Ada In Practice

Patrick Rogers

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

Ada In Practice

Release 2025-12

Patrick Rogers

Dec 27, 2025

CONTENTS:

This course explores how to use the Ada language in real-world scenarios that extend beyond the complexity of lab exercises. The resulting solutions to these scenarios are idioms and techniques, some common to various programming languages and some specific to Ada. Multiple preliminary solutions to a given scenario (when they exist) are presented and analyzed for strengths and weaknesses, leading to the final preferred solution and its analysis. As a result, readers — especially those relatively new to the language — can learn how experienced Ada developers apply the language in actual practice. Prior knowledge of Ada is required, although explanations of the underlying semantics are provided when appropriate.

This document was written by Patrick Rogers and reviewed by Gustavo A. Hoffmann, Steven Baird, Richard Kenner, Robert A. Duff, and Tucker Taft.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website[2]. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

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

you will find it in this directory:

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

In order to use this code example, just follow these steps:

1. Unpack the zip file;

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

2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

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

CHAPTER

ONE

INTRODUCTION

This course describes how to implement selected programming idioms in the Ada language.

What is an idiom? Some would say that an idiom is a workaround for an expressive deficiency in a programming language. That is not what we mean.

What we have in mind are answers to the question "In this situation, what is the most elegant implementation approach?". Elegant software is comprehensible, efficient, concise, reliable, and, as a result, maintainable, so elegance is an economically and technically desirable characteristic.

Design patterns[12] are intended to answer that question, and indeed some would equate idioms with design patterns. But what we have in mind is more general in scope.

For example, Reference Counting[3] is a well-known approach to tracking and managing the storage for objects and is conceptually independent of the programming language. However, reference counting is not a design pattern.

Likewise, Resource Acquisition Is Allocation (RAII)[4], type punning[5], interface inheritance[6], and implementation inheritance[7] are not design patterns.

Those are the kinds of situations and implementations we focus upon.

That said, we may refer to a design pattern to illustrate an idiom's purpose and/or implementation. For example, in the idiom for controlling object creation and initialization, the implementation approach happens to be the same as for expressing a Singleton[12].

In addition to language-independent situations, we also include implementations for situations specific to the Ada language. These idioms are *best practices* in situations that arise given the extensive capabilities of the language.

For example, Ada directly supports tasks (threads) via a dedicated construct consisting of local objects and a sequence of statements. Tasks can also be defined as types, and then used to define components for other composite types. As a result, there is an idiom showing how to associate a task type with an enclosing composite type so that the task components have visibility to the enclosing object's other components.

In all the idioms we want to apply the fundamental principles of software engineering, especially those of abstraction and information hiding. Therefore, we include an idiom for expressing abstractions as types, with compile-time visibility control over the representation. These are the well-known Abstract Data Types, something the Ada language directly supports but using *building blocks* instead of a single construct. For that same reason we include another idiom for defining abstractions that manage global data (Abstract Data Machines). Most of the idioms' implementations will be defined using these abstraction techniques as their starting point.

[12] E. Gamma, R. Helm, and others. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley Publishing Company, 1995.

[3] https://en.wikipedia.org/wiki/Reference_counting

[4] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

[5] https://en.wikipedia.org/wiki/Type_punning

[6] https://en.wikipedia.org/wiki/Subtyping

[7] https://en.wikipedia.org/wiki/Implementation_inheritance

1.1 Assumptions

We assume the reader knows Ada to some degree, including some advanced topics. For those lacking significant familiarity, we hope these implementations will at least give a sense for how to apply the language. We direct such readers to the online Learn courses dedicated to the Ada language itself[8].

1.2 Definitions

For the sake of avoiding duplication in the idiom entries, the following terms are defined here. Note that the Ada Language Manual includes a glossary in Section 1.3[9] (located in Annex N prior to Ada 2022). Some of the following expand on the definitions found there.

1.2.1 Suppliers and Clients

Suppliers are software units that provide programming entities to other software units, the users. These users are the *clients* of the supplied units. The concept is simple and intuitive, but by defining these terms we can convey these roles quickly in the idioms' discussions.

For example, a unit that defines a type and associated operations would be a supplier. Client units could use that type to declare objects, and/or apply the operations to such objects. The language-defined package `Ada.Text_IO` is an example of a supplier. Similarly, the unit that defines a library, such as a math library, is a supplier. Callers to the math library routines are the clients. The generic package `Ada.Numerics.Generic_Complex_Elementary_Functions`, once instantiated, would be an example supplier. (Arguably, the generic package itself would be a supplier to the client that instantiates it, but instantiation is the only possibility in that narrow case. Only the routines in the instances can be called.)

Bertrand Meyer's book on OOP[13] limits these terms specifically to the case of a type used in an object declaration. Our definitions cover that case but others as well.

Units can be both suppliers and clients, because a given supplier's facility, i.e., the interface and/or implementation, may be built upon the facilities defined by other suppliers.

1.2.2 Compile-time Visibility

In the definitions of supplier and client above, we gave an example in which a supplier's type was used by clients to declare objects of the type. For the client to legally do so — that is, for the compiler to accept this usage and process the code — the use of the supplier's type has to satisfy the scope and visibility rules of the programming language.

Good implementations harness these visibility rules to adhere to the software engineering principles of information hiding and abstraction, both of which require that nothing of the implementation be made visible to clients unless necessary. Compiler enforcement ensures rigorous adherence to those principles.

Therefore, modern languages provide some way to express this control. For example, in Ada, a package can have both a *public* part and a *private* part. Clients have no compile-time visibility to the private part, nor to the package body, as both parts contain implementation artifacts. In class-oriented languages, parts of the class can be marked as *public*, *private*, and *protected* (the details depend on the specific language).

The idioms *Abstract Data Types* (page 11) and *Abstract Data Machines* (page 17) are prime examples used throughout the other idioms.

[8] https://learn.adacore.com/courses/advanced-ada/index.html#advanced-ada-course-index
[9] http://www.ada-auth.org/standards/22rm/html/RM-1-3.html
[13] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.

The idioms explored in *Fundamental Packages* (page 7) are largely variations on expressing this control in Ada.

More details on the topic are provided in those idioms.

1.2.3 Views

In Ada, a *view* of an entity defines what the developer can legally do with that entity. For example, the declaration of an object defines a view of that object. The operations allowed by that view are determined by the type used to declare the object: a signed integer type would allow signed integer numeric operations, but not, say, bit-level operations, nor array indexing, and so on. Furthermore, the view includes whether the object is a constant.

An entity can have more than one view, depending on where in the text of the source code a view of that entity is considered. For example, let's say that the integer object introduced above is in fact a variable. Within the scope of that variable, we can refer to it by that name and update the value using assignment statements. However, if we pass that variable as the argument to a procedure call, within that subprogram (for that call) the view specifies a different name for the argument, i.e., the formal parameter name. Moreover, if that formal parameter is a mode-in parameter, within that procedure body the view of the actual parameter is as if it were a constant rather than a variable. No assignments via the formal parameter name are allowed because the view at that point in the text — within that procedure body — doesn't allow them, unlike the view available at the point of the call.

As another example, consider a tagged type named `Parent`, and a type derived from it via type extension, named `Child`. It is common for a derived type to have either additional components, or additional operations, or both. For a given object of the `Child` type, the view via type `Child` allows the developer to refer to the extended components and/or operations. But we can convert the `Child` object to a value of the `Parent` type using what is known as a *view conversion*. With that `Parent` view of the `Child` object, we can only refer to those components and operations defined for the `Parent` type. The compiler enforces this temporary view.

For further details about view conversions, please refer to that specific section of the Advanced Ada course[10].

Views are a fundamental concept in Ada. Understanding them will greatly facilitate understanding the rules of the language in general.

1.2.4 Partial and Full Views

Like objects, types also can have more than one view, again determined by the place in the program text that a view is considered. These views can be used to apply information hiding and abstraction.

The declaration of a private type defines a *partial view* of a type that reveals only some of its properties: the type name, primarily, but in particular not the type's representation. For example:

```ada
type Rotary_Encoder is private;
```

Private type declarations must occur in the *public part* of a package declaration. Anything declared there is compile-time visible to clients of the package so the type's name is visible, and potentially some other properties as well. Clients can therefore declare objects of the type name, for example, but must adhere to their partial view's effect on what is compile-time visible.

The private type's full representation must be specified within the *private part* of that same package declaration. For example:

[10] https://learn.adacore.com/courses/advanced-ada/parts/data_types/types.html#adv-ada-view-conversion

```
type Rotary_Encoder is record ... end record;
```

Therefore, within that package private part and within the package body the *full view* is available because full representation information is compile-time visible in those regions. (Parts of child units have the full view as well.) This view is necessary in those two regions of the package because the representation details are required in order to implement the corresponding operations, among other possibilities.

Because the clients only have the partial view they do not have compile-time visibility to the type's internal representation. Consequently, the compiler will not allow representation-specific references or operations in client code. The resulting benefit is that clients are independent of the type's representation and, therefore, it can be changed without requiring coding changes in the clients. Clients need only be recompiled in that case.

This application of information hiding has real-world cost benefits because changing client code can be prohibitively expensive. That's one reason why the maintenance phase of a project is by far the most expensive phase. Another reason is that *maintenance* is often a euphemism for new development. Either way, change is involved.

As a result, when defining types, developers should use private types by default, only avoiding them when they are not appropriate. Not using them should be an explicit design choice, a line item in code reviews. Not defining a major abstraction as a private type should be suspect, just as using a `struct` rather than a `class` in C++ should be suspect in that case. (In C++ anything a `struct` contains is compile-time visible to clients by default.)

For further details about type views, please refer to that specific section of the Advanced Ada course[11].

[11] https://learn.adacore.com/courses/advanced-ada/parts/data_types/types.html#adv-ada-type-view

CHAPTER

TWO

ESSENTIAL DESIGN IDIOMS FOR PACKAGES

2.1 Motivation

Packages, especially library packages, are modules, and as such are the fundamental building blocks of Ada programs. There is no language-prescribed way to use packages when designing an application, the language just specifies what is legal. However, some legal approaches are more advisable than others.

Specifically, packages should exhibit high cohesion and loose coupling[14]. Cohesion is the degree to which the declarations within a module are related to one another, in the context of the problem being solved. Unrelated entities should not be declared in the same module. This allows the reader to focus on one primary concept, which should be the subject of the package. Coupling is the degree to which a module depends upon other modules. Loose coupling enhances comprehension and maintenance because it allows readers and future developers to examine and modify the module in relative isolation. Coupling and cohesion are interrelated: higher cohesion tends to result in less coupling.

2.2 Implementation(s)

Three idioms for packages were envisioned when the language was first designed. They were introduced and described in detail in the Rationale document for the initial language design[15] and were further developed in Grady Booch's book *Software Engineering with Ada*[16], a foundational work on design with the (sequential part of the) language. Booch added a fourth idiom, the Abstract Data Machine, to the three described by the Rationale. These four idioms have proven themselves capable of producing packages that exhibit high cohesion and loose coupling, resulting in more comprehensible and maintainable source code.

These idioms pre-date later package facilities, such as private packages and hierarchical packages. We describe idioms for those kinds of packages separately.

Two of the simpler idioms are described here. The other two, that are more commonly used, are described in two separate, dedicated entries within this document.

Generic packages are not actually packages, but their instantiations are, so these design idioms apply to generic packages as well.

Because these are idioms for modules, we differentiate them by what the package declarations will contain. But as you will see, what they can contain is a reflection of the degree of information hiding involved.

[14] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.
[15] J. Ichbiah, J. Barnes, and others. *Rationale for the Design of the Ada Programming Language*. 1986.
[16] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, 1983.

2.2.1 Named Collection of Declarations

In the first idiom, the package declaration can contain other declarations only for the following:

- Objects (constants and variables)
- Types
- Exceptions

The idea is to factor out common content required by multiple clients. Declaring common content in one place and letting clients reference the one unit makes the most sense.

For example, the following package declares several physical constants used in a high-fidelity aircraft simulator. These constants are utilized throughout the simulator code, so they are declared in one place and then referenced as needed:

```
package Physical_Constants is
   Polar_Radius    : constant  := 20_856_010.51; --  feet
   Equatorial_Radius : constant  := 20_926_469.20; --  feet
   Earth_Diameter : constant  :=
     2.0 * ((Polar_Radius + Equatorial_Radius)/2.0);
   Gravity  : constant  := 32.1740_4855_6430_4; --  feet/second**2
   Sea_Level_Air_Density   : constant  := 0.002378; --  slugs/foot**3
   Altitude_Of_Tropopause  : constant  := 36089.0; --  feet
   Tropopause_Temperature  : constant  := -56.5; --  degrees-C
end Physical_Constants;
```

No information hiding is occurring when using this idiom.

2.2.1.1 Pros

Packages designed with this idiom will have high cohesion and low coupling.

The idiom also enhances maintainability because changes to the values, if necessary, need only be made in one place, although in this particular example, we would hope that no such changes will be made.

2.2.1.2 Cons

When a library package contains variable declarations, these variables comprise global data. In this sense, *global* means potential visibility to multiple clients. Global data should be avoided by default, because the effects of changes are potentially pervasive, throughout the entire set of clients that have visibility to it. In effect the developer must understand everything before changing anything. The introduction of new bugs is a common result. But if, for some compelling reason, the design really called for global data, this idiom provides the way to declare it. Note also that global *constants* are less problematic than variables because they can't be changed.

2.2.2 Groups of Related Program Units

In this idiom, the package can contain all of the declarations allowed by the first idiom, but also contains declarations for operations. These are usually subprograms but other kinds of declarations are also allowed such as protected types and objects. Hence these packages can contain:

- Objects (constants and variables)
- Types
- Exceptions
- Operations

Our intent is that the types declared in the package are used by the operations declared in the package, typically in their formal parameters and/or function return types. In this idiom, however, the types are not private.

For example:

```
package Linear_Algebra is
   type Vector is array (Positive range <>) of Real;
   type Matrix is array (Positive range <>, Positive range <>) of Real;
   function "+" (Left, Right : Vector) return Vector;
   function "*" (Left, Right : Vector) return Matrix;
   -- ...
end Linear_Algebra;
```

In this example, `Vector` and `Matrix` are the types under consideration. The type `Real` might be declared here too, but it might be better declared in a *Named Collection of Declarations* (page 8) package referenced in a with_clause. In any case, this package declares types and subprograms that manipulate values of those types.

One might also declare variables in the package, but those should not be the central purpose of the package. For example, perhaps we want to have a variable whose value is used as the default for some formal parameters. Clients can change the default for subsequent calls by first assigning a different value to the variable, unlike a hardcoded literal chosen by the developer. It would look like this:

```
Default_Debounce_Time : Time_Span := Milliseconds (75);
--  The default amount of time used to debounce an input pin.
--  This value is tunable.

procedure Await_Active
  (This : Discrete_Input;
   Debounce_Time : Time_Span := Default_Debounce_Time);
```

With this idiom, information hiding applies to the implementation of the visible subprograms in the package body as well as any internal entities declared in the body and used in implementing the visible subprograms.

As mentioned, these idioms apply to generic packages as well. For example, a more realistic approach would be to make type `Real` be a generic formal type:

```
generic
   type Real is digits <>;
package Linear_Algebra is
   type Vector is array (Positive range <>) of Real;
   type Matrix is array (Positive range <>, Positive range <>) of Real;
   function "+" (Left, Right : Vector) return Vector;
   function "*" (Left, Right : Vector) return Matrix;
   --  ...
end Linear_Algebra;
```

2.2.2.1 Pros

The types and the associated operations are grouped together and are hence highly cohesive. Such packages usually can be loosely coupled as well.

Clients have all the language-defined operations available that the type representations provide. In the case of `Vector` and `Matrix`, clients have compile-time visibility to the fact they are array types. Therefore, clients can manipulate `Vector` and `Matrix` values as arrays: for example, they can create values via aggregates and use array indexing to access specific components.

