 Account 1 will return after 2 sec
T = 13 Account 1 withdrawing 179 with Teller B
T = 13   Teller B: Transaction will take 6 sec
T = 15 Account 2 checking balance with Teller A
T = 15   Teller A: Transaction will take 7 sec
T = 15   Teller C: Acct 5 - Balance is 0
T = 15   Teller C: Transaction will take 1 sec
T = 15 Account 5 alive.
T = 15 Account 5 will return after 5 sec
T = 16   Teller C: Acct 3 - Balance is 398
T = 16 Account 3 will return after 6 sec
T = 17 Account 4 withdrawing 816 with Teller C
T = 17   Teller C: Transaction will take 1 sec
T = 18   Teller C: Acct 4 - InsufficientFunds
T = 18 Account 4 will return after 9 sec
```

Each line of the output begins with a "time stamp"—for example, `T = 15`—that gives the number of seconds that elapsed since the start of the run. The line

```
T = 11 Account 1 will return after 2 sec
```

indicates that Customer 1 has completed a transaction, and will come back for another one after 2 seconds have elapsed. The line

```
T = 13 Account 1 withdrawing 179 with Teller B
```

indicates that Customer 1 communicates to Teller B its desire to withdraw $179. The line

```
T = 13   Teller B: Transaction will take 6 sec
```

indicates that Teller B expects this transaction to take 6 seconds. We will see shortly how these times and amounts are generated.

The Bank Main Program and the Package Specifications

First we examine Program 15.8, the main program `Bank`. This program has nothing to do but cause the `Customers` and `Tellers` packages to be elaborated (by the `WITH` clauses).

Program 15.8
Body of the Bank Main Program

```
WITH Customers; USE Customers;
WITH Tellers; USE Tellers;
PROCEDURE Bank IS
------------------------------------------------------------------------
--| Main program for bank simulation; it does nothing but cause
--| the customers and tellers to come into existence.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

BEGIN -- Bank

  NULL;

END Bank;
```

Next, we look at the various package specifications. Program 15.9 provides a set of types that are used by all the other packages.

Program 15.9
Some Types Needed by the Bank Simulation

```
PACKAGE Types IS
------------------------------------------------------------------------
--| Types package for the bank simulation. This contains only public
--| declarations, and therefore needs no package body.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

   -- These 2 constants can be altered to change
    --  the behavior of the simulation
    SUBTYPE TellerRange IS Character RANGE 'A'..'C';
    NumberOfCustomers : CONSTANT Integer := 5;

    -- These 2 ranges can be altered to change
    --  the behavior of the simulation
    SUBTYPE TransactionTimeRange IS Integer RANGE 1 .. 7;
    SUBTYPE TimeBetweenVisitsRange IS Integer RANGE 1 .. 11;

    -- Global types
    SUBTYPE Money IS Integer RANGE 0 .. Integer'LAST;
    TYPE Status IS (OK, InsufficientFunds, BadCustId);
    SUBTYPE CustId IS Integer RANGE 0 .. NumberOfCustomers;

END Types;
```

How shall we model the customers? We declare a task type `Customer`; each task object represents one human customer. This task type has no entries because the customer objects are created at the start of the run, but do not need to be called. They simply live out their lives, occasionally making transactions at the bank. The customer type, and an array of customers, are defined in Program 15.10.

Program 15.10
Specification of the Customer Package

```
WITH Types; USE Types;
PACKAGE Customers IS
------------------------------------------------------------------------
--| Customer package for bank siumulation. Each customer is
--| a task object.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

    TASK TYPE Customer IS  -- Requestor task type, no entries
    END Customer;

    CustomerGroup : ARRAY (1 .. NumberOfCustomers) OF Customer;

END Customers;
```

Program 15.11 shows the specification for `Tellers`.

Program 15.11
Specification of the Teller Package

```
WITH Types; USE Types;
PACKAGE Tellers IS
```

```
------------------------------------------------------------------------
--| Teller package for bank siumulation
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  TASK TYPE Teller (TellerID: TellerRange) IS
    -- Entries to do simple transactions and return status
    ENTRY NewAcct (ID   : OUT CustID;
                   Stat: OUT Status);
    ENTRY Deposit (ID   : CustID;
                   Val  : IN Money;
                   Stat: OUT Status);
    ENTRY Withdraw (ID   : CustID;
                    Val  : IN Money;
                   Stat: OUT Status);
    ENTRY Balance (ID   : CustID;
                   Stat: OUT Status);
  END Teller;

  -- declare tellers and give them "names"
  A: ALIASED Teller(TellerID => 'A');
  B: ALIASED Teller(TellerID => 'B');
  C: ALIASED Teller(TellerID => 'C');

  TYPE TellerPointer IS ACCESS ALL Teller;

  -- a bank full of tellers
  TellerGroup : ARRAY (TellerRange) OF TellerPointer :=
    (A'Access, B'Access, C'Access);

END Tellers;
```

Here we have declared the task type `Teller` with a discriminant to initialize the teller's "name."

Each teller object has four entries that customers can call:

- `NewAcct` opens a new account for a customer.
- `Deposit` processes a customer's deposit transaction.
- `Withdraw` processes a customer's withdrawal transaction.
- `Balance` allows a customer to view its account balance.

Each operation returns a status code to the customer; the codes—`OK`, `InsufficientFunds`, and `BadCustId`—are defined in `Types`.

Because each teller object needs a different value of the discriminant (each teller needs a unique name), an array of tellers is heterogeneous. We cannot simply declare an array of tellers, but must instead declare an array of *pointers* to tellers, and initialize this array with pointers to the three teller objects. This is similar to the technique used in Program 9.8.

Program 15.12 gives the specification for `Database`. The database manager provides operations documented by the postconditions. It is a protected object, which provides mutual exclusion in case several tellers make concurrent calls of the database operations.

Program 15.12
Specification of the Database Package

```
WITH Types; USE Types;
PACKAGE Database IS
```

```
------------------------------------------------------------------------
--| Maintains bank's internal data about open accounts and balances
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  PROTECTED Manager IS

    -- All these procedures are protected, so only one call at a time
    -- will be executed, even if the calls arrive concurrently.

