1 Definition, Creation, Start, and Priorities of Actors

Introduction

This chapter describes, in detail, the concepts and approaches which allow a user to define, create, and start an actor in Ada, CHILL, and Java. Priorities are also mentioned. Examples illustrating these ideas will be given.

1.1 Definition of Actor Types

1.1.1 Definition of Actor Types in Ada

"The execution of an Ada program consists of the execution of one or more tasks. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks." (ARM, 1995, [9(1)])

These words introduce the section of the Ada Reference Manual (henceforth called simply ARM) that describes Ada's features of concurrent execution. In particular, the word *"task"* is mentioned and we will take the opportunity to explain the nature of a task. Roughly speaking, a task is a part of a program that has its own thread of control and hence can be executed concurrently. An Ada program can therefore have associated with it several threads of control. As far as Ada is concerned, a task unit is a composed structure that is described by a task declaration: a user can either define a named task type or a single anonymous task object. The following rules illustrate this:

```
task_type_declaration ::=
   TASK TYPE defining_identifier
      [known_discriminant_part] [IS task_definition];

single_task_declaration ::=
   TASK defining_identifier [IS task_definition];

task_definition ::=
   {task_item}
[ PRIVATE
   {task_item}]
END [task_identifier]

task_item ::= entry_declaration | representation_clause
```

A task declaration, then, requires a completion, i.e., a task body. Every task declaration shall have one body and every task body shall be the completion of a task declaration:

```
task_body ::=
   TASK BODY defining_identifier IS
      declarative_part
   BEGIN
      handled_sequence_of_statements
   END [task_identifier];
```

Readers familiar with the language will realize that the concept of splitting up a task into a specification and a corresponding body is a frequently encountered technique in Ada—almost every program entity can be decomposed in such a way: subprograms, packages, protected objects (to be introduced later), and tasks. The benefits of such an approach are well known in software engineering; the specification contains the interface to the entity which provides the client with just the information that are required for use. The visible entries define those parts of a task that can be accessed by other entities; they indicate how other tasks can communicate directly with the task. Entries that are not to be exported are declared in the private part of the specification, making them only accessible internally. The (probably involved) details of the implementation of the bodies of the entries can be conveniently hidden (in the body) from the external user. This prevents direct manipulation of crucial data (the interface is the sole access to the entity) and enables the programmer to alter the actual implementation without affecting the client (provided that the new algorithm is semantically equivalent, however).

1.1.2 Definition of Actor Types in CHILL

"CHILL allows for the concurrent execution of program units. A thread (process or task) is the unit of concurrent execution." Processes are then considered to be executed concurrently with the starting thread. "CHILL allows for one or more processes with the same or different definition to be active at one time." (Z200, 1996, [1.11])

"A thread is either a process or a task. A process is the sequential execution of a series of statements. It may be executed concurrently with other threads. The behaviour of a process is described by a process definition that describes the objects local to a process and the series of action statements to be executed sequentially.

A task is a sequential execution of a series of statements. It may be executed concurrently with other threads. The behaviour of a task is described by a task mode definition." (Z200, 1996, [11.1])

This is what the CHILL Reference Manual tells us and we will now look a bit closer to process and task definitions. We begin with processes:

```
<process definition statement> ::=
    <defining occurrence> : <process definition>
    [<handler>] [<simple name string>]; |
    <generic process instantiation>;

<process definition> ::=
    PROCESS([<formal parameter list>]) <process body> END
```

Such a definition statement declares a sequence of executable statements, the <process body>, that may be started for concurrent execution from different places in the program. There is an important static semantic property that forbids a process to be included in a region (to be introduced in Chapter 5) or a block other than the imaginary outermost process. That is, processes cannot be nested. We continue with some examples (Z200, 1996, [D14, D32]):

```
Call_Distributer: PROCESS();
   Wait:
   PROC(X INT);
     /* some wait action */
   END Wait;
   DO FOR EVER;
     Wait(10 /* seconds */);
     CONTINUE Operator_Is_Ready;
   OD;
END Call_Distributer;
Producer: PROCESS();
```

```
    DCL Elem1  RegionStackWithTopMode1!ElementMode;
    DO FOR EVER;
     /* compute Elem1 */
      Stack1.Push(Elem1);
    OD;
END Producer;
```