2.2.2.2 Cons

Clients can write code that depends on the type's representation, and can be relied upon to do so. Consequently, a change in the representation will potentially require redeveloping the client code, which could be extensive and expensive. That is a serious disadvantage.

However, compile-time visibility to the type representations may be necessary to meet client expectations. For example, engineers expect to use indexing with vectors and matrices. As of Ada 2012, developers can specify the meaning of array indexing but the approach is fairly heavy.

2.3 Notes

1. The rules for what these idiomatic packages contain are not meant to be iron-clad; hybrids are possible but should be considered initially suspect and reviewed accordingly.

CHAPTER

THREE

ABSTRACT DATA TYPES

3.1 Motivation

In the *Groups of Related Program Units* (page 8) idiom, client compile-time visibility to the type's representation is both an advantage and a disadvantage. Visibility to the representation makes available the expressiveness of low-level syntax, such as array indexing and aggregates, but in so doing allows client source code to be dependent on the representation. In many cases, the resulting economic and engineering disadvantages of visibility on the representation will outweigh the expressiveness advantages.

For the sake of illustration, let's create a *stack* type that can contain values of type **Integer**. (We use type **Integer** purely for the sake of convenience.) Let's also say that any given Stack object can contain at most a fixed number of values, and arbitrarily pick 100 for that upper bound. The likely representation for the Stack type will require both an array for the contained values and a *stack pointer* indicating the *top* of the stack. Hence this will be a composite type, probably a record type. If we use the *Groups of Related Program Units* (page 8) idiom the code might look like this:

```
package Integer_Stacks is
   Capacity : constant := 100;
   type Content is array  (1 .. Capacity) of Integer;
   type Stack is record
      Values : Content;
      Top    : Integer range 0 .. Capacity := 0;
   end record;
   procedure Push  (This : in out Stack; Item : in Integer);
   procedure Pop  (This : in out Stack;  Item : out Integer);
   function Empty (This : Stack) return Boolean;
end Integer_Stacks;
```

With this design the compiler will allow client code to directly read and update the two components within any Stack object. For example, given some Stack variable named X, the client can read the value of X.Top, say to determine if X is empty. But by the same token, the client code could change X.Top to some arbitrary value unrelated to the logical top of the stack, completely violating stack semantics.

As a result, where would one look in the source code to find a bug in the handling of some Stack object? It could be literally anywhere in all the client code that uses package Integer_Stacks.

Similarly, changes to the internal representation of a type may become necessary as new requirements are identified. At best, the client code will now fail to compile, making identification of the problem areas simple. At worst, the client code will remain legal but no longer functional. Perhaps an additional component was added that the original components now rely upon, or the original components are used in new ways. Conceivably every client use of Integer_Stacks might need to be changed. Once we find them all we'll have to rewrite them to address the changes in the representation. That's potentially very expensive, perhaps prohibitively so. Worse, our *fixes* will likely introduce new bugs.

These disadvantages argue for an alternative. That is the purpose of this next idiom, known as the Abstract Data Type (ADT)[17],[18].

3.2 Implementation(s)

Abstraction is one of the central principles of software engineering because it is one of the primary ways that humans manage complexity. The idea is to focus on the essentials, in effect the *what*, while ignoring all the inessential implementation details, i.e., the *how*. For example, when we drive a car and want to stop, we press the brake pedal. We don't also think about how the pedal makes the car stop, just that it does so. That's an example of abstraction. In the same way, we know that pressing the accelerator pedal increases the speed of the car, that rotating the steering wheel changes the direction of travel, and so on. If to control the car we had to think about how each part actually works — the brake cylinder and brake pads, the fuel injectors, the spark plugs, the steering shaft, the tie rods, and everything else — we'd certainly crash.

We use abstraction in programming for the same reason. In higher-level languages, an array is an abstraction for the combination of a base address and offset. A file system is composed of a number of layered abstractions, including files (at the top), then tracks, then sectors, then blocks, and ultimately down to individual bytes. A data structure, such as a stack, a queue, or a linked list, is an example of an abstraction, as is a valve, an air-lock, and an engine when represented in software. Even procedures and functions are abstractions for lower-level operations. Decomposing via abstractions allows us to manage complexity because at any given layer we can focus on *what* is being done, rather than how.

Therefore, an abstract data type is a type that is abstract in the sense that[18]:

- It is a higher level of abstraction than the built-in programming language types.
- It is functionally characterized entirely by the operations defined by the ADT itself, along with the common basic operations such as assignment, object declarations, parameter passing, and so on. In particular, clients are not allowed to perform operations that are determined by the type's internal representation. Ideally, this protection is enforced by tools.

The ADT may also be abstract in the sense of object-oriented programming but that is an unrelated issue.

In Ada we use *private types* to define abstract data types because private types make the type's name, but not its representation, visible to clients. These types are composed using syntactical building blocks: a package declaration, separated into two parts, containing a type declared in two parts, and containing declarations for subprograms to manipulate objects of the type via parameters. The compiler uses the building-blocks' compile-time visibility rules to enforce the protections against representation-based operations. (We assume the reader is familiar with private types, but this is such an important, central facility in Ada that we will explain them in some detail anyway.)

Therefore, an ADT package declaration may contain any of the following:

- Constants (but probably not variables)
- A private type
- Ancillary Types
- Exceptions
- Operations

[17] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, 1983.
[18] B. Liskov and S. Zilles. *Programming with Abstract Data Types*. ACM SIGPLAN symposium on Very high level languages, 1974.

If possible, you should declare at most one private type per ADT package. This keeps things simple and follows the "cohesive" principle. (Note that the *limited-with* construct directly facilitates declaring mutually-dependent private types that are each declared in their own dedicated packages). However, it's not unreasonable to declare more than one private type in the same package, especially if one of the types is clearly the primary type and the other private type is related to the first. For example, in defining an ADT for a maze, we could declare a private type named `Maze` to be the primary abstraction. But mazes have positions within them, and as clients have no business knowing how positions are represented, both `Maze` and `Position` could reasonably be declared as private types in the same package.

You may use any form of private type with this idiom: basic private types, tagged/abstract/limited private types, private type extensions, and so forth. What's important is that the representation occurs in the private part so that it's not compile-time visible to clients.

The abstraction's operations consist of subprograms that each have one or more formal parameters of the type. Clients will declare objects of the type and pass these objects as formal parameters to manipulate those objects.

The operations are known as *primitive operations* because they have the compile-time visibility to the private type's representation necessary to implement the required behavior.

Clients can create their own operations by calling the type's primitive operations, but client's can't compile any operation that manipulates the internal representation.

Consider the following revision to the package `Integer_Stacks`, now as an ADT:

```
package Integer_Stacks is
   type Stack is private;
   procedure Push (This : in out Stack; Item : in Integer);
   procedure Pop (This : in out Stack;  Item : out Integer);
   function Empty (This : Stack) return Boolean;
   Capacity : constant := 100;
private
   type Content is array  (1 .. Capacity) of Integer;
   type Stack is record
      Values : Content;
      Top    : Integer range 0 .. Capacity := 0;
   end record;
end Integer_Stacks;
```

The package declaration now includes the **`private`** reserved word, about half-way down by itself in the example above, thus dividing the package declaration into the *public part* and the *private part*. The compiler only allows clients compile-time visibility to the package public part. No client code that references anything in the private part will compile successfully.

The declaration for the type `Stack` now has two pieces, one in the package visible part and one in the package private part. The visible piece introduces the type name and ends with the keyword **`private`** to indicate that its representation is not provided to clients.

Client code can use the type name to declare objects because the name is visible. Likewise, clients can declare their own subprograms with parameters of type `Stack`, or use type `Stack` as the component type in a composite type declaration. Clients can use a private type in any way that's consistent with the rest of the visible type declaration, except they can't see anything representation-dependent.

The full type definition is in the package private part. Therefore, for any given object of the type, the representation details — the two record components in this example — can't be referenced in client code. Clients must instead only use the operations defined by the package, passing the client objects as the actual parameters. Only the bodies of these operations have compile-time visibility to the representation of the `Stack` parameters, so only they can implement the functionality for those parameters.

Because package-defined subprograms are the only code that can access the internals of objects of the type, the designer's intended abstract operations are strictly enforced. They are the only manipulations that a client can perform. As we mentioned, basic operations such as assignment are allowed, unless the ADT is *limited* as well as private, but these basic operations do not violate the abstraction.

You may, of course, also require other ancillary type declarations in the package, either for the implementation or as types for additional parameters for the visible operations. The array type `Content` is an example of the former case. When it is strictly an implementation artifact, as in this case, it should be in the private part so that it's hidden from clients.

The ADT idiom extends the information hiding applied by the *Groups of Related Program Units* (page 8) idiom to include the type's representation.

The compile-time lack of visibility to the representation means that clients no longer have a way to construct ADT values from the constituent parts. For example, record aggregates are no longer possible for clients using the `Stack` ADT. Likewise, clients no longer have a way to read the individual constituent components. (Whether doing so is appropriate will be addressed below.) Therefore, an ADT package may include *constructor* and *selector/accessor* subprograms. (The term *constructor* is only conceptually related to the same term in some other languages, such as C++.)

For an example of an abstraction that includes constructors and selectors, imagine there is no language-defined `Complex` number type. We could use the following ADT approach:

```ada
package Complex_Numbers is
   type Complex_Number is private;
   --  function operating on Complex_Number, eg "+" ...
   --  constructors and selectors/accessors
   function Make (Real_Part, Imaginary_Part : Float) return Complex_Number;
   function Real_Part (This : Complex_Number) return Float;
   function Imaginary_Part (This : Complex_Number) return Float;
private
   type Complex_Number is record
      Real_Part : Float;
      Imaginary_Part : Float;
   end record;
end Complex_Numbers;
```

In the above, the function `Make` is a constructor that replaces the use of aggregates for constructing `Complex_Number` values. Callers pass two floating-point values to be assigned to the components of the resulting record type. In the `Stack` ADT, a constructor for `Stack` objects wasn't required because any stack has a known initial state, i.e., empty, and the component default initialization is sufficient to achieve that state. Complex numbers don't have any predeterminable state so the constructor is required.

Likewise, functions `Real_Part` and `Imaginary_Part` are selector/accessor functions that return the corresponding individual component values of an argument of type `Complex_Number`. They are needed because the mathematical definition of complex numbers has those two parts, so clients can reasonably expect to be able to get such values from a given object. (The function names need not be distinct from the component names, but can be if desired.)

However, by default, selector/accessor functions are not included in the ADT idiom, and especially not for every component of the representation. There are no *getter* operations if you are familiar with that term.

There may be cases when what looks like an accessor function is provided, when in fact the function computes the return value. Similarly, there may be functions that simply return the value of a component but are part of the abstraction and happen to be implementable by returning the value of a component. For example, a real stack's ADT package would include a function indicating the extent of the object — that is, the number of values currently contained. In our example implementation the `Top` component happens to indicate that value, in addition to indicating the current top of the stack. The body of the Extent function can then be as follows:

```
function Extent (This : Stack) return Natural is (This.Top);
```

But a different representation might not have a Top component, in which case the function would be implemented in some other way. (For example, we could have declared a subtype of **Natural**, using Capacity as the upper bound, for the function result type.)

You should not include true *getter* functions that do not meet an abstraction-defined requirement and exist purely to provide client access to the otherwise hidden representation components included. Their usage makes the client code dependent on the representation, just as if the client had direct access. For the same reason, by default there are no *setter* procedures for the representation components. Both kinds of operations should be considered highly suspect. There's no point in hiding the representation if these operations will make it available to clients, albeit indirectly.

3.3 Pros

The advantages of an ADT are due to the strong interface presented, with guaranteed enforcement by the compiler rather than by reliance on clients' good behavior. The ADT designer can rely on client adherence to the intended abstraction because client code that violates the designer's abstraction by directly manipulating the internals of the type will not compile; clients must call the designer's operations to manipulate the objects.

A package defining a strong interface will exhibit high cohesion, thereby aiding comprehension and consequently easing both development and maintenance.

An ADT enhances maintainability because a bug in the ADT implementation must be in the package that defines the ADT itself. The rest of the application need not be explored because nothing elsewhere that accessed the representation would compile. (We ignore child packages for the time-being.) The maintenance phase is the most expensive of the project phases for correcting errors, so this is a significant benefit.

Although changes to the internal representation of an ADT may become necessary, the scope of those changes is limited to the ADT package declaration and body because legal client code cannot depend on the representation of a private type. Consequently, changes to the type's representation can only require recompilation (and hence relinking) of client code, but not rewriting.

A change in representation may have non-functional considerations that prompt a change in client usage, such as performance changes, but it will not be a matter of the legality of the client code. Illegal client usage of an ADT wouldn't have compiled successfully in the first place.

The private type is the fundamental approach to creating abstractions in Ada, just as the use of the *public*, *private*, and *protected* parts of classes is fundamental to creating abstractions in class-oriented languages. Not every type can be private, as illustrated by the client expectation for array indexing in Ada prior to Ada 2012. Not every type should be private, for example those that are explicitly numeric. But the ADT should be the default design idiom expression.

3.4 Cons

There is more source code text required in an ADT package compared to the idiom in which the representation is not hidden (the *Groups of Related Program Units* (page 8)). The bulk of the additional text is due to the functions and procedures required to provide the capabilities that the low-level representation-based syntax might have provided, i.e., the *constructor* and *selector/accessor* functions. We say *might have provided* because these additional operations are by no means necessarily included. In general, the additional text required for private types is worth the protections afforded.

3.5 Relationship With Other Idioms

The package-oriented idioms described here and *previously* (page 7) are the foundational program composition idioms because packages are the primary structuring unit in Ada. That is especially true of the *Abstract Data Type* (page 11) idiom, which is the primary type specification facility in Ada. We will describe additional package-oriented idioms, especially regarding hierarchical packages, but those kinds of packages are optional.

The basic package is not optional in Ada for a program of any significant size or complexity. (One could have a program consisting entirely of the main program, but either that program is relatively simple and small or it is badly structured.) As a consequence, other idioms will exist within packages designed using one of these idioms or some other package idiom.

3.6 Notes

1. With the package idioms that declare one or more types, especially the ADT idiom, the principle of Separation of Concerns dictates that objects of the type used by clients be declared by clients in client units, not in the same package that declares the type or types.
2. The Ada Rationale document did not introduce the concept of Abstract Data Types. The ADT concept had already been introduced and recognized as effective when the first version of Ada was being designed[Page 12, 18]. The Ada language requirements document, *Steelman*[19], uses the term "Encapsulated Definitions" and describes the information hiding to be provided. Steelman does not specify the implementation syntax because requirements documents do not include such directives. The language designers implemented those requirements via package private parts and private types.
3. The ADT is the conceptual foundation for the *class* construct's visibility control in some class-oriented languages.

[19] HOLWG. *Department of Defense Requirements for High Order Computer Programming Language "STEELMAN"*. Department of Defense, 1978.

CHAPTER

FOUR

ABSTRACT DATA MACHINES

4.1 Motivation

In some systems, only one logical "instance" of an abstraction should exist in the software. This requirement may stem from the functionality involved. For example, a subsystem-level software logging facility should be unique at that level. Likewise, the function of a hardware device may be such that only one instance should exist in both the system and the software. A security device that validates users would be an example. Another reason can be simple physical reality. There might be only one on-board or on-chip device of some sort. Execution on that board or chip entails there being only one such device present.

How can the software representing the abstraction best implement this requirement?

The Abstract Data Type (ADT) *Abstract Data Type* (page 11) idiom is the primary abstraction definition facility in Ada. Given an ADT that provides the required facility you could simply declare a single object of the type. But how could you ensure that some other client, perhaps in the future, doesn't declare another object of the type, either accidentally or maliciously?

As a general statement about program design, if there is something that must not be allowed, the ideal approach is to use the language rules to make it impossible. That's far better than debugging. For example, we don't want clients to have compile-time access to internal representation artifacts, so we leverage the language visibility rules to make such access illegal. The compiler will then reject undesired references, rigorously.