    PROCEDURE EnterCustID (ID : OUT CustID; Stat  : OUT Status);
    -- Pre:  None
    -- Post: ID is the next available customer ID; Stat is OK.

    PROCEDURE Deposit (ID : IN CustID; Amount : IN Money;
                       NewBalance : OUT Money; Stat : OUT Status);
    -- Pre:  ID and Amount are defined
    -- Post: If ID is valid, NewBalance is the resulting balance
    --       and Stat is OK; otherwise, Stat is BadCustID.

    PROCEDURE Withdraw (ID : IN CustID; Amount : IN Money;
                         NewBalance : OUT Money; Stat : OUT Status);
    -- Pre:  ID and Amount are defined
    -- Post: If ID is valid and NewBalance would be nonnegative,
    --       Stat is OK and NewBalance is returned.
    --       If ID is invalid, Stat is BadCustID; if NewBalance
    --       would be negative, Stat is InsufficientFunds

    PROCEDURE Balance (ID : IN CustID; Amount : OUT Money;
                       Stat : OUT Status);
    -- Pre:  ID is defined
    -- Post: If ID is invalid, Stat is BadCustID; otherwise,
    --       Stat is OK and Amount is current balance
  END Manager;

END Database;
```

Finally, we turn to the specification for `Reporter`, shown in Program 15.13. The protected object `ScreenManager` provides an operation `Put`, which is protected, so only one call at a time can be executed.

Program 15.13
Specification of the Reporter Package

```
PACKAGE Reporter IS
------------------------------------------------------------------------
--| Reporter package for bank simulation; Put is protected
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  PROTECTED ScreenManager IS

    PROCEDURE Put(Message: IN String);

  END ScreenManager;

END Reporter;
```

The Package Bodies in the Bank Simulation

We now turn to the bodies of the various packages. First look at `Reporter`, in Program 15.14. The `Put` procedure uses `CPUClock` to produce a "time stamp" that indicates the number of seconds elapsed since the start of the simulation. Note that this package body has its own executable statement part, following the `BEGIN`. The two statements there are executed once, when `Reporter` is elaborated at the start of the program execution. The `New_Line` statement is there because Ada input/output is usually buffered, and flushing the buffer with a `New_Line` call ensures that screen output will begin immediately after the program starts.

Program 15.14
Body of the Reporter Package

```
WITH Ada.Text_IO;
WITH CPUClock; USE CPUClock;
PACKAGE BODY Reporter IS
------------------------------------------------------------------------
--| Body of Reporter - a simple screen protector
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  PROTECTED BODY ScreenManager IS

    PROCEDURE Put(Message: IN String) IS
    BEGIN -- Put
      Ada.Text_IO.Put("T =" & Integer'Image(Integer(CPUTime))
                 & " " & Message);
      Ada.Text_IO.New_Line;
    END Put;

  END ScreenManager;

BEGIN -- Reporter

  -- These two lines are executed once, when the package is elaborated.
  ResetCPUTime;
  Ada.Text_IO.New_Line; -- gets output going

END Reporter;
```

Let us look next at the body of `Customers`. Each customer task opens an account, then executes a main loop that waits a random amount of time before beginning a transaction, then constructs a transaction from a random teller, a random transaction type, and a random amount of money. The customer then calls the appropriate entry of the selected teller. This requires four random number generators—instances of `Ada.Numerics.Discrete_Random`—each with its own range of random quantities.

Program 15.15
Body of the Customer Package

```
WITH Reporter; USE Reporter;
WITH Ada.Numerics.Discrete_Random;
WITH Tellers; USE Tellers;
WITH Types; USE Types;
PACKAGE BODY Customers IS
------------------------------------------------------------------------
--| Body of customer package. Each customer executes ten random
--| transactions with random tellers before terminating.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------
```

```
TYPE Transactions IS (Deposit, Withdraw, Balance);

PACKAGE RandomTransactions IS
  NEW Ada.Numerics.Discrete_Random (Transactions);

PACKAGE RandomTellers IS
  NEW Ada.Numerics.Discrete_Random (TellerRange);

SUBTYPE MoneyRange IS Money RANGE 1 .. 999;
PACKAGE RandomAmounts IS
  NEW Ada.Numerics.Discrete_Random (MoneyRange);

PACKAGE RandomTimesBetweenVisits IS
  NEW Ada.Numerics.Discrete_Random (TimeBetweenVisitsRange);

TASK BODY Customer IS

  -- Local variables
  ID               : CustID;
  Amount           : Money;
  Stat             : Status;
  WaitTime         : TimeBetweenVisitsRange;
  Teller           : TellerRange;
  Transaction      : Transactions;
  NumTransactions : CONSTANT Integer := 10;

  T: RandomTransactions.Generator;
  R: RandomTellers.Generator;
  A: RandomAmounts.Generator;
  V: RandomTimesBetweenVisits.Generator;

BEGIN -- Customer

  RandomTransactions.Reset(T);
  RandomTellers.Reset(R);
  RandomAmounts.Reset(A);
  RandomTimesBetweenVisits.Reset(V);

  Teller := RandomTellers.Random(R);
  TellerGroup(Teller).NewAcct(ID, Stat); -- Get new cust id
  ScreenManager.Put ("Account" & Integer'Image(ID) & " alive.");

  FOR I IN 1 .. NumTransactions LOOP

    WaitTime := RandomTimesBetweenVisits.Random(V);
    ScreenManager.Put("Account" & Integer'Image(ID)
      & " will return after" & Integer'Image(WaitTime)
      & " sec");
    DELAY Duration (WaitTime);

    Teller := RandomTellers.Random(R);
    Transaction := RandomTransactions.Random(T);
    Amount := RandomAmounts.Random(A);

    CASE Transaction IS -- Pick random transaction
      WHEN Deposit =>
        ScreenManager.Put("Account" & Integer'Image(ID)
          & " depositing" & Integer'Image(Amount)
          & " with Teller " & Teller);
        TellerGroup (Teller).Deposit (ID, Amount, Stat);
      WHEN Withdraw =>
        ScreenManager.Put("Account" & Integer'Image(ID)
          & " withdrawing" & Integer'Image(Amount)
```

copyright © 1996, Michael B. Feldman