Example 1.1: process definitions in CHILL

For tasks, a task mode definition can be used:

```
<task mode> ::=
     <task mode specification> |
     <task mode body>

<task mode specification> ::=
     TASK SPEC [ABSTRACT] [<task inheritance>
     {<task specification component>}[<invariant part>]
     END [<simple name string>]

<task mode body> ::=
     TASK BODY [ABSTRACT] [<task inheritance>
     {<task body component>}[<invariant part>]
     END [<handler>] [<simple name string>]

<task specification component> ::=
     <region specification component>

<task body component> ::=
     <region body component>
```

A region specification component can be: a declaration, a module component, a procedure specification (header), a signal definition (to be introduced later), or a visibility extension statement. A region body component can be a module component and/or the implementation of the procedures from the specification. As can be seen from the production rules, the internal parts of regions and tasks coincide. We will return to this issue in the subsequent chapters. Let us continue with an example:

```
SYNMODE StackMode1 = TASK SPEC
   GRANT ElementMode, Push, Pop;
   NEWMODE  ElementMode  =  STRUCT (a  INT, b  BOOL);
   Push: PROC(Elem  ElementMode IN) EXCEPTIONS(Overflow) END Push;
   Pop: PROC() RETURNS(ElementMode) EXCEPTIONS(Underflow) END Pop;
   SYN Length = 10_000;
   DCL StackData ARRAY (1:Length) ElementMode;
   DCL TopOfStack RANGE(0:Length) INIT := 0;
END StackMode1;

SYNMODE StackMode1 = TASK BODY
   Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow)
     IF TopOfStack = Length THEN
       CAUSE Overflow;
     ELSE
       TopOfStack +:= 1;
       StackData(TopOfStack) := Elem;
     FI;
   END Push;
   Pop: PROC() RETURNS(ElementMode) EXCEPTIONS(Underflow)
```

```
    IF TopOfStack = 0 THEN
      CAUSE  Underflow;
    ELSE
      RESULT(StackData(TopOfStack));
      TopOfStack -:=  1;
    FI;
  END Pop;
END StackMode1;
```

Example 1.2: Abstract Data Type Stack in CHILL, see (Z200, 1996, [D30])

1.1.3 Definition of Actor Types in Java

The concept for a concurrent entity in Java is the thread: "A thread is a single sequential flow of control. Thread objects allow multi-threaded Java programming; a single Java Virtual Machine can execute many threads in an interleaved or concurrent manner." (JLS, 1996, [20.20])

Java, being a class oriented language, provides intrinsic support for threading by means of the class `Thread` which can be used as a building block for user defined actors types. Any section of Java code that is to be executed within its own thread of control has to be associated with the class `Thread`.

There are two ways a user can define a new thread: one approach is to let one's class extend the class `Thread`—that is, to use Java's inheritance mechanisms to create a subclass of `Thread`. This subclass, then, should override `Thread`'s `run()` method, which initially contains only code that is not appropriate for this approach. A non-exhaustive presentation of the class `Thread` is given below (see (JLS, 1996, [20.20]) for further details):

```
public class Thread implements Runnable {
   public final static int MIN_PRIORITY = 1;
   public final static int MAX_PRIORITY = 10;
   public final static int NORM_PRIORITY = 5;
   public Thread();
   public Thread(String name);
   public Thread(Runnable runObject);
   ...
   public void run();
   public void start() throws IllegalThreadStateException;
   public final void stop() throws SecurityException;
   ...
   public final void suspend() throws SecurityException;
   public final void resume() throws SecurityException;
   ...
   public final int getPriority();
   public final void setPriority(int newPriority)
     throws SecurityException, IllegalArgumentException;
   ...
   public static void sleep(long millis) throws InterruptedException;
   ...
   public void destroy();
}
```

Example 1.3: The essential parts of the class `Thread`

The work that is to be performed by the user thread can be accommodated in the `run()` method. Thus, the class `Thread` acts merely as a framework for user threads; new components (data and methods) can be added as required by the application.

```
class PrimeThread extends Thread {
   long minPrime;
   PrimeThread(long minPrime) {
     this.minPrime = minPrime;
   }
   public void run() {
     // compute primes larger than minPrime
     ...
   }
}
```

Example 1.4: Actor type definition by extending the class `Thread`

In this example, we have derived `PrimeThread` from the class `Thread` by adding a new data item and a constructor. We have also overridden `Thread`'s `run()` method to place useful work inside it.