The occasional need to control object creation is well-known, so much so that there is a design pattern for creating an ADT in which only one instance can ever exist. Known as the "singleton" pattern, the given programming language's rules are applied such that only the ADT implementation can create objects of the type. Clients cannot do so. The implementation only creates one such object, so multiple object declarations are precluded.

Singletons can be expressed easily in Ada *Controlling Object Initialization and Creation* (page 35) but there is an alternative in this specific situation.

This idiom entry describes the alternative, known as the Abstract Data Machine (ADM). The Abstract Data Machine was introduced by Grady Booch[21] as the Abstract State Machine, but that name, though appropriate, encompasses more in computer science than we intend to evoke.

4.2 Implementation(s)

The ADM is similar to the ADT idiom in that it presents an abstraction that doesn't already exist in the programming language. Furthermore, like the ADT, operations are provided to clients to manipulate the abstraction state, which is not otherwise compile-time visible to client code.

[21] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, 1983.

Unlike the ADT, however, the ADM does not define the abstraction as a type. To understand this difference, first recall that type declarations are descriptions for objects that will contain data (the state). For example, our earlier Stack ADT was represented as a record containing two components: an array to hold the values logically contained by the Stack and an integer indicating the logical top of that array (the stack pointer). No data actually exists, i.e., is allocated storage, until objects of the type are declared. Clients can declare as many objects of type Stack as they require and each object has a distinct, independent copy of the data.

Continuing the Stack example, clients could choose to declare only one object of the Stack type, in which case only one instance of the data described by the Stack type will exist:

```
Integer_Stack : Stack;
```

But, other than convenience, there is no *functional* difference from the client declaring individual variables of the representational component types directly, one for the array and one for the stack pointer:

```
Capacity : constant := 100;
type Content is array  (1 .. Capacity) of Integer;
Values : Content;
Top    : Integer range 0 .. Capacity := 0;
```

or even this, using an anonymously-typed array:

```
Capacity : constant := 100;
Values : array  (1 .. Capacity) of Integer;
Top    : Integer range 0 .. Capacity := 0;
```

If there is to be only one logical stack, these two variables will suffice.

That's what the ADM does. The state variables are declared directly within a package, rather than as components of a type. In that way the package, usually a library package, declares the necessary state for a single abstraction instance. But, as an abstraction, those data declarations must not be compile-time visible to clients. Therefore, the state is declared in either the package private part or the package body. Doing so requires that visible operations be made available to clients, as with the ADT. Hence the combination of a package, the encapsulated variables, and the operations is the one instance of the abstraction. That combination is the fundamental concept for the ADM idiom.

Therefore, the package declaration's visible section contains only the following:

- Constants (but almost certainly not variables)
- Ancillary Types
- Exceptions
- Operations

The package declaration's private part and the package body may contain all the above, but one or the other (or both) will contain variable declarations representing the abstraction's state.

Consider the following ADM version of the package Integer_Stacks, now renamed to Integer_Stack for reasons we will discuss shortly. In this version we declare the state in the package body.

```
package Integer_Stack is
   procedure Push (Item : in Integer);
   procedure Pop (Item : out Integer);
   function Empty return Boolean;
   Capacity : constant := 100;
end Integer_Stack;
```

(continues on next page)

(continued from previous page)

```
package body Integer_Stack is
   Values : array  (1 .. Capacity) of Integer;
   Top    : Integer range 0 .. Capacity := 0;
   procedure Push  (Item : in Integer) is
   begin
      --  ...
      Top := Top + 1;
      Values (Top) := Item;
   end Push;
   procedure Pop (Item : out Integer) is ... end Pop;
   function Empty return Boolean is (Top = 0);
end Integer_Stack;
```

Note how the procedure and function bodies directly access the local variables hidden in the package body.

For those readers familiar with programming languages that can declare entities to be "static," the effect is as if the two variables in the package body are static variables.

When using this idiom, there is only one stack (containing values of some type, in this case type Integer). That's why we changed the name of the package from `Integer_Stacks`, i.e., from the plural form to the singular. It may help to note that what is now the package name was the name of the client's variable name when there was a `Stack` type involved.

As with the ADT idiom, clients of an ADM can only manipulate the encapsulated state via the visible operations. The difference is that the state to be manipulated is no longer an object passed as an argument to the operations. For illustration, consider the `Push` procedure. The ADT version requires the client to pass the `Stack` object intended to contain the new value (i.e., the actual parameter for the formal named `This`):

```
procedure Push (This : in out Stack; Item : in Integer);
```

In contrast, the ADM version has one less formal parameter, the value to be pushed:

```
procedure Push (Item : in Integer);
```

Here is a call to the ADM version of Push:

```
Integer_Stack.Push (42);
```

That call places the value 42 in the (hidden) array `Integer_Stack.Values` located within the package body. Compare that to the ADT approach, in which objects of type `Stack` are manipulated:

```
Answers : Stack;
--  ...
Push (Answers, 42);
```

That call places the value 42 in the (hidden) array `Answers.Values`, i.e., within the `Answers` variable. Clients can declare as many `Stack` objects as they require, each containing a distinct copy of the state defined by the type. In the ADM version, there is only one stack and therefore only one instance of the state variables. Hence the `Stack` formal parameter is not required.

Rather than declare the abstraction state in the package body, we could just as easily declare it in the package's private section:

```
package Integer_Stack is
   procedure Push (Item : in Integer);
   procedure Pop (Item : out Integer);
   function Empty return Boolean;
```

(continues on next page)

(continued from previous page)

```
   Capacity : constant := 100;
private
   Values : array  (1 .. Capacity) of Integer;
   Top    : Integer range 0 .. Capacity := 0;
end Integer_Stack;
```

Doing so doesn't change anything from the client code point of view; just as clients have no compile-time visibility to declarations in the package body, they have no compile-time visibility to the items in the package private part. This placement also doesn't change the fact that there is only one instance of the data. We've only changed where the data are declared. (We will ignore the effect of child packages here.)

Because the two variables are implementation artifacts we don't declare them in the package's visible part.

Note that the private section wasn't otherwise required when we chose to declare the data in the package body.

The ADM idiom applies information hiding to the internal state, like the ADT idiom, except that the state is not in an object declared by the client. Also, like the *Groups of Related Program Units* (page 8), the implementations of the visible subprograms are hidden in the package body, along with any non-visible entities required for their implementation.

There are no constructor functions returning a value of the abstraction type because the abstraction is not represented as a type. However, there could be one or more initialization procedures, operating directly on the hidden state in the package private part or package body. In the Stack ADM there is no need for them because of the abstraction-appropriate default initial value, as is true of the ADT version.

The considerations regarding selectors/accessors are the same for the ADM as for the ADT idiom, so they are not provided by default. Also like the ADT, so-called *getters* and *setters* are highly suspect and not provided by the idiom by default.

As mentioned, the ADM idiom can be applied to hardware abstractions. For example, consider a target that has a single on-board rotary switch for arbitrary use by system designers. The switch value is available to the software via an 8-bit integer located at a dedicated memory address, mapped like so:

```
Switch : Unsigned_8 with
   Volatile,
   Address => System.Storage_Elements.To_Address (16#FFC0_0801#);
```

Reading the value of the memory-mapped Switch variable provides the rotary switch's current value.

However, on this target the memory at that address is read-only, and rightly so because the only way to change the value is to physically rotate the switch. Writing to that address has no effect whatsoever. Although doing so is a logical error no indication is provided by the hardware, which is potentially confusing to developers. It certainly looks like a variable, after all. Declaring it as a constant wouldn't suffice because the user could rotate the switch during execution.

Furthermore, although mapped as a byte, the physical switch has only 16 total positions, read as the values zero through fifteen. An unsigned byte has no such constraints.

The compiler will enforce the read-only view and the accessor operation can handle the range constraint. The ADM is a reasonable choice because there is only one such physical switch; a type doesn't bring any advantages in this case. The following illustrates the approach:

```
with Interfaces; use Interfaces;
package Rotary_Switch is
```

(continues on next page)

(continued from previous page)

```
   subtype Values is Unsigned_8 range 0 .. 15;
   function State return Values;
end Rotary_Switch;
```

Clients can then call the function `Rotary_Switch.State` to get the switch's current value, as a constrained subtype. The body will handle all the details.

```
with System.Storage_Elements;  use System.Storage_Elements;
package body Rotary_Switch is
   Switch : Unsigned_8 with Volatile, Address => To_Address (16#FFC0_0801#);
   function State return Values is
   begin
      if Switch in Values then
         return Switch;
      else
         raise Program_Error;
      end if;
   end State;
end Rotary_Switch;
```

The range check in the function body might be considered over-engineering because the switch is a physical device that cannot have more than 16 values, but physical devices have a habit of springing surprises. Note that attribute Valid[20] would not be useful here because there are no invalid bit patterns for an unsigned integer. If, on the other hand, we were working with an enumeration type, for example, then `'Valid` would be useful.

4.3 Pros

In terms of abstraction and information hiding, the ADM idiom provides the same advantages as the ADT idiom: clients have no visibility to representation details and must use the operations declared in the package to manipulate the state. The compiler enforces this abstract view. The ADM also has the ADT benefit of knowing where any bugs could possibly be located. If there is a bug in the behavior, it must be in the one package defining the abstraction itself. No other code would have the compile-time visibility necessary.

In addition, less source code text is required to express the abstraction.

4.4 Cons

The disadvantage of the ADM is the lack of flexibility.

An ADM defines only one abstraction instance. If more than one becomes necessary, the developer must copy-and-paste the entire package and then change the new package's unit name. This approach doesn't scale well.

Furthermore, the ADM cannot be used to compose other types, e.g., as the component type in an array or record type. The ADM cannot be used to define the formal parameter of a client-defined subprogram, cannot be dynamically allocated, and so on.

But if one can know with certainty that only one thing is ever going to be represented, as in the hardware rotary switch example, the ADM limitations are irrelevant.

[20] https://learn.adacore.com/courses/advanced-ada/parts/data_types/types_representation.html#adv-ada-valid-attribute

CHAPTER

FIVE

PROGRAMMING BY EXTENSION

5.1 Motivation

When declaring entities in a package, developers should ensure that the client view — the package visible part — contains no implementation artifacts. Doing so is important conceptually, but also practically, because any declarations visible to clients inevitably will be used by clients and, as a result, will become permanent fixtures because removal would cause expensive changes in the client code.

The intended client API declarations must be in the package visible part, of course. The question, then, is whether to declare implementation artifacts in the package private part or in the package body. Those are the two parts of a package that do not make declarations compile-time visible to client code.

Some of these entities must be declared in the package private part because they are required in the declaration of some other entity appearing in that part. For example, when using the *ADT idiom* (page 11), an ancillary type might be required for the completion of the private type. That was the case with the *ADT version* (page 13) of the `Integer_Stacks` package, repeated here for convenience:

```
package Integer_Stacks is
   type Stack is private;
   --  ...
   Capacity : constant := 100;
private
   type Content is array  (1 .. Capacity) of Integer;
   type Stack is record
      Values : Content;
      Top    : Integer range 0 .. Capacity := 0;
   end record;
end Integer_Stacks;
```

The array type `Content` is required for the `Stack` record component because anonymously-typed array components are illegal. Clients have no business using the type `Content` directly so although it would be legal to declare it in the public part, declaration in the private part is more appropriate.

Likewise, a function called to provide the default initial value for a private type's component must be declared prior to the reference. If the function is truly only part of the implementation, we should declare it in the package private part rather than the public part.

In contrast, there may be implementation-oriented entities that are referenced only in the package body. They could be declared in the package body but could alternatively be declared in the package declaration's private part. Those are the entities (declarations) in question for this idiom.

For a concrete example, here is an elided *ADM version of the stack abstraction* (page 18), with the stack state declared in the package body:

```
package Integer_Stack is
   procedure Push (Item : in Integer);
   procedure Pop (Item : out Integer);
   function Empty return Boolean;
   Capacity : constant := 100;
end Integer_Stack;

package body Integer_Stack is
   Values : array  (1 .. Capacity) of Integer;
   Top    : Integer range 0 .. Capacity := 0;
   procedure Push  (Item : in Integer) is
   begin
      --  ...
      Top := Top + 1;
      Values (Top) := Item;
   end Push;
   procedure Pop (Item : out Integer) is ...
   function Empty return Boolean is ...
end Integer_Stack;
```

We could add the private part to the package declaration and move the state of the *ADM* (page 17) — the two variables in this case — up there without any other changes. The subprogram bodies have the same visibility to the two variables either way. (There is no requirement for the `Content` type because `Values` is not a record component; anonymously-typed array objects are legal.) From the viewpoint of the language and the abstraction, the location is purely up to the developer.

5.2 Implementation(s)

When you have a choice of placement, putting the state in either the package private part or the package body is reasonable, but only one of the two is amenable to future requirements.

Specifically, placement in the private part of the package allows *programming by extension*[23] via hierarchical *child* packages. Child packages can be written immediately after the *parent* package but can also be written years later, thus accommodating changes due to new requirements.

Programming by extension allows us to extend an existing package's facilities without having to change the existing package at all. Avoiding source code changes to the existing package is important because doing so might be very expensive. In certified systems, the changed package would require re-certification, for example. Changes to the parent package are avoidable because child packages have compile-time visibility to the private part of the ancestor package (actually the entire ancestor package hierarchy, any of which could be useful). Thus, the extension in the child package can depend on declarations in the existing parent package's private part.

Therefore, if the developer can know, with certainty, that no visibility beyond the one package will ever be appropriate, the declaration should go in the package body. Otherwise, it should go in the package private part, just in case an extension becomes necessary later.

Using our ADM stack example, we could move the state from the package body to the private part:

```
package Integer_Stack is
   procedure Push (Item : in Integer);
   procedure Pop (Item : out Integer);
   function Empty return Boolean;
   Capacity : constant := 100;
```

(continues on next page)

[23] J. Barnes. *Programming In Ada 95*. Addison-Wesley, 1998.

(continued from previous page)

```
private
   Values : array  (1 .. Capacity) of Integer;
   Top    : Integer range 0 .. Capacity := 0;
end Integer_Stack;
```

Note that the private part was not otherwise required by the language in this example.

With that change, a new child package could be created with extended functionality:

```
package Integer_Stack.Utils is
   procedure Reset;
end Integer_Stack. Utils;

package body Integer_Stack.Utils is
   procedure Reset is
   begin
      Top := 0;
   end Reset;
end Integer_Stack.Utils;
```

These child packages are not client code, they contain extensions to the existing abstraction. Hence they are part of what may be considered a subsystem consisting of the original package and the new child package. The child package contains an extension of the abstraction defined by the parent package, so the child is directly related. Given that characterization of child packages we can say that the parent package private part is not visible to client code and, therefore, does not represent a *leak* of implementation details to clients.

5.3 Pros

We can extend an abstraction without changing the source code defining that abstraction, thereby meeting new requirements without incurring potentially expensive redevelopment.

5.4 Cons

Clients could abuse the hierarchical package visibility rules by creating a child package that doesn't really extend the existing package abstraction.

Abuse of the visibility rules allows child packages that can break the abstraction. For example, if we only change the name of procedure `Reset` in package `Integer_Stack.Utils` to `Lose_All_Contained_Data` then the routine has a rather different complexion.

Similarly, abuse of the visibility rules allows child packages that can indirectly leak state from the parent package. For example:

```
package Integer_Stack.Leaker is
   function Current_Top return Integer;
end Integer_Stack.Leaker;

package body Integer_Stack.Leaker is
   function Current_Top return Integer is (Top);
end Integer_Stack.Leaker;
```

We could do that without even requiring a package body, using an expression function for the completion:

```
package Integer_Stack.Leaker is
   function Current_Top return Integer;
```

(continues on next page)

(continued from previous page)

```
private
   function Current_Top return Integer is (Top);
end Integer_Stack.Leaker;
```

The function must be completed in the private part because that is where compile-time visibility to the parent begins within a child package.