```
          & " with Teller " & Teller);
        TellerGroup (Teller).Withdraw (ID, Amount, Stat);
      WHEN Balance =>
        ScreenManager.Put("Account" & Integer'Image(ID)
          & " checking balance"
          & " with Teller " & Teller);
        TellerGroup (Teller).Balance (ID, Stat);
    END CASE;
  END LOOP;
  ScreenManager.Put("Account" & Integer'Image(ID) & " closed.");
 END Customer;

END Customers;
```

Program 15.16 shows the body of `Database`. This is very straightforward; the protected object `Manager` performs operations on the array `Accounts`. Each account is represented by a record that contains a flag (which is set when the account is opened) and shows the current balance. The four operations of the database manager are all protected: Only one at a time can be executed even if there are simultaneous calls from multiple tasks.

Program 15.16
Body of the Database Package

```
WITH Reporter; USE Reporter;
WITH Types; USE Types;
PACKAGE BODY Database IS
------------------------------------------------------------------------
--| Body of database package. The protected procedures ensure that
--| only one call at a time is exexuted.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  TYPE AccountRecord IS RECORD
    Valid: Boolean := False;
    Balance: Money := 0;
  END RECORD;

  TYPE AccountType IS ARRAY (CustID) OF AccountRecord;

  Next: CustID;
  Accounts: AccountType;

  PROTECTED BODY Manager IS

    PROCEDURE EnterCustID (ID: OUT CustID; Stat: OUT Status) IS
    BEGIN
      Next := Next + 1;
      ID := Next;
      Accounts(ID).Valid := True;
      Stat := OK;
    END EnterCustID;

    PROCEDURE Deposit (ID: CustID; Amount: IN Money;
                       NewBalance: OUT Money; Stat: OUT Status) IS
    BEGIN
      IF NOT Accounts(ID).Valid THEN
        Stat := BadCustID;
      ELSE
        Accounts(ID).Balance := Accounts(ID).Balance + Amount;
        NewBalance := Accounts(ID).Balance;
        Stat := OK;
```

```
      END IF;
    END Deposit;

    PROCEDURE Withdraw (ID: CustID; Amount  IN Money;
                    NewBalance: OUT Money; Stat: OUT Status) IS
    BEGIN
      IF NOT Accounts(ID).Valid THEN
        Stat := BadCustID;
      ELSIF Accounts(ID).Balance - Amount <= 0 THEN
        Stat := InsufficientFunds;
      ELSE
        Accounts(ID).Balance := Accounts(ID).Balance + Amount;
        NewBalance := Accounts(ID).Balance;
        Stat := OK;
      END IF;
    END Withdraw;

    PROCEDURE Balance (ID: CustID; Amount: OUT Money;
                       Stat: OUT Status) IS
    BEGIN
      IF NOT Accounts(ID).Valid THEN
        Stat := BadCustID;
      ELSE
        Amount := Accounts(ID).Balance;
        Stat := OK;
      END IF;
    END Balance;

  END Manager;

END Database;
```

Finally, Program 15.17 shows the body of `Tellers`. The `Teller` task body contains two auxiliary procedures: `SimulateWait` selects a random length of time and then waits that long. This simulates the varying length of time taken by a transaction in an actual bank. `ReportResult` just assembles a message and sends it to `Reporter`; the message contents depend on the status code returned by the transaction.

Program 15.17
Body of the Tellers Package

```
WITH Reporter; USE Reporter;
WITH Types; USE Types;
WITH Database; USE Database;
WITH Ada.Numerics.Discrete_Random;
PACKAGE BODY Tellers IS
------------------------------------------------------------------------
--| Body of teller package. A teller object just waits a random
--| length of time (to simulate the time of a transaction), then
--| waits for a customer to ask for a transaction.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  PACKAGE RandomTransactionTimes IS
    NEW Ada.Numerics.Discrete_Random (TransactionTimeRange);
  T: RandomTransactionTimes.Generator;

  TASK BODY Teller IS

    NewBalance      : Money;
    Del             : Integer;
```

```
  Stat            : Status;

  PROCEDURE SimulateWait IS
    WaitTime: TransactionTimeRange
      := RandomTransactionTimes.Random(T);
  BEGIN
    ScreenManager.Put
      ("  Teller " & TellerID & ": Transaction will take"
      & Integer'Image(WaitTime) & " sec");
    DELAY Duration(WaitTime);
  END SimulateWait;

  PROCEDURE ReportResult (Stat: Status; TellID: TellerRange;
                          ID: CustID; NewBalance: Money) IS
  BEGIN
    CASE Stat IS
      WHEN OK =>
        ScreenManager.Put("  Teller " & TellerID
          & ": Acct"  & Integer'Image(ID)
          & " - Balance is" & Integer'Image(NewBalance));
      WHEN BadCustID =>
        ScreenManager.Put("  Teller " & TellerID
          & ": Acct"  & Integer'Image(ID)
          & " - Invalid Account Number");
      WHEN InsufficientFunds =>
        ScreenManager.Put("  Teller " & TellerID
          & ": Acct"  & Integer'Image(ID)
          & " - InsufficientFunds");
    END CASE;
  END ReportResult;