Alternatively, a class can be declared to implement the `Runnable` interface, which requires this class to override the interface's sole method—`run()`:

```
public interface Runnable {
   public abstract void run();
}
```

Our prime example (JLS, 1996, D[20.20]) in this new style then reads:

```
class PrimeRun implements Runnable {
   long minPrime;
   PrimeRun(long minPrime) {
     this.minPrime = minPrime;
   }
   public void run() {
     // compute primes larger than minPrime
     ...
   }
}
```

Example 1.5: Implementing the `Runnable` interface

As mentioned above, this is only the first step. We still need to associate `PrimeRun` with the class `Thread`. See Section 1.2.3 for details and a pictorial illustration of the two approaches.

1.2 Definition and Creation of Actors

1.2.1 Definition and Creation of Actors in Ada

Once a task type's definition has been successfully elaborated, it can be used to create task objects explicitly in a way similar to the creation of objects of other types, say `Integer`. For a single task declaration, the elaboration implicitly creates an object of an anonymous task type.

Task types are limited types—hence, neither assignment nor comparison are available[1], but this is the only restriction. It is possible to define an access type to a task type and to create task objects dynamically, using the `new` operator. We now give some examples of task type declaration and task object creation (ARM, 1995, [9.1(22)]):

[1] it would be difficult to define an appropriate semantics

```
task type Server is
   entry Next_Work_Item(WI : in Work_Item);
   entry Shut_Down;
end Server;

task body Server is
begin
   Work_Loop: loop
     select
       accept Next_Work_Item(WI : in Work_Item) do
         -- do what has do be done
       end Next_Work_Item;
     or
       accept Shut_Down do
         -- bring the system down
         exit Work_Loop;
       end Shut_Down;
     or
       terminate;
     end select;
   end loop;
end Server;

task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;

task Controller is
   entry Request(Level)(D : Item);  --  a family of entries
end Controller;

task User;  --  has no entries
```

Example 1.6: Task type definitions in Ada

The latter two declarations are examples of single task declaration. Being a type, the first two declarations can be used to create task objects in a straightforward manner:

```
paxp02, pdec01, pax10f, ipc405 : Server;
Teletype : Keyboard_Driver(TTY);
Secure_Consoles : array(1 .. 100) of Keyboard_Driver;
```

By declaring an access type, we can create tasks dynamically:

```
type Keyboard is access Keyboard_Driver;
Terminal : Keyboard := new Keyboard_Driver(VT100);
```

1.2.2 Definition and Creation of Actors in CHILL

Again, let's consider processes first: "A process is created by the evaluation of a start expression… An unspecified number of processes with the same definition may be created and may be executed concurrently… The creation of a process causes the creation of its locally declared locations, except those declared with the attribute `STATIC`, and of locally defined values and procedures. The locally declared locations, values and procedures are said to have the same activation as the created process to which they belong. The imaginary outermost process, which is the whole CHILL program under execution, is considered to be created by a start expression executed by the system under whose control the program is executing. At the creation of a process, its formal parameters, if present, denote the values and locations as delivered by the corresponding actual parameters in the start expression." (Z200, 1996, [11.1])

A start expression is as follows:

```
<start expression> :=
    START <process name> ([<actual parameter list>])
```

Besides creating and starting an instance of the process definition, the start expression delivers a so-called `INSTANCE` value that identifies the created process. Thus, a location of mode `INSTANCE` can be declared and assigned the result of the start expression, as in:

```
DCL PID INSTANCE;
PID := START Call_Distributer;
START Producer;
```

The first creation causes an instance of `Call_Distributer` to be bound to `PID` so that it can be manipulated in the sequel, whereas throwing the `INSTANCE` value away results in this instance of `Producer` being no longer reachable. Although a process definition is the basis of a process instance creation, a process definition cannot be regarded as a mode definition in general. Elaborating the `START` expression creates a process instance, but it is not possible for a user to directly declare those instances, i.e., `DCL My_Call_Distributer Call_Distributer` is not allowed. Instead, an instance of the process definition is created implicitly in the context of the elaboration of the `START` expression. We will, thus, regard a process definition as an actor type definition (on the program text level only, of course) that has the above mentioned limitations.