Code reviews are the only way to detect these abuses, although detection of potential cases could be automated with an analysis tool such as Libadalang[22].

5.5 Relationship With Other Idioms

We assume the use of the *Abstract Data Type* (page 11) or *Abstract Data Machine* (page 17) idioms for the existing package abstraction, as well as for the child package.

5.6 Notes

This guideline will already be used when developing a subsystem (a set of related packages in an overall hierarchy) as a structuring approach during initial development. The idiom discussed here is yet another reason to use the private part, but in this case for the sake of the future, rather than initial, development.

The very first version of Ada (Ada 83) did not have hierarchical library units so, typically, anything not required in the private part was declared in the package body. Declaring them in the private part would only clutter the code that had to be there, without any benefit. The author's personal experience and anecdotal information confirms that after Ada 95 introduced hierarchical library units, some declarations in existing package bodies tended to "percolate up" to the package declarations' private parts.

[22] https://github.com/AdaCore/libadalang

CHAPTER

SIX

CONSTRUCTOR FUNCTIONS FOR ABSTRACT DATA TYPES

6.1 Motivation

In languages supporting object-oriented programming (OOP), including Ada, *constructors* are not inherited when one type is derived from another. That's appropriate because, in general, they would be unable to fully construct values for the new type. The purpose of this idiom is to explain how Ada defines constructors, how the language rules prevent constructor inheritance, and how to design the constructor code in light of those rules.

Ada uses tagged types to fully support dynamic OOP. Therefore, in the following, a *derived type* refers to a tagged type that is declared as a so-called *type extension* — a form of inheritance — based on some existing *parent* tagged type. The extension consists of additional components and/or additional or changed operations beyond those inherited from the existing parent type.

This discussion assumes these tagged types are declared in packages designed using the *Abstract Data Type* (page 11) (ADT) idiom. We strongly recommend the reader be comfortable with that idiom before proceeding.

As abstract data types, the parent type is a private type, and the derived type is a *private extension*. A private extension is a type extension declaration that does not reveal the components added, if any. The parent type could itself be an extended type, but the point is that these types will all be private types one way or another. Declarations as private types and private extensions are not required by the language for inheritance, but as argued in the ADT idiom discussion, doing so is recommended in the strongest terms. OOP doesn't change that, and in fact the encapsulation and information hiding that are characteristic of the ADT idiom are foundational principles for OOP types.

For an example of a private extension, given a tagged type named `Graphics.Shape` one can declare a new type named `Circle` via type extension:

```
type Circle is new Graphics.Shape with private;
```

This declaration will be in the public part of a package, but, as a private type extension, the additional components are not compile-time visible to client code, conforming to ADT requirements. That's what the reserved word **private** indicates in the type declaration.

Instead of a distinct constructor syntax, Ada uses regular functions to construct objects. Specifically, so-called *constructor functions* are functions that return an object of the type.

```
type Circle is new Graphics.Shape with private;

function New_Circle (Radius : Float) return Circle;
```

Like any function there may be formal parameters specified, but not necessarily.

Functions and procedures that manipulate objects of the private type are *primitive operations* for the type if they are declared in the same package as the type declaration itself. For procedures, that means they have formal parameters of the type. For functions, that means they either have formal parameters of the type, or return a value of the type, or both.

Declaration within the same package as the type itself provides the compile-time visibility to the type's representation required to implement the subprograms.

Other operations might be declared in the same package too, but if they do not manipulate or return values of the type they are not primitive operations for the type. (Their location in that package is somewhat suspect and should be reviewed explicitly.)

Primitive operations, and only primitive operations, are inherited during type derivation.

If you think in terms of Abstract Data Types all these rules make sense.

Now, here's the rub.

Constructor functions require that same compile-time visibility so the intuitive approach will be to declare them in the same package declaration as the type. As a result, they will be primitive operations for that type.

However, that means that the constructor functions will be inherited, contrary to the expectation for constructors. Therefore, Ada has rules specific to primitive constructor functions that have the effect of preventing their inheritance.

The explanation and illustration for these rules first requires explanation of the word *abstract*. We mentioned above that the package enclosing the type will be designed with the *Abstract Data Type* (page 11) idiom. In that idiom *abstract* means that the type represents an abstraction. (See that section for the details.)

The term *abstract* also has a meaning in OOP, one that is unrelated to an ADT. In OOP, an *abstract* type is one that defines an interface but at most a partial implementation. As such, the type can serve as the ancestor type for derived types but cannot be used to declare objects. An abstract type in Ada includes the reserved word `abstract` in the declaration. For example:

```ada
type Foo is abstract tagged private;
```

Similarly, subprograms can be abstract. These again define an interface, via the subprogram formal parameters and result type, but are not callable units. In Ada these too include the word `abstract` in their declarations, for example:

```ada
procedure Do_Something (This : in out Foo) is abstract;
```

Now we can explain how Ada prevents constructor inheritance.

Whenever a tagged type is extended, all inherited constructor functions automatically become abstract functions for the extended type, just as if they were explicitly declared abstract.

However, only abstract types can legally have abstract primitive operations. Concrete types may not, so that we can never dynamically dispatch to a subprogram without an actual implementation.

Therefore, unless the extended child type is itself abstract, the type extension will be illegal. The compiler will reject the declaration of the child type, thus preventing this inappropriate constructor inheritance.

For an example, both to illustrate the code and the Ada rules, consider this simple package declaration that presents the tagged private type `Graphics.Shape`:

```ada
package Graphics is
   type Shape is tagged private;
   function Make (X, Y : Float) return Shape;
   ...
private
```

(continues on next page)

(continued from previous page)

```
   type Shape is tagged record
      X : Float := 0.0;
      Y : Float := 0.0;
   end record;
end Graphics;
```

Note in particular the primitive constructor function named Make that constructs a value of type Shape.

Because type Shape is tagged, other types can extend it:

```
with Graphics;
package Geometry is
   type Circle is new Graphics.Shape with private;  -- a private extension
   --  ...
private
   type Circle is new Graphics.Shape with record
      Radius : Float;
   end record;
end Geometry;
```

Type `Circle` automatically inherits the components and primitive operations defined by type Shape, including the constructor function Make. No additional declarations are required in order to inherit these operations or components. The inherited operations are now primitive operations for the new type.

Inherited primitive operations have an unchanged formal parameter and result-type profile, except for the controlling parameter type name, so although Make now returns a `Circle` object, the function still only has parameters for the X and Y components. Hence this version of Make could not set the Radius component in the returned `Circle` value to anything other than some default.

Therefore, to prevent this inherited function from being available, two Ada rules come into play. The first rule specifies that the implicit function is inherited as if declared explicitly abstract:

```
function Make (X, Y : Float) return Circle is abstract;
-- as actually inherited, implicitly
```

Note the reserved word **abstract** in the implicit function declaration. This declaration doesn't actually appear in the source code because all the inherited primitive operations are implicitly declared.

Another rule specifies that only abstract types can have abstract primitive subprograms. Type `Circle` is not abstract in this sense, therefore the combination of those two rules makes the `Circle` type extension illegal. Package `Geometry` will not compile successfully.

Failing to compile is safe — it prevents clients from having a callable function that in general cannot suffice — but requires an alternative so that sufficient constructor functions are possible.

Therefore, a general design idiom is required for defining constructor functions for concrete tagged Abstract Data Types.

6.2 Implementation(s)

The general approach uses functions for constructing objects but prevents these functions from being inherited. The problem is thus circumvented entirely.

To prevent their being inherited, the implementation prevents the constructor functions from being primitive operations. However, these functions require compile-time visibility to the parent type's representation in order to construct values of the type, as this typically involves assigning values to components in the return object. The alternative approach must supply the compile-time visibility that primitive operations have.

Therefore, the specific implementation is to declare constructor functions in a separate package that is a *child* of the package declaring the tagged type. This takes advantage of the *hierarchical library units* capability introduced in Ada 95.

Operations declared in a child package are not primitive operations for the type in the parent package, so they are not inherited when that type is extended. Consequently they do not become abstract.

In addition, the required visibility to the parent type's representation in the private part will be available to the functions' implementations because the private part and body of a child package have compile-time visibility to the parent package's private part.

In this idiom, any package declaring a tagged type, either directly or by type extension, will have a *constructors* child package if constructors are required. For example:

```
package Graphics.Constructors is
   function Make (X, Y : Float) return Shape;
end Graphics.Constructors;
```

and similarly, for `Circle`:

```
package Geometry.Constructors is
   function Make (X, Y, R : Float) return Circle;
end Geometry.Constructors;
```

Each of these two package declarations will have a package body containing the body of the corresponding function. In fact such packages can declare as many constructor functions as required, overloaded or not.

Clients that want to use a constructor function will specify the constructor package in the context clauses for their units, as usual. The constructor package body for an extended type might very well do so itself, as shown below:

```
with Graphics.Constructors; use Graphics.Constructors;
package body Geometry.Constructors is
   function Make (X, Y, R : Float) return Circle is
     (Circle'(Make (X, Y) with Radius => R));
end Geometry.Constructors;
```

Of course, the name "Constructors" is not required for the child packages. It could be "Ctors", for example (a name common in C++), or something else. But whatever the choice, regularity enhances comprehension so the same child package name should be used throughout.

6.3 Pros

The issue is sidestepped entirely, and as an additional benefit, the parent packages are that much simpler because the constructor function declarations and bodies are no longer present there. The *constructors* child packages themselves will be relatively simple since they contain only the constructor functions and any ancillary code required to implement them. Simpler code enhances comprehension and correctness.

Having the constructors declared in separate packages applies the principle of Separation of Concerns, between the code defining the type's semantics and the code for constructing objects of the type. This principle also enhances comprehension.

6.4 Cons

There will be a child package for each tagged type that requires constructors, hence more packages and files (assuming one unit per file, which is desirable in itself, even if not required by the language).

Some developers might argue for having fewer files, presumably containing larger units. In the author's experience larger units make comprehension, and therefore correctness, unjustifiably difficult if smaller units are possible. Some units are unavoidably large and complicated but often we can achieve relative simplicity.

For those developers, however, the constructor package could be declared instead as a nested package located within the package defining the tagged type. Doing so would achieve the same effect as using a child package because the contained functions would not be primitive. Therefore, they would not inherited.

This alternative would reduce the number of files back to the minimum. However, the defining package would be relatively more complicated because of this nested package. Note that the nested package declaration would require a nested package body too.

In short, the alternative reduces the number of files at the cost of additional unit complexity. (If the issue with the larger number of files is difficulty in locating individual entities of interest, any decent IDE will make doing so trivial.)

The alternative also loses the distinction between clients that use objects of the type and clients that create those objects, because, with the child package approach, the latter will be the only clients that have context clauses for the constructor packages.

6.5 Relationship With Other Idioms

N/A

6.6 Notes

For those interested, in this section we provide a discussion of alternatives to the implementation presented, and why they are inadequate.

Changing the behavior of an inherited operation requires an explicit conforming subprogram declaration and therefore a new subprogram body for that operation. This change is known as *overriding* the inherited operation.

Package Geometry could declare a function with the additional parameters required to fully construct a value of the new type. In this case the new constructor would include the Radius parameter:

```
function Make (X, Y, Radius : Float) return Circle;
```

But such a function would not be overriding for the inherited version because the parameter and result type profile would be different. This function Make would overload the inherited function, not override it. The inherited function remains visible, as-is.

In fact, we could even have the compiler confirm that this is not an overriding function by declaring it so:

```
not overriding function Make (X, Y, Radius : Float) return Circle;
```

In general, specifying that a subprogram is not overriding is less convenient than specifying that it is overriding. We only do so in these examples to make everything explicit.

Because that new function is not overriding, the inherited version remains implicitly abstract and the type extension remains illegal. Developers could also override the inherited function, which would make the code legal, but as we have said such a function cannot properly construct values in general, and might be called accidentally. For example:

```
with Graphics;
package Geometry is
   type Circle is new Graphics.Shape with private;

   overriding function Make (X, Y : Float) return Circle;

   not overriding function Make (X, Y, Radius : Float) return Circle;
   -- overloading

   ...
private
   --  ...
end Geometry;
```

Although the overridden Make does not have a Radius parameter and could only assign some default to that component, if that default is reasonable then the overridden function could be called on purpose, i.e., not accidentally. That's not a general approach, however.

Alternatively, developers could use procedures as their constructors, with a mode-out parameter for the result. The procedure would not become implicitly abstract in type extensions, unlike a function.

```
package Graphics is
   type Shape is tagged private;
   procedure Make (Value : out Shape;  X, Y : in Float);
private
   --  ...
end Graphics;
```

And then the client extension would inherit the procedure:

```
with Graphics;
package Geometry is
   type Circle is new Graphics.Shape with private;
   --  procedure Make (Value : out Circle;  X, Y : in Float);  -- inherited
private
   --  ...
end Geometry;
```

However, although now legal, the inherited procedure would not suffice, lacking the required parameter for the Radius component.

Developers might then add an overloaded version with the additional parameter:

```
with Graphics;
package Geometry is
   type Circle is new Graphics.Shape with private;

   --  procedure Make (Value : out Circle;  X, Y : in Float);
   -- inherited

   not overriding procedure Make (Value : out Circle;  X, Y, R : in Float);
   -- not inherited
private
   --  ...
end Geometry;
```

But the same issues arise as with functions. Clients might accidentally call the wrong procedure, i.e., the inherited routine that doesn't have a parameter for the `Radius`. That routine would not even mention the `Radius` component, much less assign a default value, so it would have to be overridden in order to do so. This too is not a general approach.

CHAPTER

SEVEN

CONTROLLING OBJECT INITIALIZATION AND CREATION

7.1 Motivation

Developers are responsible for ensuring that no uninitialized objects are read in Ada programs. Default initialization is a good way to meet this requirement because it is guaranteed to happen and requires no actions on the part of the client code. But of the many kinds of types provided by Ada, only access types have a language-defined default initial value. Fortunately, Ada supports user-defined default initialization for user-defined types.

Default initialization is conveniently expressed, especially because components of record types can have default initial values. Record types are perhaps the most commonly used non-numeric type in the language. Sometimes a given type was *wrapped* inside a record type purely for the sake of default component initialization, e.g., numeric types. That wrapping approach is less common than in earlier versions of the language, given the comparatively more recent aspect `Default_Value` for scalar types, and `Default_Component_Value` for scalar array components.

These facilities are often sufficient to express an abstraction's initial state. For example, we can expect that container objects will be initially empty. Consider a bounded stack ADT. The representation is likely a record type containing an array component and a Top component indicating the index of the last array component used. We can default initialize objects to the empty state simply by setting Top to zero in the record component's declaration:

```ada
type Content is array (Positive range <>) of Element;
type Stack (Capacity : Positive) is record
   Values : Content (1 .. Capacity);
   Top    : Natural := 0;
end record;
```

For an unbounded container such as a simple binary tree, if the representation is an access type, the automatic default value **null** initializes `Tree` objects to the empty state.

```ada
package Binary_Trees is
   type Tree is limited private;
   ...
private
   type Leaf_and_Branch is record ...
   type Tree is access Leaf_and_Branch;
   ...
end Binary_Trees;
```

In both cases, simply declaring an object in the client code is sufficient to ensure it is initially empty.

However, not all abstractions have a meaningful default initial state. Default initialization will not suffice to fully initialize objects in these cases, so explicit initialization is required.

An explicit procedure call could be used to set the initial state of an object (passed to a mode-out parameter), but there is no guarantee that the call will occur and no way to force a client to make it.

In contrast, the declaration of the object is guaranteed to occur, and as part of the declaration the object can be given an explicit initial value. The initial value can be specified by a literal for the type, by the value of another object of that type, or by the value of that type returned from a function call.

```ada
declare
   X : Integer := Some_Existing_Integer_Object;
   Prompt : constant String := "Name? ";
   Reply : constant String := Response (Prompt);
begin
   ...
end;
```

The initial value can also specify constraints, if required. In the code above, the object `Prompt` has a lower bound of `Positive'First` and an upper bound set to the length of the literal. The specific bounds of `Reply` are determined by the function, and need not start at `Positive'First`.