BEGIN -- Teller
  RandomTransactionTimes.Reset(T); -- seed random sequence
  ScreenManager.Put("  Teller  " & TellerID  & " - at your service");
  LOOP
    SELECT -- Wait for any transaction request
      ACCEPT NewAcct(Id : OUT CustId; Stat: OUT Status) DO
        SimulateWait;
        Database.Manager.EnterCustID(Id, Stat);
        NewBalance := 0;
        ReportResult (Stat, TellerID, ID, NewBalance);
      END NewAcct;
    OR
      ACCEPT Deposit (Id: CustId; Val: IN Money; Stat: OUT Status) DO
        SimulateWait;
        Database.Manager.Deposit (Id, Val, NewBalance, Stat);
        ReportResult (Stat, TellerID, ID, NewBalance);
      END Deposit;
    OR
      ACCEPT Withdraw (Id: CustId; Val: IN Money; Stat: OUT Status) DO
        SimulateWait;
        Database.Manager.Withdraw (Id, Val, NewBalance, Stat);
        ReportResult (Stat, TellerID, ID, NewBalance);
      END Withdraw;
    OR
      ACCEPT Balance (Id: CustId; Stat: OUT Status) DO
        SimulateWait;
        Database.Manager.Balance (Id, NewBalance, Stat);
        ReportResult (Stat, TellerID, ID, NewBalance);
      END Balance;
    OR
      TERMINATE; -- if no more customers
    END SELECT;
  END LOOP;
```

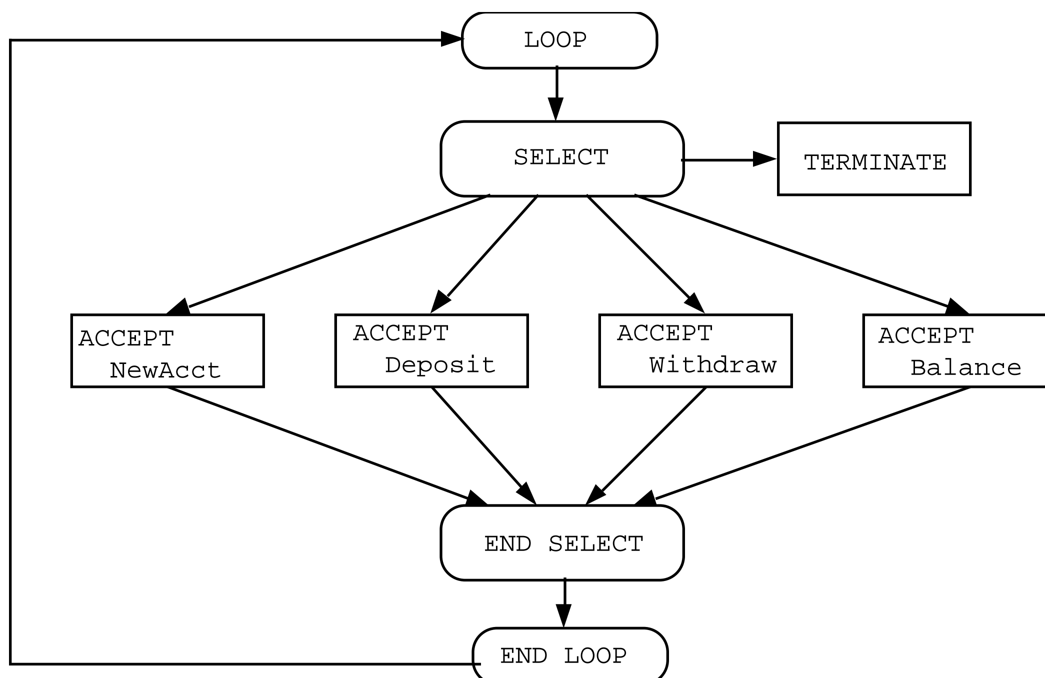
```
  END Teller;

END Tellers;
```

The `SELECT` *Statement*

The teller task body also introduces a very interesting new Ada statement type, the `SELECT` statement. Figure 15.2 gives a flowchart-like depiction of the statement