To overcome these obstacles, for example, to define actor types/modes as "true" types from which instances can be derived by the user or which can be used as formal parameters in subprogram specifications, CHILL has the concept of a task mode, the definition of which we have already encountered. "A task is created as part of the creation and initialisation of a task mode location. It is called to belong to this task mode location." (Z200, 1996, [11.1])

```
DCL paxp02, pdec01 Server; /* paxp02 and pdec01 are ``up'' */
```

Likewise, a task mode can be used as a formal parameter in subprogram parameter profiles or to create task locations dynamically. For the latter, the approach is straightforward: first, a reference mode to a task mode has to be defined which can then be used to allocate space for the task locations dynamically, using the intrinsic CHILL routines `ALLOCATE` or `GETSTACK`:

```
SYNMODE Parser_Ptr = REF Parser;
DCL GNAT_Parser Parser_Ptr;
GNAT_Ptr := ALLOCATE(Parser); /* ready to perform lexical analysis */
```

1.2.3 Definition and Creation of Actors in Java

Remember that there are two approaches of defining actor types in Java. As a result, we have a class. In the first case, it is a descendant of `Thread` and hence can be used directly to create a `Thread` object, using Java's features of instance creation. So we can write:

```
PrimeThread P; // to set up a reference variable and
P = new PrimeThread(143); // to create a new object
```

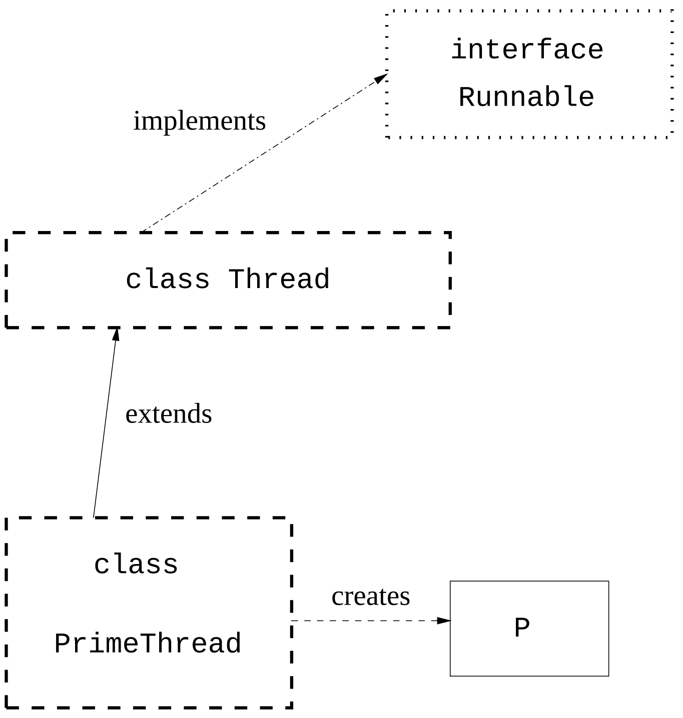

Figure 1.1: The first approach of actor type definition in Java

The JVM must now create a separate thread of control for `P` in which `P` can be executed. See Figure 1.1 for an illustration of this first approach, deriving directly from `Thread`.

For the second approach to define an actor type, remember that we have, as a first step, already implemented the interface `Runnable` (in the class `PrimeRun`), but there remains work to do in order to create a complete thread based on this class. As mentioned above, the JVM cannot put code into a separate thread of control unless this code happens to be associated with `Thread`: "Creating a `Thread` object creates a thread and that is the only way to create a thread." (JLS, 1996, [17.12]) To achieve this binding, an instance of our class can be used to create a new `Thread` object by means of one of `Thread`'s specialized constructors—namely, `Thread(Runnable runObject)`. This constructor initializes a newly created `Thread` object so that it has the given `runObject` as its separate run object. The `run()` method of class `Thread` simply calls this run object's `run()` method, i.e., the method we have implemented in `PrimeRun`.

It is now straightforward to realize these thoughts:

```
PrimeRun P = new PrimeRun(143); // an instance of our class
Thread PrimeRunThread = new Thread(P); // a new Thread object
```

Figure 1.2 shows the second approach of defining actor types and declaring actors.

At a first glance, both approaches of defining and creating threads seem quite different, but careful study reveals similarities. For the definition of threads, it can be noted that the class `Thread` itself implements the `Runnable` interface. So if we are inheriting from `Thread`, our subclass also implements this interface. And, lastly, thread creation always requires `Thread` object creation, either by instantiating a subclass of `Thread` or by directly creating `Thread` objects that take a runnable object as a parameter. Hence, the invariant for separate threads of control (association with `Thread`, see Figure 1.1 and Figure 1.2) is always maintained.