An object cannot be used before it is declared. Since this explicit initial value is part of the declaration, the object cannot be read before it is initialized. That fact is the key to the implementation approaches.

However, although the object declaration is guaranteed to occur, explicit initialization is optional. But unlike a procedure call, we can force the initial value to be given. There are two ways to force it, so there are two implementations presented.

In addition, a specific form of explicit initialization may be required because not all forms of initialization are necessarily appropriate for a given abstraction. Imagine a type representing a thread lock, implemented in such a way that default initialization isn't an option. Unless we prevent it, initialization by some other existing object will be possible:

```ada
declare
   X : Thread_Lock := Y;    --  Y is some other Thread_Lock object
begin
   --  ...
end;
```

This would amount to a copy, which might not make sense. Imagine the lock type contains a queue of pending callers...

More generally, if a type's representation includes access type components, initialization by another object will create a shallow copy of the designated objects. That is typically inappropriate.

Using an existing object for the initial value amounts to a complete copy of that other object, perhaps more of a copy than required. For example, consider a bounded container type, e.g., another stack, backed by an array and an index component named Top. At any time, for any stack, the contained content is in the slice of the array from 1 up to Top. Any array component at an index greater than Top has a junk value. Those components may never even have been assigned during use. Now consider the declaration of a `Stack` object, A, whose initial value is that of another existing `Stack` named B.

```ada
A : Stack := B;
```

The entire value of B is copied into A, so `B.Top` is copied to `A.Top`, which makes sense. But likewise, the entire array in B will be copied to the array in A. For a stack with a large backing array that might take a significant amount of time. If B is logically full then the time required for the full array copy is unavoidable. But if only a few values are contained by B, the hit could be avoided by only copying up to Top.

And of course, the initial value might require client-specific information.

Calling a *constructor function* (page 27) for the initial value would be the right approach in these cases, returning an object of the type. The function might even take an existing object as a parameter, creating a new object with only the necessary parts copied.

Therefore, for some abstractions, not only do we need to guarantee explicit object initialization, we may also need to restrict the form of initial value to a function call.

The other purpose of the idiom is controlling, for some type, whether object creation itself is to be allowed by clients. As you will see, controlling object initialization can be used to control object creation.

Preventing object creation is not typical but is not unknown. The singleton design pattern[24] is an example, in which a type is defined but corresponding object creation by clients is not intended. Instead, the abstraction implementation creates a single object of the type. The abstraction is a type, rather than an *ADM* (page 17), for the sake of potential extension via inheritance. We will illustrate this design pattern and implementation using a real-world hardware device.

7.2 Implementation(s)

There are two ways to force an explicit initial value as part of an object declaration. One is a matter of legality at compile-time so it is enforced by the compiler. The other is enforced by a run-time check.

Note that both approaches are type-specific, so when we say *objects* we mean objects of a type that has been designed with one of these two idiom implementations. Neither implementation applies to every object of every type used in the client code. (SPARK, a formal language based closely on Ada, statically ensures all objects are initialized before read.)

The *ADT idiom* (page 11) describes Ada *building blocks* that developers can use to compose types with semantics that we require. We can declare a type to be private, for example, so that the implementation is not compile-time visible to clients.

In addition to private types, we can decorate a type declaration with the reserved word **`limited`** so that assignment is not allowed (among other things) for client objects of the type. We can combine the two building blocks, creating a type that is both private and limited.

Throughout this discussion we will assume that these designs are based on *Abstract Data Types* (page 11), hence we assume the use of private types. That's a general, initial design assumption but in this case private types are required by the two idiom implementations. The types are not necessarily limited as well, but in one situation they will be limited too. But in both implementations the primary types will be private types.

7.2.1 Compile-Time Legality

We can combine the private type and limited type building blocks with another, known as *unknown discriminants*, to force explicit object initialization by clients, to control the form of explicit initialization, and, when required, to control client object creation itself. Limited and private types are fairly common building blocks, but *unknown discriminants* are less common so we will first explain them, and then show how to utilize the combinations for this idiom.

Discriminants are useful for our purpose because types with discriminants are *indefinite types* (under certain circumstances). Indefinite types do not allow object declarations without also specifying some sort of constraints for those objects. Unconstrained array types, such as **`String`**, are good examples. We cannot simply declare an object of type **`String`** without also specifying the array bounds, one way or another:

[24] https://en.wikipedia.org/wiki/Singleton_pattern

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Initialization_Demo is
   S1 : String (1 .. 11) := (others => ' ');
   S2 : String := "Hello World";
   S3 : String := S1;
begin
   Put_Line ('"' & S1 & '"');
   Put_Line ('"' & S2 & '"');
   Put_Line ('"' & S3 & '"');
end Initialization_Demo;
```

In the code above, `String` objects S1, S2, and S3 all have the same constraints: a lower bound of `Positive'First` and an upper bound of 11. S1 gives the bounds directly, whereas S2 and S3 take their constraints from their initial values. A function that returned a `String` value would suffice for the initial value too and would thus serve to specify the array bounds. There are other ways to specify a constraint as well, but we can ignore them in this idiom because the building blocks we'll use preclude them.

Types with discriminants are indefinite types unless the discriminants have default values. That fact will not apply in this idiom because of the characteristics of the building blocks. You will see why in a moment. The important idea is that we can leverage the object constraint requirements of indefinite types to force explicit initialization on declarations.

Discriminants come in two flavors. So-called *known* discriminants are the most common. These discriminants are known in the sense that they are compile-time visible to client code. Clients then have everything needed for declaring objects of the corresponding type. For example, here is the type declaration for a bounded stack ADT:

```
type Stack (Capacity : Positive) is private;
```

In the above, `Capacity` is the physical number of components in the array backing the bounded implementation. Clients can, therefore, have different objects of the type with different capacities:

```
Trays : Stack (Capacity => 10);
Operands : Stack (100);
```

The existence of `Capacity` is known to clients via the partial view, so the requirement for the constraint is visible and can be expressed.

In contrast, types may have *unknown discriminants* in the client's view. The syntax reflects their confidential nature:

```
type Foo (<>) is private;
```

The parentheses are required as usual, but the *box* symbol appears inside, instead of one or more discriminant declarations. The box symbol always indicates *not specified here* so in this case no discriminants are included in the view. There may or may not be discriminants in the full view, but client's don't have compile-time visibility to that information because the type is private.

Unknown discriminants can be specified for various kinds of types, not only private types. See the *Notes section* (page 52) for the full list. That said, combining them with private type declarations, or private type extension declarations, is the most common usage when composing abstraction definitions. For example:

```
package P is
   type Q (<>) is private;
private
   type Q is range 0 .. 100;
end P;
```

Clients of package P must use type Q as if Q requires discriminant constraints, even though clients don't have compile-time visibility to whatever constraints are actually required, if any. In the above, Q is just an integer type in the full view. No constraint is required to create objects of type Q, but clients cannot take advantage of that fact because they only have the partial view. Only the package private part, the package body, and child units have the visibility required to treat Q as an integer type.

Q might actually be completed as an indefinite type, but the constraint required need not be a discriminant constraint. In the following, objects of type Q require an array bounds constraint:

```
package P is
   type Q (<>) is private;
private
   type Q is array  (Positive range <>) of Integer;
end P;
```

Code with the full view must respect the index bounds requirement, but the semantics of the partial view remain the same.

As illustrated, the consequence of combining indefinite types with private types is that, when declaring objects, clients must express a constraint but cannot do so directly. The constraints must instead be provided by the initial value. Hence, for these types, the initial value is now a requirement that the compiler enforces on client object declarations.

But because the type is private, the initial value cannot be specified by a literal. Instead, the initial value must be either an existing object of the type, or the result of a call to a function that returns an object of the type.

Consider the following:

```
package P is
   type Q (<>) is private;
   function F return Q;
private
   type Q is range 0 .. 100;
end P;

package body P is
   function F return Q is (42);
   --  since that is the answer to everything...
end P;

with P;
procedure Demo is
   Obj1 : P.Q;  --  not legal, requires initial value for constraint
   Obj2 : P.Q := 42;  --  not legal, per client's partial view
   Obj3 : P.Q := P.F;
   Obj4 : P.Q := Obj3;
begin
   null;
end Demo;
```

The declaration for Obj1 is illegal because no constraint is provided. Because P.Q is also private, the declaration of Obj2 is illegal because clients don't have the full view supporting integer usage. But the initial value can be provided by a function result (Obj3), thereby also specifying the required constraint. And an existing object can be used to give the constraints to other objects during their declarations (Obj4). Explicit client initialization in these two ways is required by the compiler for indefinite private types.

But as illustrated by the spin-lock example, initialization by an existing object is not always appropriate. We can restrict the initial value to a function call result by making the type limited as well as private and indefinite. Then only constructor functions can be used legally for the initial values, and the compiler will require them to be called during object declarations (e.g., Obj3 above). That's what we'd do for the spin-lock type. We'd make the type limited in the completion too, to prevent copying in any form, including the function result. (The function result would then be built in place instead of copied.)

To recap, the primary purpose of the idiom, for a given type, is to ensure that clients initialize objects of that type as part of the object declarations. In this first implementation we meet the requirement by composing the type via building blocks that:

1. require a constraint to be given when declaring any object of the type, and
2. require an initial value to give that constraint, and
3. allow only objects and function call results as the initial values, and
4. when necessary, allow only function call results to be used for the initial values.

The compiler will reject declarations that do not adhere to these rules. Explicit initialization in the client code is thus guaranteed.

For a concrete example, consider a closed loop process controller, specifically a proportional–integral–derivative (PID) controller[25]. A PID controller examines the difference between an intended value, such as the desired speed of your automobile, and the current value (the actual speed). In response to that difference the controller increases or decreases the throttle setting. This measurement and resulting control output response happens iteratively at some rate. This is a sophisticated ADT, and explaining how a PID controller actually works is beyond the scope of this document. There are numerous web sites available that describe them in detail. What you should know for our purpose is that they are used to control physical processes, such as your car's cruise control system, that affect our lives directly. Ensuring proper initialization is part of ensuring correct use.

The PID controller must be explicitly initialized because there is no default initial state that would allow subsequent safe use. Only a partial meaningful state can be defined by default. Specifically, a PID controller can be enabled and disabled by the user (the external process control engineer) at arbitrary times. We can define default initialization such that the objects are initially in the disabled state. When disabled, the output computation actually affects nothing, so starting from that state would be safe. However, there is nothing to prevent the user from enabling the controller object without first configuring it. Configuring the various parameters is essential for safe and predictable behavior.

To address that problem, we could add operation preconditions requiring the object to be in some *configured* state, but that isn't always appropriate. Such a precondition would just raise an exception, which isn't in the use-cases. (Statically proving prior configuration in the client code would be a viable alternative, but that's also beyond the scope of this document.)

Therefore, default initialization doesn't really suffice for this ADT. We need to force initialization (configuration) during object creation so that enabling the ADT output will always be safe. This idiom implementation does exactly that.

The following is a cut-down version of the package declaration using this idiom implementation, with some operations and record components elided for the sake of simplicity. In the full version the unit is a generic package for the sake of not hard-coding the floating point types. We use a regular package and type **Float** here for convenience. The full version is here:

- AdaCore/Robotics_with_Ada/src/control_systems (GitHub)[26]

```
package Process_Control is

   type PID_Controller (<>) is tagged limited private;
```

(continues on next page)

[25] https://en.wikipedia.org/wiki/Proportional–integral–derivative_controller
[26] https://github.com/AdaCore/Robotics_with_Ada/blob/master/src/control_systems/

(continued from previous page)

```
   type Bounds is record
      Min, Max : Float;
   end record with
     Predicate => Min < Max;

   type Controller_Directions is (Direct, Reversed);

   type Millisecond_Units is mod 2**32;

   subtype Positive_Milliseconds is
     Millisecond_Units range 1 .. Millisecond_Units'Last;

   function Configured_Controller
     (Proportional_Gain : Float;
      Integral_Gain     : Float;
      Derivative_Gain   : Float;
      Invocation_Period : Positive_Milliseconds;
      Output_Limits     : Bounds;
      Direction         : Controller_Directions := Direct)
   return PID_Controller;

   procedure Enable
     (This             : in out PID_Controller;
      Process_Variable : Float;   --  current input value from the process
      Control_Variable : Float);  --  current output value

   procedure Disable (This : in out PID_Controller);

   procedure Compute_Output
     (This             : in out PID_Controller;
      Process_Variable : Float;          --  the input, Measured Value/Variable
      Setpoint         : Float;
      Control_Variable : in out Float);  --  the output, Manipulated Variable

   --  ...

   function Enabled (This : PID_Controller) return Boolean;

private

   type PID_Controller is tagged limited record
      --  ...
      Enabled : Boolean;
   end record;

end Process_Control;
```

As you can see, the PID controller type is indefinite limited private:

```
type PID_Controller (<>) is tagged limited private;
```

It is also tagged, primarily for the sake of the distinguished receiver call syntax. We don't really expect type extensions in this specific ADT, although nothing prevents them.

Therefore, the language requires an initial value when creating objects of the type, and because the type is limited, a function must be used for that initial value. The compiler will not compile the code containing the declaration otherwise. The only constructor function provided is `Configured_Controller` so it is guaranteed to be called. (A later child package could add another *constructor function* (page 27). For that matter, we probably should have declared this one in a child package. In any case one of them is guaranteed to be called.)

Here is an example declaration taken from the steering control module for an RC car written in Ada[27].

The PID controller, named `Steering_Computer`, is declared within the body of a task `Servo` that controls a motor, `Steering_Motor`, in response to requested directions from the remote control. `Steering_Motor` is an instance of an ADT named `Basic_Motors`, and is declared elsewhere. The `Servo` task is declared within the body of a package that contains various values referenced within the task, such as the various PID gain parameters, that are not shown.

```
task body Servo is
   Next_Release       : Time;
   Target_Angle       : Float;
   Current_Angle      : Float := 0.0;
   --  zero for call to Steering_Computer.Enable
   Steering_Power     : Float := 0.0;
   --  zero for call to Steering_Computer.Enable
   Motor_Power        : NXT.Motors.Power_Level;
   Rotation_Direction : NXT.Motors.Directions;
   Steering_Offset    : Float;
   Steering_Computer  : PID_Controller :=
     Configured_Controller
       (Proportional_Gain => Kp,
        Integral_Gain     => Ki,
        Derivative_Gain   => Kd,
        Invocation_Period => System_Configuration.Steering_Control_Period,
        Output_Limits     => Power_Level_Limits,
        Direction         => Closed_Loop.Direct);
begin
   Global_Initialization.Critical_Instant.Wait (Epoch => Next_Release);
   Initialize_Steering_Mechanism (Steering_Offset);
   Steering_Computer.Enable (Process_Variable => Current_Angle,
                             Control_Variable => Steering_Power);
   loop
      Current_Angle := Current_Motor_Angle (Steering_Motor) -
                       Steering_Offset;
      Target_Angle := Float (Remote_Control.Requested_Steering_Angle);
      Limit (Target_Angle, -Steering_Offset, +Steering_Offset);
      Steering_Computer.Compute_Output
        (Process_Variable => Current_Angle,
         Setpoint         => Target_Angle,
         Control_Variable => Steering_Power);
      Convert_To_Motor_Values (Steering_Power,
                               Motor_Power,
                               Rotation_Direction);
      Steering_Motor.Engage (Rotation_Direction, Motor_Power);

      Next_Release := Next_Release + Period;
      delay until Next_Release;
   end loop;
end Servo;
```

Because `Steering_Computer` must be declared before it can be passed as a parameter, the call to configure the object's state necessarily precedes any other operation (e.g., Enable).

7.2.2 Run-Time Checks

Ada 2022 adds another building block, `Default_Initial_Condition` (DIC), that can be used as an alternative to the unknown discriminants used above. We must still have a private type or private type extension, and the type may or may not be limited, but unknown discriminants will not be involved. The compiler would not allow the combination, in fact.

[27] https://blog.adacore.com/making-an-rc-car-with-ada-and-spark

DIC is an aspect applied to a private type or private extension declaration. Developers use it to specify a developer-defined Boolean condition that will be true at run-time after the default initialization of an object of the type. Specifically, if `Default_Initial_Condition` is specified for a type, a run-time check is emitted for each object declaration of that type that uses default initialization. The check consists of the evaluation of the DIC expression. The exception `Assertion_Error` is raised if the check fails. You can think of this aspect as specifying the effects of default initialization for the type, with a verification at run-time when needed. No check is emitted for those declarations that use explicit initialization.

For example, the following is a partial definition of a `Stack` ADT. It is only a partial definition primarily because Pop is not provided, but other operations would be included as well. Moreover, a fully realistic version would be a generic package. We have used a subtype named `Element` as a substitute for the generic formal type what would have had that name. Note that there is a `Default_Initial_Condition` aspect specifying that any object of type `Stack` is initially empty as a result of default initialization. The *argument* to the function call is the corresponding type name, representing the current instance object, thus any object of the type.