```
LOOP
  SELECT -- Wait for any transaction request
    ACCEPT NewAcct(Id : OUT CustId; Stat: OUT Status) DO
    . . .
    END NewAcct;
  OR
    ACCEPT Deposit (Id: CustId; Val: IN Money; Stat: OUT Status) DO
    . . .
    END Deposit;
  OR
    ACCEPT Withdraw (Id: CustId; Val: IN Money; Stat: OUT Status) DO
    . . .
    END Withdraw;
  OR
    ACCEPT Balance (Id: CustId; Stat: OUT Status) DO
    . . .
    END Balance;
  OR
    TERMINATE; -- if no more customers
  END SELECT;
END LOOP;
```

Figure 15.2
A Loop with a `SELECT` Statement



copyright © 1996, Michael B. Feldman

The diagram shows that the teller processes *one* transaction at each iteration of the loop. The Ada run-time system provides each entry with a FIFO queue; entry calls are placed in the queues in the order of their arrival. Now consider several cases, assuming the teller has just arrived at the `SELECT` statement:

1. *The bank is quiet; no customers are calling entries.* In this case, the teller waits at the `SELECT` until a customer entry call arrives, accepts that call, and executes the statements within that `DO-END` block. The `DO-END` block is called a *rendezvous;* as in a real bank, the customer waits while the transaction is processed. After executing the rendezvous code, the teller loops around to the `SELECT` again and the customer goes about its other business.

2. *Several customers have called the same entry of the same teller.* In this case, the teller accepts the first call. The teller can accept only the *first* call because the queue is FIFO. The entry call is dequeued, the teller loops around to the `SELECT`, and the customer proceeds with other matters.

3. *Calls are waiting at the heads of more than one entry queue.* In this case, the teller chooses one of these callers and proceeds as in case 2. The mechanism for choosing a caller is not specified by the Ada standard and, therefore, depends on the specific Ada run-time system. To give just two possibilities, the choice can rotate among the queue heads—one queue per iteration—or it can be random.

The fifth `SELECT` alternative, `TERMINATE`, is provided to allow the teller to terminate when there is no more activity on any of its entry queues. Termination conditions are rather complicated to explain for the full generality of Ada tasks; this is a subject beyond the scope of this book. Here we simply say that the teller will select the `TERMINATE` alternative, leave its apparently infinite loop, and quit when there is nothing left for it to do.

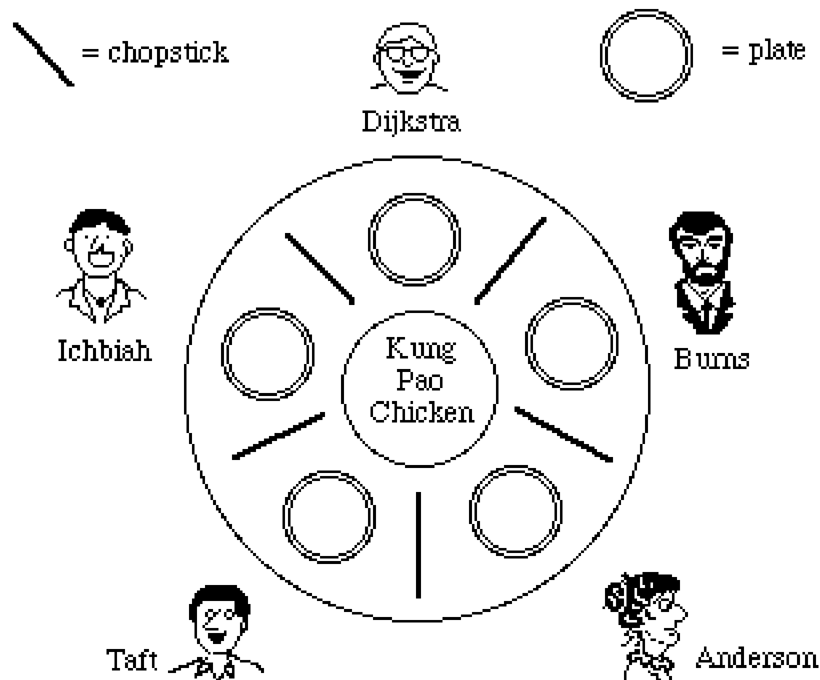
This example has shown the use of tasks to simulate real-life situations. The next, and last, section introduces a more humorous example, which has served for more than 20 years as a vehicle for studying problems of resource allocation and deadlock.

15.6 Application: The Dining Philosophers

Imagine a group of five brilliant philosophers who lead very sheltered lives: Each has nothing to do but think deep thoughts, stopping occasionally to eat a meal. Their plates are automatically refilled from an infinite supply of delicious Chinese food. However, like most philosophers, these thinkers must interrupt their simple lives to solve an especially difficult problem: They sit around a circular table and only five chopsticks—made of titanium and therefore unbreakable—are provided, one chopstick between each pair of plates. Figure 15.3 depicts the philosophers and their dining table.

copyright © 1996, Michael B. Feldman

Figure 15.3
The Ada 95 Philosophical Society



It has become traditional to name the philosophers for major contributors to the field. Our philosophers are (counterclockwise from the top of the cartoon)

- Edsger Dijkstra, the Dutch computer science professor who first described the Dining Philosophers in 1971
- Jean Ichbiah, the French software engineer who led the original Ada design team beginning in the late 1970s
- Tucker Taft, the American software engineer who led the Ada 9X design team and whose work resulted in the Ada 95 standard
- Christine Anderson, the American aerospace engineer who managed the Ada 95 design project for the U.S. government
- Alan Burns, the British professor who has written very wisely and well about concurrency in Ada

How should the philosophers behave? To eat, a philosopher must pick up the chopsticks to his or her immediate left and right. The problem is caused by the circularity of the table: Each left chopstick is also another philosopher's right chopstick. Suppose each philosopher first grabs his or her left-hand chopstick and refuses to put it down while waiting for the right-hand chopstick. In this case, nobody will get to eat and all the philosophers will die of hunger. This state of affairs—each philosopher waiting indefinitely for his or her right-hand neighbor to act—is called *deadlock,* or, sometimes, *deadly embrace.* We will discuss some deadlock-avoiding philosopher algorithms a bit later.

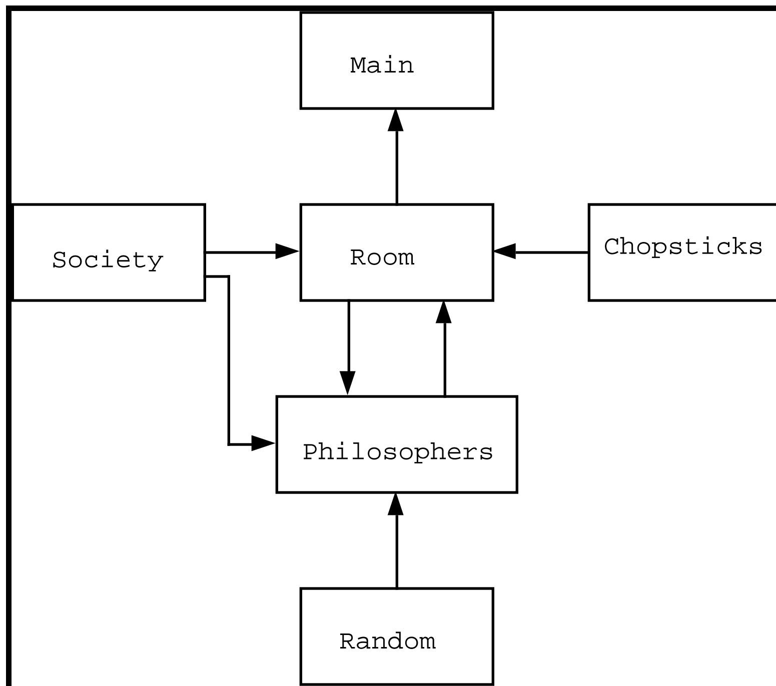
Modeling the Philosophers

Let us model the philosophical society with an Ada program. As with the bank simulation, a small excerpt from the program's execution looks like this:

```
T = 3      Jean Ichbiah      Eating meal 1 for 3 seconds.
T = 6      Jean Ichbiah      Yum-yum (burp)
T = 6    Chris Anderson    Second chopstick 3
T = 6      Jean Ichbiah      Thinking 6 seconds.
T = 6    Chris Anderson    Eating meal 1 for 3 seconds.
T = 9     Tucker Taft       First chopstick 4
T = 9     Tucker Taft       Second chopstick 5
T = 9     Tucker Taft       Eating meal 2 for 10 seconds.
T = 9    Chris Anderson    Yum-yum (burp)
T = 9    Chris Anderson    Thinking 5 seconds.
T = 9  Edsger Dijkstra   Second chopstick 2
T = 9  Edsger Dijkstra   Eating meal 1 for 5 seconds.
```

Figure 15.4 shows the various units of the program and the dependencies among them.

Figure 15.4
Package Dependencies for Dining Philosophers



Program 15.18 is the specification for `Society`, a types package similar to the one in the bank example. As you can see, it just provides a subtype to index the philosophers and a set of philosopher names in string form.

Program 15.18
The Society Package