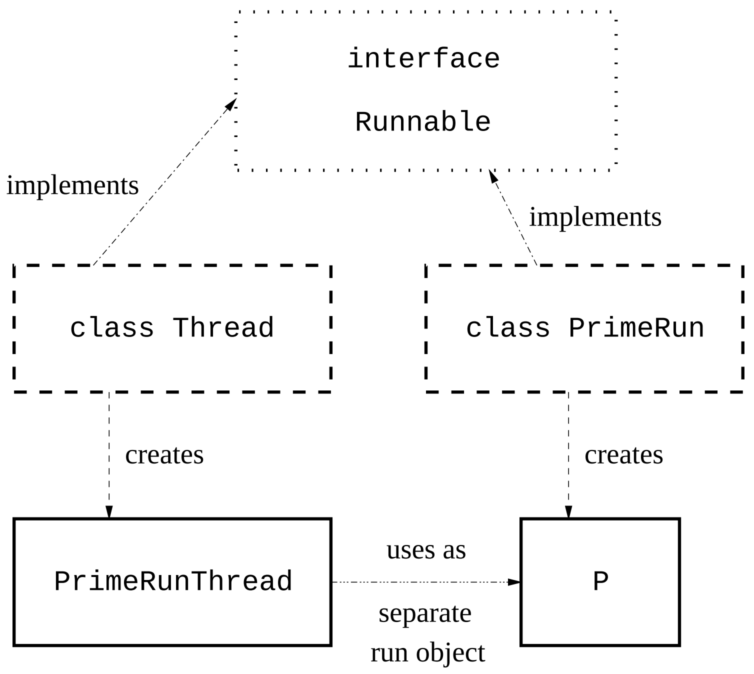

Figure 1.2: The second approach of actor type definition in Java

1.3 Start of Actors

1.3.1 Start of Actors in Ada

"The execution of a task of a given task type consists of the execution of the corresponding task body. The initial part of this execution is called the *activation* of the task; it consists of the elaboration of the declarative part of the task body. Should an exception be propagated by the elaboration of its declarative part, the activation of the task is defined to have failed, and it becomes a completed task." (ARM, 1995, [9.2(1)])

We have seen so far that task objects can be created in three ways: explicitly via object declaration, implicitly by means of single task declaration, and dynamically, using access types and the `new` allocator. Let's consider the former two first: Once the creation has been successfully done, the task objects exist as entities. These tasks, however, are not active yet. Tasks in Ada are started automatically. For a given declarative region, R, all task of R that are subject to explicit object declaration (including single tasks) are activated together. This event takes place within the context of the handled sequence of statements of R, just prior to executing these statements. Less formally, such tasks become activated when the Ada Machine reaches the `begin` of the block in which they are declared.

Tasks that are created dynamically are activated whilst their allocators are being evaluated, after any per-object initializations, but before the access value is returned. Their activation is thus not deferred until the ensuing `begin`—they are activated immediately after creation.

The task that is responsible for the creation (in either of the three above mentioned ways) is known as *"activator"*. This can actually be another task or, in case the new tasks are declared in the main procedure or one of main's subprocedures, the so-called hypothetical main task[2]. While such activations are in progress, the activator is suspended. If all activations complete successfully, the activator carries on normally with its execution; otherwise, if any activation fails, `Tasking_Error` is raised in the activator (**not in the new task!**) and the task that failed becomes terminated and is never activated at all. This

[2] tasks that were declared in library packages (library tasks) are started in the context of the elaboration (`with`ing) of the corresponding library packages

should be contrasted with the case where an exception is raised in a declarative part of an entity other than a task body; here, **all** tasks that were created during that elaboration become terminated. The reason why activators must be suspended whilst their children are being activated is that any problems with the creation and activation of the children must be reported to the parent. The inclined reader is referred to (Burns, 1998, [Sec. 4.3]) for a wonderful introduction to task hierarchies and dependence during creation, activation, and termination.