```ada
package Bounded_Stacks is

   subtype Element is Integer;
   --  arbitrary substitute for generic formal type

   type Stack (Capacity : Positive) is limited private with
     Default_Initial_Condition => Empty (Stack);

   procedure Push (This : in out Stack; Value : Element) with
     Pre  => not Full (This),
     Post => not Empty (This);

   function Full (This : Stack) return Boolean;

   function Empty (This : Stack) return Boolean;

private

   type Contents is array (Positive range <>) of Element;

   type Stack (Capacity : Positive) is limited record
      Content : Contents (1 .. Capacity);
      Top     : Natural :=0;
   end record;

   function Full (This : Stack) return Boolean is
     (This.Top = This.Capacity);

   function Empty (This : Stack) return Boolean is
     (This.Top = 0);

end Bounded_Stacks;

package body Bounded_Stacks is

   procedure Push (This : in out Stack; Value : Element) is
   begin
      This.Top := This.Top + 1;
      This.Content (This.Top) := Value;
   end Push;

end Bounded_Stacks;

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

(continues on next page)

(continued from previous page)

```
with Bounded_Stacks; use Bounded_Stacks;

procedure Demo is
   S : Stack (Capacity => 10);
begin
   Push (S, 42);
   Put_Line ("Done");
end Demo;
```

The function `Empty` returns **True** when Top is zero, and zero is assigned to Top during default initialization. Consequently, `Assertion_Error` is not raised when `Demo` executes because the object S was indeed default initialized to the empty state.

We said that when DIC is applied to a type, the run-time check is emitted for all object declarations of that type that rely on default initialization. But suppose the type does not define any default initialization. We can detect these uninitialized objects at run-time if we set the DIC Boolean expression to indicate that there is no default initialization defined for this type. The checks will then fail for those objects. That's the second implementation approach to the initialization requirement.

Specifically, we can express the lack of default initialization by a DIC condition that is hard-coded to the literal **False**. The evaluation during the check will then necessarily fail, raising `Assertion_Error`. Hence, for this type, explicit initialization is guaranteed in a program that does not raise `Assertion_Error` for this cause.

The following is an example of the DIC set to **False**:

```
package P is

   type Q is limited private with
     Default_Initial_Condition => False;

   function F return Q;

private

   type Q is range -1 .. 100;

end P;

package body P is

   function F return Q is (42);

end P;

with Ada.Text_IO;  use Ada.Text_IO;

with P; use P;

procedure Main is
   Obj1 : constant Q := F;
   Obj2 : Q;  --  triggers Assertion_Error
begin
   Put_Line (Obj1'Image);
   Put_Line (Obj2'Image);
   Put_Line ("Done");
end Main;
```

In the above, `Assertion_Error` is raised by the elaboration of `Obj2` because the DIC check necessarily fails. There is no check on the declaration of `Obj1` because it is initialized, explicitly.

To recap, we can ensure initialization for objects of the type by detecting, during elaboration at run-time, any objects not explicitly initialized.

This approach is sufficient because when elaboration of an object declaration raises an exception, no use of that object is possible. That's guaranteed because the frame containing that declarative part is immediately abandoned and the exception is propagated up to the previous level. A local handler never can apply. But even if there is a matching handler in the previous level, there's really nothing much to be done. Re-entering the frame containing the declaration will raise the exception all over again, necessarily. Thus the code will have to be changed and recompiled, meeting the goal of the idiom.

We can illustrate this assurance using `Storage_Error`. Consider the following program, in which the main procedure calls an inner procedure P:

```ada
with Text_IO; use Text_IO;

procedure Main is

   procedure P (Output : out Float) is
      N : array (Positive) of Float; -- Storage_Error is likely
   begin
      Put_Line ("P's body assigns N's components and uses them");
      -- The following indexes and component values are arbitrary
      -- and used purely for illustration...
      N := (others => 0.0);
      -- other computations and assignments to N ...
      Output := N (5);
   exception
      when Storage_Error =>
         Output := N (1);
   end P;

   X : Float;

begin
   P (X);
   Put_Line (X'Image);
   Put_Line ("Done");
exception
   when Storage_Error =>
      Put_Line ("Main completes abnormally");
end Main;
```

When `Main` calls P, the elaboration of the declarative part of P almost certainly fails because there is insufficient storage to allocate to the object `P.N`, hence `Storage_Error` is raised. (If your machine can handle the above, congratulations.) Even though procedure P has a handler specifically for `Storage_Error`, that handler never applies because the declarative part is immediately abandoned. Instead, the exception is raised in the caller, where it can be caught. This behavior is essential to ensure that problematic objects are not referenced in the local handlers. In the above, the handler in P for `Storage_Error` references the object `P.N` to assign the `P.Output` parameter. If that assignment could happen — again, it cannot — what would it mean, functionally? No one knows.

Handling `Storage_Error` is a little tricky anyway. Does the OS give the program a chance to execute a handler? If so, is there sufficient storage remaining to execute the exception handler's statements? In any case you can see the problem that the declaration failure semantics preclude.

Therefore, although the DIC approach is not enforced at compile-time, it is nevertheless sufficient to ensure no uninitialized object of the type can be used.

7.2.3 Preventing Object Creation by Clients

The other idiom requirement is the ability to control object creation itself. The implementation is trivially achieved using an indefinite limited private type: we can prevent client object creation simply by not providing any constructor functions. Doing so removes any means for initializing objects of the type, and since the type is indefinite there is then no way for clients to declare objects at all. The compiler again enforces this implementation.

For a concrete example, we can apply the Singleton design pattern to represent the *time stamp counter* (TSC[28]) provided by x86 architectures. The TSC is a 64-bit hardware register incremented once per clock cycle, starting from zero at power-up. We can use it to make a timestamp abstraction. As explained by Wikipedia page[29], some care is required when using the register for that purpose on modern hardware, but it will suffice to illustrate the idiom implementation. Note that the Singleton pattern is itself somewhat controversial in the OOP community, but that's beyond the scope of this document.

Why use the Singleton pattern in this case? Ordinarily, clients of some ADT will reasonably expect that the states of distinct objects are independent of each other. When using an ADT to represent a single piece of hardware, however, this presumption of independence will not hold because the device is shared by all the objects, unavoidably. The singleton idiom prevents the resulting problems by precluding the existence of multiple objects in the first place.

In this specific case, the time stamp counter hardware is read-only, so the lack of independence is not an issue. Multiple objects would not be a problem. But many devices are not read-only, so the singleton pattern is worth knowing.

First we'll define a singleton ADT representing the TSC register itself, then we will extend that type to add convenience operations for measuring elapsed times. We'll use the design approach of indefinite limited private types without any constructor functions in order to ensure clients cannot create objects of the type. The type will also be tagged for the sake of allowing type extensions. Adding the tagged characteristic doesn't change anything regarding the idiom implementation.

```ada
with Interfaces;

package Timestamp is

   type Cycle_Counter (<>) is tagged limited private;

   type Cycle_Counter_Reference is access all Cycle_Counter;

   function Counter return not null Cycle_Counter_Reference;

   type Cycle_Count is new Interfaces.Unsigned_64;

   function Sample (This : not null access Cycle_Counter) return Cycle_Count;

private

   type Cycle_Counter is tagged limited null record;

   function Read_TimeStamp_Counter return Cycle_Count with
     Import,
     Convention    => Intrinsic,
     External_Name => "__rdtsc",
     Inline;
   --  This gcc builtin issues the machine instruction to read the time-stamp
   --  counter, i.e., RDTSC, which returns a 64-bit count of the number of
   --  system clock cycles since power-up.
```

(continues on next page)

[28] https://en.wikipedia.org/wiki/Time_Stamp_Counter
[29] https://en.wikipedia.org/wiki/Time_Stamp_Counter

(continued from previous page)

```
   function Sample (This : not null access Cycle_Counter)
                    return Cycle_Count is
     (Read_TimeStamp_Counter);
     --  The formal parameter This is not referenced

end Timestamp;
```

Note also that the primitive function named Counter is not a constructor — it doesn't return an object of the Cycle_Counter type. As such, it cannot be used as an initial value for a Cycle_Counter object declaration. Clients cannot, therefore, create their own objects of type Cycle_Counter.

Instead, function Counter returns an access value designating an object of the type. Because clients cannot declare objects themselves the function is the only way to get an object, albeit indirectly. Therefore, the function can control how many objects are created. As you will see, the function only creates a single object of the type.

The type Cycle_Counter is completed as a null record because the state is maintained in the hardware register we're reading.

The function Sample reads the timestamp counter register by calling the Read_TimeStamp_Counter function. That second function accesses the TSC register by executing an assembly language instruction dedicated to that purpose. We could have Sample issue that instruction instead, without declaring a separate function, but there is no run-time cost (due to the inlining) and separating them emphasizes that one is a member of the API and the other is an implementation artifact. Note that Sample does not actually reference the formal parameter This. The parameter exists just to make Sample a primitive function. Assuming we don't have a use-clause for Timestamp, to call Sample we could say:

```
TimeStamp.Counter.Sample
```

for example:

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

procedure Demo_TimeStamp is
begin
   for K in 1 .. 10 loop
      Put_Line (Timestamp.Counter.Sample'Image);
   end loop;
end Demo_TimeStamp;
```

The above calls the Timestamp.Counter function and then implicitly dereferences the resulting access value to call the Sample function using the distinguished receiver syntax. The resulting number is then converted to a **String** value and output to Standard_Output.

We could have instead used positional call notation for the call to Sample:

```
Timestamp.Sample (Timestamp.Counter)
```

In that case we need the package name on the references, or we'd add a use-clause.

The package body is shown below. Only the function Counter has a body because Sample is completed in the package declaration's private part and Read_TimeStamp_Counter is an imported intrinsic, i.e., without a body.

```
package body Timestamp is
```

(continues on next page)

(continued from previous page)

```
   The_Instance : Cycle_Counter_Reference;

   -------------
   -- Counter --
   -------------

   function Counter return not null Cycle_Counter_Reference is
   begin
      if The_Instance = null then
         The_Instance := new Cycle_Counter;
      end if;
      return The_Instance;
   end Counter;

end Timestamp;
```

Function Counter creates the single object that this singleton implementation creates. It does so by lazily allocating an object dynamically. If Counter is never called (because some subclass is used instead) then no object of type Cycle_Counter is created. At most one Cycle_Counter object is ever created.

We could instead declare The_Instance as a Cycle_Counter object in the package body, mark it as aliased, and return a corresponding access value designating it. But when objects are large, declaring one that might never be used is wasteful. The indirection avoids that wasted storage at the cost of an access object, which is small. On the other hand, now the heap is involved.

Note that we could have declared The_Instance in the private part of the package declaration. Type extensions in child packages could then use it, if needed. Presumably we'd make The_Instance be of some access to class-wide type so that extensions could use it to allocate objects of their specific type, otherwise extensions in child packages would have no need for it. But that only saves the storage for an access object in the child packages, so we leave the declaration in the parent package body. See the *Programming by Extension idiom* (page 23) for a discussion of whether to declare an entity in the package private part or the package body.

Next, we declare a type extension in a child package. The child package body will contain its own object named The_Instance, returning an access value designating the specific extension type. The client API in the package declaration follows that of the parent type Cycle_Counter, but with additional primitives for working with samples.

```
package Timestamp.Sampling is

   type Timestamp_Sampler is new Cycle_Counter with private;

   type Timestamp_Sampler_Reference is access all Timestamp_Sampler;

   function Counter return not null Timestamp_Sampler_Reference with Inline;
   --  returns an access value designating the single instance

   procedure Take_First_Sample  (This : not null access Timestamp_Sampler)
     with Inline;
   procedure Take_Second_Sample (This : not null access Timestamp_Sampler)
     with Inline;

   function First_Sample  (This : not null access Timestamp_Sampler)
                          return Cycle_Count;
   function Second_Sample (This : not null access Timestamp_Sampler)
                          return Cycle_Count;
   function Elapsed       (This : not null access Timestamp_Sampler)
                          return Cycle_Count;
```

(continues on next page)

(continued from previous page)

```
private

   type Timestamp_Sampler is new Cycle_Counter with record
      First  : Cycle_Count := 0;
      Second : Cycle_Count := 0;
   end record;

end Timestamp.Sampling;

package body Timestamp.Sampling is

   The_Instance : Timestamp_Sampler_Reference;

   -------------
   -- Counter --
   -------------

   function Counter return not null Timestamp_Sampler_Reference is
   begin
      if The_Instance = null then
         The_Instance := new Timestamp_Sampler;
      end if;
      return The_Instance;
   end Counter;

   -----------------------
   -- Take_First_Sample --
   -----------------------

   procedure Take_First_Sample (This : not null access Timestamp_Sampler) is
   begin
      This.First := Sample (This);
   end Take_First_Sample;

   ------------------------
   -- Take_Second_Sample --
   ------------------------

   procedure Take_Second_Sample (This : not null access Timestamp_Sampler) is
   begin
      This.Second := Sample (This);
   end Take_Second_Sample;

   ------------------
   -- First_Sample --
   ------------------

   function First_Sample (This : not null access Timestamp_Sampler)
                       return Cycle_Count is
     (This.First);

   -------------------
   -- Second_Sample --
   -------------------

   function Second_Sample (This : not null access Timestamp_Sampler)
                        return Cycle_Count is
     (This.Second);

   -------------
```

(continues on next page)

(continued from previous page)

```
   -- Elapsed --
   -------------

   function Elapsed (This : not null access Timestamp_Sampler)
                     return Cycle_Count is
     (This.Second - This.First + 1);

end Timestamp.Sampling;
```

The inherited Sample function is called in the two procedures that take the two samples of the timestamp register. The formal parameter This is passed to the calls, but as mentioned earlier the argument is not referenced within Sample. All the formal parameter does is participate in dispatching the calls to Sample, in this case meaning that the inherited version of Sample is the one called because This is of the extended type.

But Sample is not overridden in this child package, therefore effectively we are calling the parent version. Is Sample ever likely to be overridden? Arguably not, because it is so directly dependent on the underlying hardware. Of course, some future type extension may override Sample for some unforeseen reason — that's the point of making it possible, after all. Presumably the overridden version would also call the parent version, otherwise the timestamp counter would not be accessed. Because we can't say for certain that it will never need to be overridden, we have made Sample a primitive function, thus overridable.

Suppose we came to the opposite conclusion, that Timestamp.Sample would never need to be overridden. In that case we have some options worth exploring.

Clearly function Sample must be part of the client API, but that doesn't force it to be a primitive function.

We could have declared Sample in Timestamp as a visible non-primitive operation, i.e., without a formal parameter or function result of the ADT type:

```
function Sample return Cycle_Count with Inline;
```

As a non-primitive function it would be neither inherited nor overridable. But we'd still be able to call it in client code.

Yet, as a non-primitive, this version looks like an implementation artifact, hence out of place as part of the visible client API. It isn't illegal by any means, it just *looks* wrong.

Furthermore, if we are going to make Sample a non-primitive function, why not remove it and replace it with the other non-primitive function Read_Timestamp_Counter? Or make the body of Sample call the imported intrinsic, and do away with function Read_Timestamp_Counter? There is no clear winner here.

An attractive alternative would be to make Sample be a class-wide operation. To do so, we make the formal parameter class-wide instead of removing it:

```
function Sample (This : not null access Cycle_Counter'Class)
                return Cycle_Count
  with Inline;
```

In the version above, the formal parameter type is now (anonymous) access to Cycle_Counter'Class, i.e., class-wide, so in this version Sample can be passed a value designating an object of type Cycle_Counter or any type derived from it. We don't want to have a null access value passed so we add that to the parameter specification.

In this version the function is again not a primitive operation and so is neither inherited nor overridable, but because it mentions type Cycle_Counter it looks like a reasonable part of an Abstract Data Type. As it happens this version of Sample also doesn't actually reference the formal parameter, so it is somewhat unusual. Ordinarily in the body we'd expect the class-wide formal to be used in dynamic dispatching calls to primitive operations, but that's not required by the language.

Ultimately whether to make `Sample` a primitive operation is a judgment call. We don't know that `Sample` will never need to be overridden so we declare it as a primitive op.

With all that said, here is an example program using the child type. Because the timestamp register is updated once per clock cycle, if we know the system clock frequency we can use the counter to measure elapsed time. In the demo below we measure the accuracy of the delay statement by delaying for a known time, with samples taken before and after the delay statement. We can then compare the known delay time to the measured elapsed time, printing the difference.

Note the constant `Machine_Cycles_Per_Second`. Before you run the demo you will likely need to change it in the source code to your machine's clock frequency.

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

procedure Demo_Sampling_Cycle_Counter is

   Delay_Interval : constant Duration := 1.0;
   --  arbitrary, change if desired
   Elapsed_Time   : Duration;

   GHz : constant := 1_000_000_000;

   Machine_Cycles_Per_Second : constant := 1.9 * GHz;
   --  This is the system clock rate on the machine running this executable.
   --  It corresponds to the rate at which the time stamp counter hardware is
   --  incremented. Change it according to your target.

   use type Timestamp.Cycle_Count;  --  for "<"
begin
   Put_Line ("Using" & Machine_Cycles_Per_Second'Image
             & " Hertz for system clock");

   Put_Line ("Delaying for" & Delay_Interval'Image & " second(s) ...");

   Counter.Take_First_Sample;
   delay Delay_Interval;
   Counter.Take_Second_Sample;

   Put_Line ("First sample            :" & Counter.First_Sample'Image);
   Put_Line ("Second sample           :" & Counter.Second_Sample'Image);

   if Counter.Second_Sample < Counter.First_Sample then
      Put_Line ("RDTSC counter wrapped around!?");
      return;
   end if;

   Elapsed_Time := Duration (Elapsed (Counter)) / Machine_Cycles_Per_Second;

   Put_Line ("Elapsed count           :" & Elapsed (Counter)'Image);
   Put_Line ("Specified delay interval:" & Delay_Interval'Image);
   Put_Line ("Measured delay interval :" & Elapsed_Time'Image);
end Demo_Sampling_Cycle_Counter;
```

In the above, `Delay_Interval` is set to 1.0 so the program will delay for 1 second, with samples taken from the TSC before and after. Delay statement semantics are such that at least the amount of time requested is delayed, so some value slightly greater than 1 second is expected. There will be overhead too, so an elapsed time slightly larger than requested should be seen. The value of `Delay_Interval` is arbitrary, change it to whatever you like.

If you have set the Machine_Cycles_Per_Second properly but still get elapsed measurement values that are much larger than expected or don't make sense at all, it may be that your machine does not support using the TSC[30] this way reliably.

7.3 Pros

Ensuring explicit initialization is easily achieved. The abstraction should likely be a private type anyway, and the syntax for the required additional building blocks is light: all are just additional decorations on the declaration of the private type or private extension. The compiler does the rest, either at compile-time itself or via a generated check verified at run-time.

Likewise, ensuring that only the implementation can create objects of a type is straightforward. We take the same approach for ensuring initialization via function calls in object declarations, but then don't provide any such functions. Only the implementation will have the required visibility to create objects of the type, and can limit that number of objects to one (or any other number). Client access to this hidden object must be indirect, but that is not a heavy burden.

7.4 Cons

None.

7.5 Relationship With Other Idioms

The *Abstract Data Type* (page 11) is assumed, in the form of a private type.

7.6 Notes

Only certain types can have unknown discriminants. For completeness here is the list:

- A private type
- A private extension
- An incomplete type
- A generic formal private type
- A generic formal private type extension
- A generic formal derived type
- Descendants of the above

The types above will either have a corresponding completion or a generic actual parameter to either define the discriminants or specify that there are none.

As we mentioned, Default_Initial_Condition is new in Ada 2022. The other implementation, based on indefinite private types, is supported by Ada 2022 but also by earlier versions of the language. However, if the type is also limited, Ada 2005 is the earliest version allowing that implementation. Prior to that version an object of a limited type could not be initialized in the object's declaration.

[30] https://en.wikipedia.org/wiki/Time_Stamp_Counter

CHAPTER

EIGHT

TYPE PUNNING

8.1 Motivation

When declaring an object, the type chosen by the developer is presumably one that meets the operational requirements. Sometimes, however, the chosen type is not sufficient for clients of that object. Normally that situation would indicate a design error, but not necessarily.

Consider a device driver that receives external data, such as a network or serial I/O driver. Typically the driver presents incoming data to clients as arrays of raw bytes. That's how the data enter the receiving machine, so that's the most natural representation at the lowest level of the device driver code. However, that representation is not likely sufficient for the higher-level clients. These clients are not necessarily at the application level, they may be lower than that, but they are clients because they use the data presented.

For example, the next level of the network processing code must interpret the byte arrays in order to implement the network protocol. Interpreting the data requires reading headers that are logically contained within slices of those bytes. These headers are naturally represented as record types containing multiple fields. How, then, can the developer apply such record types? An array of bytes contains bytes, not headers.

Stated generally, on occasion a developer needs to manipulate or access the value of a given object in a manner not supported by the object's type. The issue is that the compiler will enforce the type model defined by the language, to some degree of rigor, potentially resulting in the rejection of the alternative access and manipulation code. In such cases the developer must circumvent this enforcement. That's the purpose of this idiom.

A common circumvention technique, across programming languages, is to apply multiple distinct types to a single given object. Doing so makes available additional operations or accesses not provided by the type used to declare the object in the first place. The technique is known as type punning[31] in the programming community because different types are used for the same object in much the same way that a pun in natural languages uses different meanings for words that sound the same when spoken.

Ada is accurately described as "a language with inherently reliable features, only compromised via explicit escape hatches having well-defined and portable semantics when used appropriately."[35] The foundation for this reliability is static named typing with rigorous enforcement by the compiler.

Specifically, the Ada compiler checks that the operations and values applied to an object are consistent with the one type specified when the object is declared. Any usage not consistent with that type is rejected by the compiler. Similarly, the Ada compiler also checks that any type conversions involve only those types that the language defines as meaningfully convertible.

[31] https://en.wikipedia.org/wiki/Type_punning

[35] S. Tucker Taft. *Post in Internet Relay Chat on Comp.Lang.Ada channel*. https://groups.google.com/g/comp.lang.ada/c/9WXgvv8Xjuw/m/JMyo9_P7nxAJ, 1993.

By design, this strong typing model does not lend itself to circumvention (thankfully). That's the point of Ada's *escape hatches* — they provide standard ways to circumvent these and other checks. To maintain the integrity of the type model not many escape hatches exist. The most commonly used of these, *unchecked conversion*, allows type conversion between arbitrary types. Unchecked conversions remain explicit, but the compiler does not limit them to the types defined as reasonable by the language.

8.2 Implementation(s)

There are two common approaches for expressing type punning in Ada. We show both in the following subsections. The purpose in both approaches is to apply a different type, thereby making available a different type-specific view of the storage.

8.2.1 Overlays

The first approach applies an alternative type to an existing object by declaring another object at the same location in memory but with a different type. Given an existing object, the developer:

1. declares (or reuses) an alternative type that provides the required operations and values not provided by the existing object's type, and
2. declares another object of this alternative type, and
3. as part of the new object's declaration, specifies that this new object shares some or all of the storage occupied by the existing object.

The result is a partial or complete storage overlay. Because there are now multiple distinct types involved, there are multiple views of that shared storage, each view providing different operations and values. Thus, the shared storage can be legally manipulated in distinct ways. As usual, the Ada compiler verifies that the usage corresponds to the type view presented by the object name referenced.

For example, let's say that we have an existing object, and that a signed integer is most appropriate for its type. On some occasions let's also say we need to access individual bits within that existing object. Signed integer types don't support bit-level operations, unlike unsigned integers, but we've said that a signed type is the best fit for the bulk of the usage.

One of the ways to enable bit access, then, is to apply another type that does have bit-level operations. We could overlay the existing object with an unsigned integer type of the same size, but let's take a different approach for the sake of illustration. Instead, we'll declare an array type with components that can be represented as a single bit. The length of the array type will reflect the number of bits used by the signed integer type so that the entire object will be overlaid. (A record type would work too, with a component allocated to each bit.)

The type `Boolean` will suffice for the array component type as long as we force the single-bit representation. `Boolean` array components are likely to be represented as individual bytes otherwise. Alternatively, we could just make up an integer type with range `0 .. 1` but that seems unnecessary, unless numeric values would make the code clearer. (Maybe the requirements specify ones and zeros for the bit values.) In any case we'll need to force single bits for the components.

Assuming values of type `Integer` require exactly 32 bits, the following code illustrates the approach:

```ada
type Bits32 is array (0 .. 31) of Boolean with Component_Size => 1;

X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

In the above, we use X'Address to query the starting address of object X. We use that address to specify the location of the overlay object Y. As a result, X and Y start at the same address.

We marked X as explicitly **aliased** because **Integer** is not a by-reference type. The **Address** attribute is not required to provide a useful result otherwise. (Maybe the compiler would have put X into a register, for example.) The compiler, seeing 'Address applied, would probably do the right thing anyway, but this makes it certain.

Here is a simple main program that illustrates the approach.

Listing 1: main.adb

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

procedure Main is

   X : aliased Integer := 42;

   type Bits32 is array (0 .. 31) of Boolean
     with Component_Size => 1;

   Y : Bits32 with Address => X'Address;

begin
   X := Integer'First;
   Put_Line (X'Image);
   for Bit in Bits32'Range loop
      Put (if Y (Bit) then '1' else '0');
   end loop;
   New_Line;
end Main;
```

Object Y starts at the same address as X and has the same size, so all references to X and Y refer to the same storage. The source code can manipulate that memory as either a signed integer or as an array of bits, including individual bit access, using the two object names. The compiler will ensure that every reference via X is compatible with the integer view, and every reference via Y is compatible with the array view.

In the above example, we've ignored the endianess issue. If you wanted to change the sign bit, for example, or display the bits in the "correct" order, you'd need to handle that detail.

This expression of type-punning does not use an escape hatch but does achieve the effect. (We don't include address clauses as an escape hatch because address clauses aren't dedicated to overlaying multiple objects of different types. On the other hand, even one Ada object with an address specified overlays that object's view with the machine storage view of that address...)

8.2.2 Unchecked Conversions on Address Values

The common implementation of type punning, across multiple languages, involves converting the address of a given object into a pointer designating the alternative type to be applied. Dereferencing the resulting pointer provides a different compile-time view of the object. Thus, the operations defined by the alternative type are made applicable to the object.

Expressing this approach in Ada requires unchecked conversion because, in Ada, address values are semantically distinct from pointer values (*access values*). An access value might be represented by an address value, but because architectures vary, that implementation in not guaranteed. Therefore, the language does not define checked conversions between addresses and access values. We need the escape hatch.

Unchecked conversion requires instantiation of the generic function Ada. Unchecked_Conversion, including a context clause for that unit, making it a relatively heavy mechanism. This heaviness is intentional, and the with_clause at the top of the client unit makes it noticeable. Although ubiquitous use strongly suggests abuse of the type model, in this case unchecked conversion is necessary. Nevertheless, we'd hide its use within the body of some visible unit.

Let's start with a simple example. There is an accelerometer that provides three signed 16-bit acceleration values, one for each axis. Accelerations can be both positive and negative so the signed type is appropriate. These values are read from the device as two unsigned bytes. The two bytes are read individually so two reads are required per acceleration value. (This is an actual, real-world device.) Because the acceleration values are signed 16-bit integers, we need to convert two unsigned bytes into a single signed 16-bit quantity. We can use type punning, based on a pointer designating the 16-bit signed type, to achieve that effect. There are certainly other ways to do this, but we're starting with something simple for the sake of illustrating this idiom.

In the following fragment, type Acceleration is a signed 16-bit integer type already declared elsewhere:

```
type Acceleration_Pointer is access all Acceleration
  with Storage_Size => 0;

function As_Acceleration_Pointer is new Ada.Unchecked_Conversion
  (Source => System.Address, Target => Acceleration_Pointer);
```

The access type is general, not pool-specific, but that is optional. We tell the compiler to reserve no storage for the access type because an allocation would be an error that we want the compiler to catch. Whether or not the compiler actually reserves storage for an individual access type is implementation-dependent, but this way we can be sure. In any case the compiler will reject any allocations.

Given this access type declaration we can then instantiate Ada.Unchecked_Conversion. The resulting function name is a matter of style but is appropriate because the function allows us to treat an address as a pointer to an Acceleration value. We aren't changing the address value, we're only providing another view of that value, which is why the function name is not To_Acceleration_Pointer.

The following is the device driver routine for getting the scaled accelerations from the device. The type Three_Axis_Accelerometer is the device driver *ADT* (page 11), and type Axes_Accelerations is a record type containing the three axis values. The procedure gets the raw acceleration values from the device and scales them per the current device sensitivity, returning all three in the mode-out record parameter.

```
procedure Get_Accelerations
  (This : Three_Axis_Accelerometer;
   Axes : out Axes_Accelerations)
is

   Buffer : array (0 .. 5) of UInt8 with Alignment => 2, Size => 48;
   Scaled : Float;

   type Acceleration_Pointer is access all Acceleration
     with Storage_Size => 0;

   function As_Acceleration_Pointer is new Ada.Unchecked_Conversion
     (Source => System.Address, Target => Acceleration_Pointer);

begin
   This.Loc_IO_Read (Buffer (0), OUT_X_L);
   This.Loc_IO_Read (Buffer (1), OUT_X_H);
   This.Loc_IO_Read (Buffer (2), OUT_Y_L);
```

(continues on next page)

(continued from previous page)

```
   This.Loc_IO_Read (Buffer (3), OUT_Y_H);
   This.Loc_IO_Read (Buffer (4), OUT_Z_L);
   This.Loc_IO_Read (Buffer (5), OUT_Z_H);

   Get_X : declare
      Raw : Acceleration renames
               As_Acceleration_Pointer (Buffer (0)'Address).all;
   begin
      Scaled := Float (Raw) * This.Sensitivity;
      Axes.X := Acceleration (Scaled);
   end Get_X;

   Get_Y : declare
      Raw : Acceleration renames
               As_Acceleration_Pointer (Buffer (2)'Address).all;
   begin
      Scaled := Float (Raw) * This.Sensitivity;
      Axes.Y := Acceleration (Scaled);
   end Get_Y;

   Get_Z : declare
      Raw : Acceleration renames
               As_Acceleration_Pointer (Buffer (4)'Address).all;
   begin
      Scaled := Float (Raw) * This.Sensitivity;
      Axes.Z := Acceleration (Scaled);
   end Get_Z;
end Get_Accelerations;
```

This procedure first reads the six bytes representing all three acceleration values into the array `Buffer`. Procedure `Loc_IO_Read` is defined by the driver *ADT* (page 11). The constants `OUT_n_L` and `OUT_n_H`, also defined by the driver, specify the low-order and high-order bytes requested for the given *n* axis. Then the declare blocks do the actual scaling and that's where the type punning is applied to the `Buffer` content.