```
PACKAGE Society IS
------------------------------------------------------------------------
--| Dining Philosophers - Ada 95 edition
--| Society gives unique ID's to people, and registers their names
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  SUBTYPE Unique_DNA_Codes IS Positive RANGE 1..5;

  Name_Register : ARRAY(Unique_DNA_Codes) OF String(1..18) :=
```

```
     ("Edsger Dijkstra   ",
      "Alan Burns        ",
      "Chris Anderson    ",
      "Tucker Taft       ",
      "Jean Ichbiah      ");

END Society;
```

Next, we examine Program 15.19, the main program. As you can see, it consists only of a `New_Line` to flush the output buffer and an entry call,

```
Room.Maitre_D.Start_Serving;
```

which presses the “start button” of the maître d’, or manager, of the dining room. We will see the details of `Room` shortly.

Program 15.19
The Diners Main Program

```
WITH Ada.Text_IO;
WITH Room;
PROCEDURE Diners IS
------------------------------------------------------------------------
--| Dining Philosophers - Ada 95 edition
--| This is the main program, responsible only for telling the
--| Maitre_D to get busy.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

BEGIN -- Diners

  Ada.Text_IO.New_Line;     -- artifice to flush output buffer
  Room.Maitre_D.Start_Serving;

END Diners;
```

Program 15.20 shows the specification of the philosophers package. Each philosopher is a task object with a discriminant to assign its name and a “start button” to cause it to begin its lifetime of eating and thinking. The “start button” parameters `Chopstick1` and `Chopstick2` are used to assign chopsticks to the philosophers; each philosopher will always pick up `Chopstick1` first, then `Chopstick2`. In this way, the task that presses the “start button” can determine each philospher’s eating algorithm.

Philosophers exist only to eat and think, not to communicate with the outside world. However, to allow us to observe their behavior, each philosopher reports his or her current state to the maître d’, who will broadcast a running account of the happenings in the dining room. To make this possible, this package also provides an enumeration type `States`:

```
TYPE States IS (Breathing, Thinking, Eating, Done_Eating,
                Got_One_Stick, Got_Other_Stick, Dying);
```

Program 15.20
Philosopher Specification

```
WITH Society;
PACKAGE Phil IS
------------------------------------------------------------------------
--| Dining Philosophers - Ada 95 edition
--| Philosopher is an Ada 95 task type with discriminant
```

```
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  TASK TYPE Philosopher (My_ID : Society.Unique_DNA_Codes) IS

    ENTRY Start_Eating (Chopstick1 : IN Positive;
                        Chopstick2 : IN Positive);

  END Philosopher;

  TYPE States IS (Breathing, Thinking, Eating, Done_Eating,
                  Got_One_Stick, Got_Other_Stick, Dying);

END Phil;
```

The dining room specification is given in Program 15.21. This is where the set of chopsticks, an array of type `Chop.Stick`, is defined. `Maitre_D` is a task, with a “start button,” as mentioned above, and a second entry, `Report_State`, that is called by the various philosophers.

Program 15.21
Room Specification

```
WITH Chop;
WITH Phil;
WITH Society;
PACKAGE Room IS
------------------------------------------------------------------------
--| Dining Philosophers - Ada 95 edition
--| Room.Maitre_D IS responsible for assigning seats at the
--|   table, "left" and "right" chopsticks, and for reporting
--|   interesting events to the outside world.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  Table_Size : CONSTANT := 5;
  SUBTYPE Table_Type IS Positive RANGE 1 .. Table_Size;

  Sticks : ARRAY (Table_Type) OF Chop.Stick;

  TASK Maitre_D IS
    ENTRY Start_Serving;
    ENTRY Report_State (Which_Phil : IN Society.Unique_DNA_Codes;
                        State      : IN Phil.States;
                        How_Long   : IN Natural := 0;
                        Which_Meal : IN Natural := 0);
  END Maitre_D;

END Room;
```

We have not yet shown a specification for chopsticks; we will come back to this after looking at the philosopher and room package bodies.

Implementing the Philosophers and the Dining Room

The body of the philosopher package is shown in Program 15.22. As in the bank customer case, each philosopher task draws random numbers to simulate its eating and thinking times. The start-button parameters `Chopstick1` and `Chopstick2` are saved in the variables `First_Grab` and `Second_Grab`, respectively. The basic philosopher algorithm is

```
   FOR Meal IN Life_Time LOOP

     Room.Sticks (First_Grab).Pick_Up;
     Room.Sticks (Second_Grab).Pick_Up;

     Meal_Time := Meal_Length.Random(M);
     DELAY Duration (Meal_Time);

     Room.Sticks (First_Grab).Put_Down;
     Room.Sticks (Second_Grab).Put_Down;

     Think_Time := Think_Length.Random(T);
     DELAY Duration (Think_Time);

   END LOOP;
```

The code in Program 15.22 is slightly more elaborate, to allow the philosopher to report its current state to the maître d'.

Program 15.22
Philosopher Body

```
WITH Society;
WITH Room;
WITH Ada.Numerics.Discrete_Random;
PACKAGE BODY Phil IS
------------------------------------------------------------------------
--| Dining Philosophers - Ada 95 edition
--| Philosopher is an Ada 95 task type with discriminant.
--| Chopsticks are assigned by a higher authority, which
--|   can vary the assignments to show different algorithms.
--| Philosopher always grabs First_Grab, then Second_Grab.
--| Philosopher is oblivious to outside world, but needs to
--|   communicate its life-cycle events to the Maitre_D.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  SUBTYPE Think_Times IS Positive RANGE 1..8;
  PACKAGE Think_Length IS
    new Ada.Numerics.Discrete_Random (Result_Subtype => Think_Times);

  SUBTYPE Meal_Times IS Positive RANGE 1..10;
  PACKAGE Meal_Length IS
    new Ada.Numerics.Discrete_Random (Result_Subtype => Meal_Times);

  TASK BODY Philosopher IS  -- My_ID is discriminant

    SUBTYPE Life_Time IS Positive RANGE 1..5;

    Who_Am_I    : Society.Unique_DNA_Codes := My_ID; -- discrim
    First_Grab  : Positive;
    Second_Grab : Positive;
    Meal_Time   : Meal_Times;
    Think_Time  : Think_Times;
    T           : Think_Length.Generator;
    M           : Meal_Length.Generator;