The following example is taken from (Barnes, 1996, [p. 421]); it illustrates the behaviour of a block containing the declarations of two tasks, `A` and `B` (Figure 1.3 illustrates the time-related behaviour):

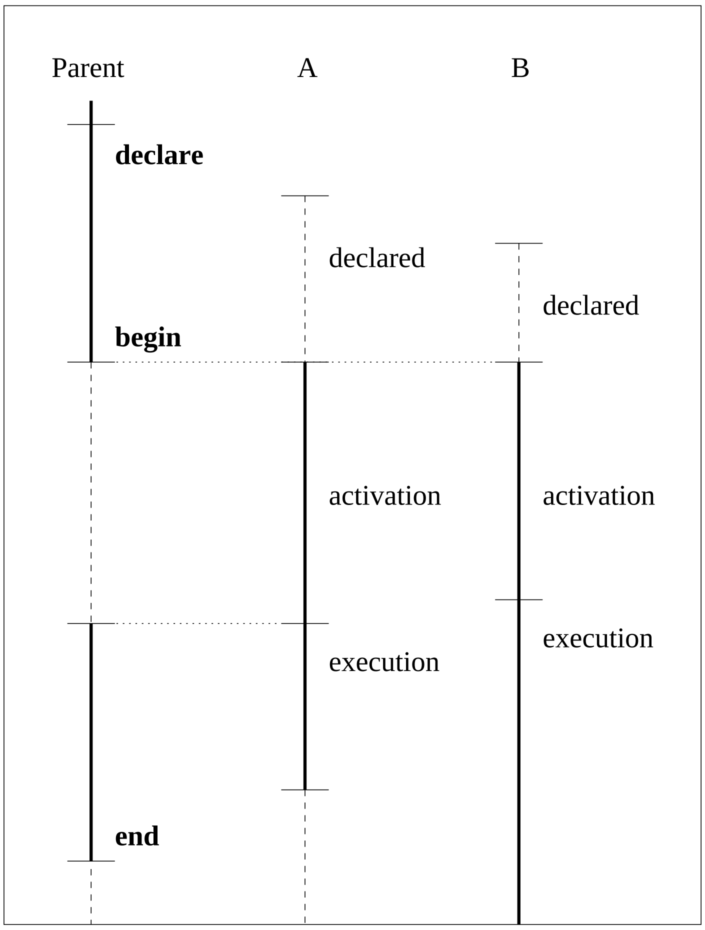

Figure 1.3: Ada task activation

```
-- executed by Parent
declare
   ...
   A : T;
   B : T;
   ...
begin
   ...
end;
```

For the sake of illustration, we show task `A` finishing its activation after task `B` so that the parent resumes execution when task `A` enters its execution stage. We also show task `A` finishing its execution and therefore becoming completed and terminated before task `B`. The parent is shown reaching its `end` and therefore completing execution of the block after task `A` has terminated but before task `B` has terminated. The parent is therefore suspended until task `B` is terminated when it can then resume execution with the statements following the block. (For more information on task termination and completion, see Chapter 2).

Once a task has been successfully activated, the execution of the sequence of statements of its task body commences—concurrently with all other tasks.

1.3.2 Start of Actors in CHILL

A CHILL task is started as part of the creation and initialization of a task mode location. A dynamically created task is started as soon as the `ALLOCATE` or `GETSTACK` routine returns successfully.

A process is started by the evaluation of a `START` expression. "It becomes active (i.e., under execution) and is considered to be executed concurrently with other threads. The created process is an activation of the definition indicated by the process name of the process definition. An unspecified number of processes with the same definition may be created and may be executed concurrently... (Note that each evaluation of the `START` expression yields a new instance of the corresponding process definition.) Each process is uniquely identified by an `INSTANCE` value, yielded as the result of the `START` expression or the evaluation of the `THIS` operator." (Z200, 1996, [11.1]) Since creation and start of actors are interwoven in CHILL, the reader is referred to Section 1.2.2 for examples.

1.3.3 Start of Actors in Java

As a result of a `Thread` object creation, we have a `Thread` object, but the thread it represents is not active yet (this holds for both ways of defining and creating threads). A thread, `A`, is activated when another thread, `B`, calls the `start()` method of the `Thread` object that represents `A`.

```
public void start() throws IllegalThreadStateException
```

"Invoking this method causes this thread to begin execution; this thread calls the `run()` method of this `Thread` object. The result is that two threads are running concurrently: the current thread (which returns from the call to the `start()` method) and the thread represented by this `Thread` object (which executes its `run()` method)." (JLS, 1996, [20.20.14])

Returning to our framework, we conclude that `A` then invokes the `Thread` object's `run()` method.

```
public void run()
```

"The general contract of this method is that it should perform the intended action of the thread. The `run()` method of class `Thread` simply calls the `run()` method of the separate run object, if there is one; otherwise, it does nothing." (JLS, 1996, [20.20.13])

If we used the first approach to create a thread, calling `run()` simply executes the (`Thread`'s overridden) `run()` method of the subclass (there is no separate run object), whereas providing a separate run object (the second approach), causes `Thread`'s `run()` to invoke the `run()` method of the runnable object—that is, the class that implemented the interface `Runnable` in the first place.