  BEGIN

    Think_Length.Reset(T);
    Meal_Length.Reset(M);
```

```
     -- get assigned the first and second chopsticks here

    ACCEPT Start_Eating (Chopstick1 : IN Positive;
                         Chopstick2 : IN Positive) do
      First_Grab  := Chopstick1;
      Second_Grab := Chopstick2;
    END Start_Eating;

    Room.Maitre_D.Report_State (Who_Am_I, Breathing);

    FOR Meal IN Life_Time LOOP

      Room.Sticks (First_Grab).Pick_Up;
      Room.Maitre_D.Report_State
        (Who_Am_I, Got_One_Stick, First_Grab);

      Room.Sticks (Second_Grab).Pick_Up;
      Room.Maitre_D.Report_State
        (Who_Am_I, Got_Other_Stick, Second_Grab);

      Meal_Time := Meal_Length.Random(M);
      Room.Maitre_D.Report_State (Who_Am_I, Eating, Meal_Time, Meal);

      DELAY Duration (Meal_Time);

      Room.Maitre_D.Report_State (Who_Am_I, Done_Eating);

      Room.Sticks (First_Grab).Put_Down;
      Room.Sticks (Second_Grab).Put_Down;

      Think_Time := Think_Length.Random(T);
      Room.Maitre_D.Report_State (Who_Am_I, Thinking, Think_Time);
      DELAY Duration (Think_Time);

    END LOOP;

    Room.Maitre_D.Report_State (Who_Am_I, Dying);

  END Philosopher;

END Phil;
```

Program 15.23 shows the body of the dining room package. As with the bank tellers, the philosophers are declared as `ALIASED` variables, each with its own discriminant value. The maître d' task has the job of assigning seats and chopsticks to philosophers. Seats are assigned by the statement

```
Phils :=
  (Dijkstra'Access,
   Anderson'Access,
   Taft'Access,
   Ichbiah'Access,
   Burns'Access);
```

Chopsticks are assigned by pressing the “start buttons” as follows:

```
Phils (1).Start_Eating (1, 2);
Phils (3).Start_Eating (3, 4);
Phils (2).Start_Eating (2, 3);
Phils (5).Start_Eating (1, 5);
Phils (4).Start_Eating (4, 5);
```

copyright © 1996, Michael B. Feldman

The peculiar order of starting the tasks is just to ensure that the action starts early; note that philosophers 1 and 3 can begin eating immediately. Philosophers 1, 2, 3, and 4 are told to grab their left chopsticks first, but philosopher 5 (`Burns`) is told to grab its right chopstick first. This ensures that the situation in which all the philosophers hold only their left chopsticks cannot occur. The circularity is broken and there is no deadlock.

Program 15.23
Room Body

```
WITH Ada.Text_IO;
WITH Chop;
WITH Phil;
WITH Society;
WITH Calendar;
PRAGMA Elaborate (Phil);
PACKAGE BODY Room IS
------------------------------------------------------------------------
--| Dining Philosophers, Ada 95 edition
--| A line-oriented version of the Room package
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  -- philosophers sign into dining room, giving Maitre_D their DNA code

  Dijkstra  : ALIASED Phil.Philosopher (My_ID => 1);
  Burns     : ALIASED Phil.Philosopher (My_ID => 2);
  Anderson  : ALIASED Phil.Philosopher (My_ID => 3);
  Ichbiah   : ALIASED Phil.Philosopher (My_ID => 4);
  Taft      : ALIASED Phil.Philosopher (My_ID => 5);

  TYPE Philosopher_Ptr IS ACCESS ALL Phil.Philosopher;
  Phils : ARRAY (Table_Type) OF Philosopher_Ptr;

  TASK BODY Maitre_D IS

    T          : Natural;
    Start_Time : Calendar.Time;
    Blanks : CONSTANT String := "     ";

  BEGIN

    ACCEPT Start_Serving;
    Ada.Text_IO.New_Line;
    Ada.Text_IO.Put_Line
      ("Ada 95 Philosophical Society is Open for Business!");

    Start_Time := Calendar.Clock;

    -- now Maitre_D assigns phils to seats at the table

    Phils :=
      (Dijkstra'Access,
       Anderson'Access,
       Taft'Access,
       Ichbiah'Access,
       Burns'Access);

    -- and assigns them their chopsticks.

    Phils (1).Start_Eating (1, 2);
    Phils (3).Start_Eating (3, 4);
    Phils (2).Start_Eating (2, 3);
```

```
    Phils (5).Start_Eating (1, 5);
    Phils (4).Start_Eating (4, 5);

    LOOP
      SELECT
        ACCEPT Report_State (Which_Phil : IN Society.Unique_DNA_Codes;
                             State      : IN Phil.States;
                             How_Long   : IN Natural := 0;
                             Which_Meal : IN Natural := 0) do

          T := Natural (Calendar."-" (Calendar.Clock, Start_Time));

          CASE State IS

            WHEN Phil.Breathing =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "Breathing");
            WHEN Phil.Thinking =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "Thinking"
                      & Integer'Image (How_Long) & " seconds.");
            WHEN Phil.Eating =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "Eating meal"
                      & Integer'Image (Which_Meal)
                      & " for"
                      & Integer'Image (How_Long) & " seconds.");
            WHEN Phil.Done_Eating =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "Yum-yum (burp)");
            WHEN Phil.Got_One_Stick =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "First chopstick"
                      & Integer'Image (How_Long));
            WHEN Phil.Got_Other_Stick =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "Second chopstick"
                      & Integer'Image (How_Long));
            WHEN Phil.Dying =>
              Ada.Text_IO.Put_Line ("T =" & Integer'Image (T) & " "
                      & Blanks(1..Which_Phil)
                      & Society.Name_Register(Which_Phil)
                      & "Croak");

          END CASE; -- State

        END Report_State;

      OR
        TERMINATE;
      END SELECT;
```

copyright © 1996, Michael B. Feldman

```
    END LOOP;

  END Maitre_D;

END Room;
```

Having assigned seats and chopsticks, bringing the philosophers to life, the maître d' enters its main loop, which is just to wait for a philosopher to report a state, use a CASE statement to determine which state was reported, and display an appropriate message on the screen.

The Chopsticks Package

We have left the chopstick package for last because it introduces some new material about protected types. Program 15.24 shows the specification for the chopsticks package. A chopstick is a protected object with a data structure of its own, declared in its PRIVATE section. The Boolean flag In_Use reflects the fact that a chopstick cannot be picked up while it is in use by another philosopher. Also, the Pick_Up operation is specified as an entry, rather than a protected procedure as in earlier examples.

Program 15.24
Chopstick Specification

```
PACKAGE Chop IS
------------------------------------------------------------------------
--| Dining Philosophers - Ada 95 edition
--| Each chopstick is an Ada 95 protected object
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  PROTECTED TYPE Stick IS
    ENTRY Pick_Up;
    PROCEDURE Put_Down;
  PRIVATE
    In_Use: Boolean := False;
  END Stick;

END Chop;
```

Program 15.25 gives the body of the chopsticks package. The chopstick's protected procedure Put_Down sets the flag In_Use to False. The entry Pick_Up behaves like a protected procedure, but with one important difference: The entry body—which sets In_Use to True—is executed only when it makes sense to do so, namely when the chopstick is not in use. The clause

```
WHEN NOT In_Use
```

is called a *barrier condition,* and serves to ensure that a chopstick can be held by at most one philosopher.

Program 15.25
Chopstick Body

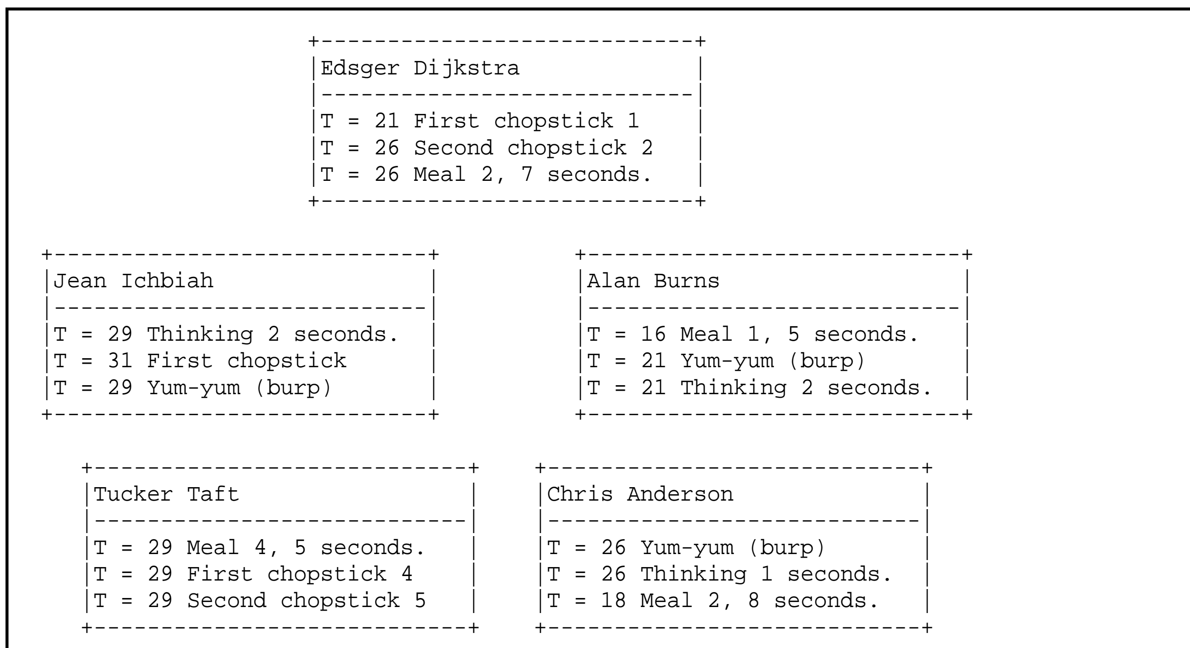
```
PACKAGE BODY Chop IS
------------------------------------------------------------------------
--| Chopstick Body
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996
------------------------------------------------------------------------

  PROTECTED BODY Stick IS
```

```
    ENTRY Pick_Up WHEN NOT In_Use IS
    BEGIN
      In_Use := True;
    END PIck_Up;

    PROCEDURE Put_Down IS
    BEGIN
      In_Use := False;
    END Put_Down;

  END Stick;
```

A More Interesting Philosophers Display

As a final example, consider Figure 15.5. Here each philosopher is given its own window, in which its activity is displayed. Such a screen display would give an animated depiction of the philosophers as shown in Figure 15.3. Because all input/output activity in the philosophical society is carried out by the maître d', this window-oriented display can be achieved by modifying *only* the body of `Room` so that the maître d' task uses operations from the windows package of Programs 2.19 and 2.20. We leave this interesting modification as an exercise.

Figure 15.5
The Dining Room with Windows



As we pointed out at the beginning of this chapter, a single chapter in a book of this kind cannot really do justice to a topic as rich and interesting as concurrent programming. Our intention here has been to introduce you to the subject through some brief examples and two longer simulations. We hope this chapter has stimulated your interest in pursuing concurrency, and Ada's concurrency facilities in particular, through further reading and projects.

Exercises

1. Investigate whether your Ada implementation supports time-slicing and, if it does, whether time-slicing can be turned on and off at will. Experiment with doing so, using Program 15.2 as a test program.

2. Experiment with using different starting orders in Program 15.4. Is there any difference in the behavior?

3. In the bank example, each teller has four entries and, therefore, four distinct queues. This is not very realistic; in a real bank, a teller has only one queue and processes whichever transaction is at the head of that queue. Modify the tellers package, and other units as necessary, so that each teller task has a single entry with an additional parameter to indicate the nature of the transaction. Use an enumeration type to represent the transaction types.

4. In the dining philosophers example, modify the body of the `Room` package (Program 15.23) so that the maître d' task uses operations from the `Windows` package (Programs 2.19 and 2.20) to display each philosopher's activity in a window as shown in Figure 15.5. *Hint:* The maître d' should open all the windows before bringing the philosophers to life.

copyright © 1996, Michael B. Feldman