After we have introduced all the necessary details concerning actor types and actors, let us now return to our theoretical framework presented in Chapter 0: Table 1.1 below lists the summary of what we have worked out so far. For each language, the interpretation of actors on the three levels—static program structure, dynamic program structure, and execution—is given.

Concept	*Static program structure*	*Dynamic program structure*	*Execution*
Neutral	actor def	actor object	seq of actions
Ada task objects	object dcl	task object	seq of actions
CHILL process instances	`START` expr	process instance	seq of actions
CHILL task objects	loc decl	task object	seq of actions
Java thread objects	object def	thread object	seq of actions

Table 1.1: Actor types and Actors as they appear in our three languages

1.4 Priorities

1.4.1 Priorities in Ada

Each task can be given a priority: "A task priority is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The base priority of a task is the priority with which it was created, or to which it was later set by `Dynamic_Priorities.Set_Priority`. At all times, a task also has an active priority, which generally reflects its base priority as well as any priority it inherits from other sources. The task's active priority is used when the task competes for processors. Similarly, the task's active priority is used to determine the task's position in any queue when `Priority_Queuing` is specified. Priority inheritance is the process by which the priority of a task or other entity is used in the evaluation of another task's active priority." (ARM, 1995, [D1(15)])

Ada's Task Dispatching Model (to be introduced in Chapter 6) guarantees that the task with the highest priority is the winner in the competition for resources. The package `System` defines the type `Any_Priority` to be a subtype of `Integer` with an implementation defined range (30 values at least shall be included). The type `Priority`, which is the reference for all priority issues, is declared to be a subtype of `Any_Priority`.

There are two ways for specifying task priorities. The first—using `pragma Priority`—is known as static setting. The pragma is allowed to occur immediately within a task definition and contains an expression that represents the desired priority. For example,

```
task type T(Desired_Priority : Priority := Default_Priority) is
   pragma Priority(Desired_Priority);
   ...
end T;
```

results in objects of type `T` being assigned `Desired_Priority` as their base priority:

```
T1 : T(0); -- an urgent task
Lazy : T(Priority'Last);
Moderate : T; -- same as Moderate : T(Default_Priority);
```

For the dynamic setting of priorities, Ada provides the package `Dynamic_Priorities` in which the procedure `Set_Priority` resides. The latter can be used to alter a task's priority after it is in existence. Tasks that don't come with their own `pragma Priority` inherit the base priority from their respective activators. In the absence of such a pragma for the main procedure, the environmental task's priority is set to `Default_Priority`.

> Task priorities are dealt with in the Annex D, "Real-Time Systems", of the Ada Reference Manual. An implementation need not conform to this annex, however. The core languages itself says nothing about priorities, so priorities may not be available to everyone. This is somewhat awkward since in Ada 83, priorities were included in the core language (Ada 83 did not have any annexes that might have been ignored by an implementation). On the other hand, the authors are firmly convinced that whenever a compiler vendor is able to produce an Ada 95 compiler that complies to the core languages, the extra effort of conforming to Annex D is only little.

1.4.2 Priorities in CHILL

The CHILL manual says little about priorities—they are only mentioned in the context of choosing a thread from a set of delayed processes or tasks. In particular, the author was not able to find information pertinent to a general association of priorities and threads.

For the special case of delaying a running thread (see Chapter 2), a priority can be specified that is later used to decide which thread is to be reactivated.

1.4.3 Priorities in Java

"Every thread has a priority. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to implement mutual exclusion. When code running in some thread creates a new `Thread` object, the newly created thread has its priority initially set equal to the priority of the creating thread. But the priority of a thread `T` may be changed at any time if some thread invokes the `setPriority()` method of the `Thread` object that represents `T` (and the security manager approves execution of the `setPriority()` operation)."

```
public final static int MIN_PRIORITY = 1;
```

The constant value of this field is 1, the smallest allowed priority for a thread.

```
public final static int MAX_PRIORITY = 10;
```

The constant value of this field is 10, the largest allowed priority for a thread.

```
public final static int NORM_PRIORITY = 5;
```

The constant value of this field is 5, the normal priority for a thread that is not a daemon. (JLS, 1996, [20.20.1–3])

Summary and Comparison

Having outlined all the details of declaring, creating, and starting of actor types and actors in the three languages, we will now—as promised in the introduction to this chapter—compare these issues. We stick to the order implied by the whole chapter: definition, creation, and start. Priorities will be touched only briefly; we shall discuss them in length (and compare the ways the languages tackle priority matters) in Chapter 6, when we address Real-Time aspects.

The reader should be warned, however, since it is the inherent nature of a comparison to be subjective and interspersed by the author's opinion. I will, to the best of my abilities, try to keep to the criteria stated above.

Definition

Actor (type) definition—because it is the first stage of all concurrent programming—is of crucial importance. All three languages cater for this by providing detailed means for the definition of actor types and actors. Albeit the names are different—*task*, *process* or *task*, and *thread*, respectively—it is possible to map a concurrent application's requirements onto the structures provided. Data and "work to be performed" can be directly associated with variables and entries, locations and statements/procedures, and fields and methods (especially, `run()`), respectively. This should not seem a surprise—one should bear in mind that the three languages were designed with the intention of supporting the development of concurrent systems. One requirement of such programming languages is clearly the demand that they should facilitate modelling of real world issues within the languages. Perhaps a little flaw, Java fails to support directly the distinction between specifications and implementations with respect to methods and the actor itself[3].

If we consider the possibility to define single actors directly—that is, without a type from which they can be derived—, we remember that Ada allows for such declarations—a task can be declared as a single task. In CHILL, a process definition is special actor type definition: an instance cannot be declared (by the user in the natural way) but is implicitly provided when the `START` expression is evaluated. A CHILL task can only be derived from a task mode. In either of Java's approaches, classes are involved. Single actors are convenient if an application requires exactly one incarnation of a certain actor—for example, to prevent multiple starts of this actor. Ada achieves this by a single task; in CHILL and Java, we would use a technique known as *Singleton Design Pattern*. A task type and, respectively, a class is equipped with a constructor that takes care of this. A static counter inside the task type and the class, respectively, could be used to ensure that exactly one instance is created.

On the other hand, all three languages permit actors types to be defined so that they can be used as components in subsequent structures.

If versatility were a criterion, Ada would be the winner here.

Creation

Ada single tasks and CHILL processes are implicitly created while elaborating the definition and evaluating the `START` expression, respectively. Note that an Ada single task declaration results in exactly one incarnation being created, whereas each evaluation of the `START` expression creates a new instance that is independent of all other instances (of the same or of a different process definition).

CHILL tasks are created in the context of the creation and initialization of a task mode location. Java thread incarnation requires the instantiation of a subclass of `Thread` and named Ada tasks come into existence by explicitly declaring objects of the respective task type. It can be summarized that the approaches to explicit actor creation bear resemblance in the three languages.

Ada, CHILL, and Java support dynamic creation of actors by means of access or reference types. Strictly speaking, Java has only this possibility, since classes are reference types by definition.

[3] interfaces, however, can be used to overcome this obstacle

Start

There are two extremes: Ada tasks are always started automatically by the scope rules of the language model (it is, therefore, not possible to start a task twice). Whenever the Ada Machine reaches the `begin` of a declarative region, *R*, all static tasks of *R* (if any) are (activated and then) started. Dynamically created tasks are started as part of the evaluation of their allocators. Java, on the other hand, gives the programmer free control over when a thread is to be started. In particular, a thread may be defined and created, but actual start is subject to the whim of the programmer only. Although the call to `start()` is of asynchronous nature—once the call has been issued, the caller can proceed independently—, a given Java thread instance cannot be started several times. Any attempt to start a thread while it is still active results in the exception `IllegalThreadStateException` being raised.

CHILL, with processes and tasks, lies somewhat in between. A process has to be started explicitly (but cannot be started several times, since each time a `START` expression is evaluated, a new distinct instance is created); whereas tasks are started when corresponding objects are declared (again, they cannot be started several times). It should be noted, however, that tasks cannot be substituted for processes and vice versa[4].

Priorities

Ada and Java support priorities and the operations one would expect: priorities can be set, changed, and inherited. CHILL seems to exclude direct manipulation of priorities. It is neither clear how they can be associated with processes or tasks nor how they can be altered.

A discussion of the impacts of priorities on scheduling and dispatching is inappropriate here and therefore deferred to Chapter 6.

[4] the reason will be given in Chapter 3