In each block, the address of one of the bytes in the array is converted into an access value designating a two-byte `Acceleration` value. The X acceleration is first in the buffer, so the address of `Buffer (0)` is converted. Likewise, the address of `Buffer (2)` is converted for the Y axis value, and for the Z value, `Buffer (4)` is converted. (We could have said `Buffer'Address` instead of `Buffer (0)'Address`, they mean the same thing, but an explicit index seemed more clear, given the need for the other indexes.)

But we want the designated axis acceleration value, not the access value, so we also dereference the converted access value via `.all`, and rename the result for convenience. The name is Raw because the value needs to be scaled. Each dereference reads two bytes, i.e., the bytes at indexes 0 and 1, or 2 and 3, or 4 and 5.

That's the way the device driver is written currently, but it could be simpler. Clients always get all three accelerations via this procedure, so we could have used unchecked conversion to directly convert the entire array of six bytes into a value of the record type `Axes_Accelerations` containing the three 16-bit components. Type punning would not be required in that case. (The components would still need scaling, of course.)

Note that to get individual values we can't just convert a slice of the array because that's illegal: array slices cannot be converted. We'd need some other way to refer to a two-byte pair within the array. Type punning would be an appropriate approach.

For that matter we could use type punning but have the record type be the designated type returned from the address conversion, rather than a single axis value. Then we'd just convert `Buffer'Address` and not need to specify array indexes as all. This would be the same as converting the array to the record type, but with a level of indirection added.

For the network packet example, we want to apply record type views to arbitrary sequences within an array of raw bytes, so indexing will be required. Just as we indexed into the accelerometer `Buffer` for the addresses of the individual 16-bit acceleration values, we can index into the network packet array to get the starting addresses of the individual headers. Regular record component access syntax can then be used. Reading the record components reads the corresponding raw bytes in the array.

For a specific example, we can read the IP header in a packet's array of bytes using the header's record type and an access type designating that record type:

```
Min_IP_Header_Length : constant := 20;

--  IP packet header RFC 791.
type IP_Header is record
   Version         : UInt4;
   Word_Count      : UInt4;
   Type_of_Service : UInt8;
   Total_Length    : UInt16;
   Identifier      : UInt16;
   Flags_Offset    : UInt16;
   Time_To_Live    : UInt8;
   Protocol        : Transport_Protocol;
   Checksum        : UInt16;
   Source          : IP_Address;
   Destination     : IP_Address;
end record with
  Alignment => 2,
  Size      => Min_IP_Header_Length * 8;

for IP_Header use record
   Version         at 0  range 4 .. 7;
   Word_Count      at 0  range 0 .. 3;
   Type_of_Service at 1  range 0 .. 7;
   Total_Length    at 2  range 0 .. 15;
   Identifier      at 4  range 0 .. 15;
   Flags_Offset    at 6  range 0 .. 15;
   Time_To_Live    at 8  range 0 .. 7;
   Protocol        at 9  range 0 .. 7;
   Checksum        at 10 range 0 .. 15;
   Source          at 12 range 0 .. 31;
   Destination     at 16 range 0 .. 31;
end record;
for IP_Header'Bit_Order use System.Low_Order_First;
for IP_Header'Scalar_Storage_Order use System.Low_Order_First;

type IP_Header_Access is access all IP_Header;
pragma No_Strict_Aliasing (IP_Header_Access);

--  and so on, for the other kinds of headers...
```

Note that **pragma** *No_Strict_Aliasing* stops the compiler from doing some optimizations, based on the assumption of a lack of aliasing, that could cause unexpected results in this approach.

```
function As_IP_Header_Access is new Ada.Unchecked_Conversion
  (Source => System.Address, Target => IP_Header_Access);

function As_ARP_Header_Access is new Ada.Unchecked_Conversion
  (Source => System.Address, Target => ARP_Packet_Access);

--  and so on, for the other kinds of headers...
```

We can then implement a function, visible to clients, for acquiring the access value from a given memory buffer's data:

```
function IP_Hdr (This : Memory_Buffer) return IP_Header_Access is
  (As_IP_Header_Access (This.Packet.Data (IP_Pos)'Address));
```

In the function, `Data` is the packet's array of raw bytes, and `IP_Pos` is a constant specifying the index into the array corresponding to the first byte of the IP header. As you can see, this is the same approach as shown earlier for working with an array of bytes containing acceleration values.

Similar functions support ARP[32] headers, TCP[33] headers, and so on.

8.3 Pros

Both approaches work and are fairly simple, although the first is simplest. The second approach, based on converting addresses to access values, is more flexible. That's because the address to be converted can be changed at run-time, whereas the object overlay specifies the address exactly once during elaboration.

8.4 Cons

We're assuming access values are represented as addresses. There's no guarantee of that. But on typical architectures it will likely work.

That said, not all types can support the address conversion approach. In particular, unconstrained array types may not work correctly because of the existence of the additional in-memory representation of the bounds. An access value designating such an object might point at the bounds of the array whereas the address of the object would point to the first element.

In either approach, the developer is responsible for the correctness of the address values applied, either for the second object's declaration or for the pointer conversion. For example, this includes the alternative type's alignment. Otherwise, all bets are off.

8.5 Relationship With Other Idioms

None.

8.6 Notes

Generic package `System.Address_To_Access_Conversions` is an obvious alternative to our use of unchecked conversions between addresses and access values. The generic is convenient: it provides the access type as well as functions for converting in both directions. But it will require an instantiation for each designated type, so it offers no reduction in the number of instantiations required over that of `Ada.Unchecked_Conversion`. (For more details on this generic package, please refer to the section on access and address[34].)

Moreover, because that generic package is defined by the language, the naïve user might think it will work for all types. It might not. Unconstrained array types remain a potential problem. For that reason, the GNAT implementation issues a warning in such cases.

[32] https://en.wikipedia.org/wiki/Address_Resolution_Protocol
[33] https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[34] https://learn.adacore.com/courses/advanced-ada/parts/resource_management/access_types.html#adv-ada-access-address

CHAPTER

NINE

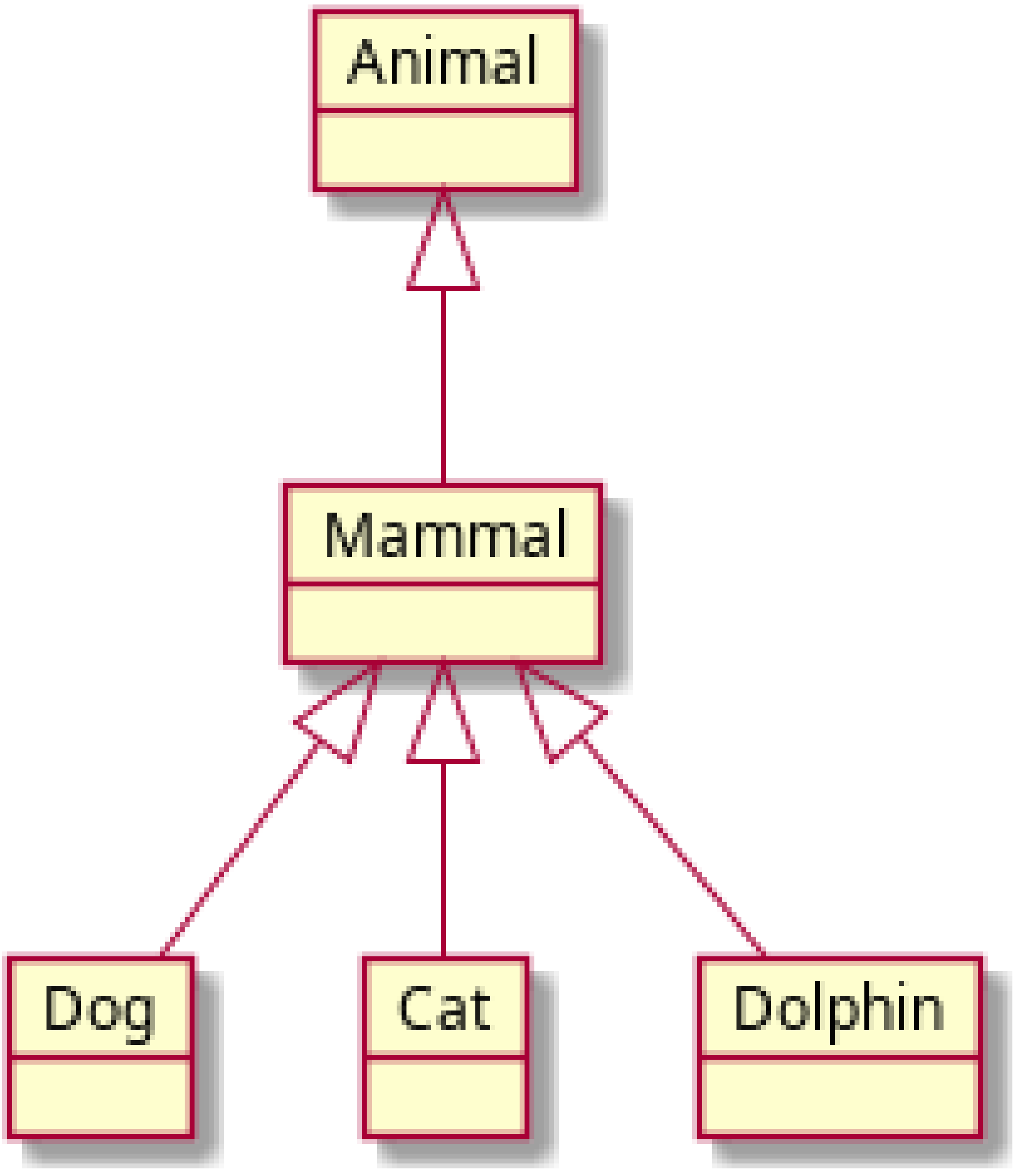
USING BUILDING BLOCKS TO EXPRESS INHERITANCE IDIOMS

9.1 Motivation

Betrand Meyer's magisterial book on OOP[38] includes a taxonomy of inheritance idioms. Two especially well-known entries in that taxonomy are Subtype Inheritance[36] and Implementation Inheritance[37]. The name of the first idiom is perhaps confusing from an Ada point of view because Ada subtypes have a different meaning. In Ada terms we are talking about *derived types*. A derived type is a new, distinct type based on (i.e., derived from) some existing type. We will informally refer to the existing ancestor type as the *parent* type, and the new type as the *child* type. The term *Subtype* in the idiom name refers to the child type.

Subtype Inheritance is the most well-known idiom for inheritance because it's based on the notion of a taxonomy, in which categories and disjoint subcategories are identified. For example, we can say that dogs, cats, and dolphins are mammals, and that all mammals are animals:

[38] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.
[36] https://en.wikipedia.org/wiki/Subtyping
[37] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)



By saying that the subcategories are disjoint we mean that, for example, dogs are neither cats nor dolphins and cannot be treated as if they are.

In software, we use various constructs to represent the categories and subcategories and use inheritance to organize them. As mentioned above, in Ada, we express that inheritance via derived types representing the categories and subcategories. Ada's strong typing ensures they are treated as disjoint entities.

Although the derived child type is distinct from the parent type, the child is the same *kind* as the parent type. Some authors use *kind of* as the name for the relationship between the child and parent. Meyer uses the term *is-a*[Page 61, 38], a popular term that we will use too. For example, a cat *is a* mammal, and also is an animal.

The fundamental difference between *Subtype Inheritance* (page 65) and *Implementation Inheritance* (page 68) is whether clients have compile-time visibility to the *is-a* relationship between the parent and child types. The relationship exists in both idioms but is only visible to clients in one. In Subtype Inheritance, clients do have compile-time visibility to the relationship, while in Implementation Inheritance, clients don't have that visibility.

Consequently, with Subtype Inheritance, all of the inherited operations become part of the child type's visible interface. In contrast, with Implementation Inheritance, none of those parent capabilities are part of the visible interface: the inherited parent capabilities are only available internally, to implement the child type's representation and its primitive operations.

9.1.1 Building Blocks

Ada uses distinct *building block* constructs to compose types that have specific characteristics and capabilities. In particular, Ada packages, with their control over compile-time visibility, are modules. Private types are combined with packages to define *abstract data types* (page 11) having hidden representations. Sets of related types are presented explicitly by class-wide types.

In addition, simple reserved words may be attached to a type declaration to refine or expand the capabilities of the type. These type declarations include declarations for derived types, providing considerable flexibility and expressive power for controlling the client's view of the child and parent types.

For example, in Ada, full dynamic OOP capabilities require type declarations to be decorated with the reserved word **tagged**. However, from its earliest days, Ada has also supported a static form of inheritance, using types that are not tagged. The implementation we describe below works with both forms of inheritance.

The developer also has a choice of whether the parent type and/or the child type is a private type. Using private types is the default design choice, for the sake of designing in terms of abstract data types, but is nevertheless optional.

In addition, a type can be both private and tagged. This possibility raises the question of whether the type is *visibly tagged*, i.e., whether the client view of the type includes the tagged characteristic, and hence the corresponding capabilities. Recall that a private type is declared in two steps: the first part occurs in the visible part of the package and introduces the type name to clients. The second part — the type completion — appears in the package private part and specifies the type's actual representation. The question arises because the first step, i.e., the declaration in the package's visible part, need not be tagged, yet can be tagged in the completion in the package private part. For example:

```
package P is
   type Foo is private;  --  not visibly tagged for clients
   -- operations on type Foo
private
   type Foo is tagged record  -- tagged completion
      ...
   end record;
end P;
```

In the above, Foo is not visibly tagged except in the package private part and the package body. As a consequence, the capabilities of tagged types are not available to clients using type Foo. Clients cannot refer to Foo'Class, for example. (The opposite arrangement — tagged in the visible client view but not actually tagged in the private view — is not legal, because clients would be promised capabilities that are not actually available.)

When the parent type is tagged, the type derivation syntax for the child is a *type extension* declaration that introduces the child type's name, specifies the parent type, and then extends the parent representation with child-specific record components, if any. For example:

```
type Child is new Parent with record ... end record;
```

Even though the child type declaration does not include the reserved word **tagged** the child will be a tagged type because the parent type is tagged. The compiler would not allow the extension construct for a non-tagged parent type.

Just as a private type can be visibly tagged or not, a private type can be *visibly derived* or not. When it is visibly derived, clients have a view of the private type that includes the fact of the derivation from the parent type. Otherwise, clients have no view of the parent type. Whether or not the child is visibly derived, the representation is not compile-time visible to clients, as for any private type. For example, type Foo is not visibly derived in the following:

```ada
package P is
   type Foo is tagged private;  --  visibly tagged but not visibly derived
   -- ...
end P;
```

To be visibly derived, we declare the child type as a private type using a *private extension*. A private extension is like a type extension, in that it introduces the child type name and the parent type. But like any private type declaration, it does not specify the type's representation. This is the first of the two steps for declaring a private type; hence it appears in the package visible part. For example:

```ada
with ...
package P is
   type Child is new Parent with private;  -- visibly derived from Parent
private
   type Child is new Parent with record ... end record;
end P;
```

The representation additions are not expressed until the private type's completion in the package private part, using a type extension. The steps are the same two for any private type: a declaration in the package visible part, with a completion in the package private part. The difference is the client visibility to the parent type.

9.2 Implementation(s)

There are two implementations presented, one for each of the two inheritance idioms under discussion. First, we will specify our building block choices, then show the two idiom expressions in separate subsections.

- We use tagged types for the sake of providing full OOP capabilities. That is the most common choice when inheritance is involved. The static form of inheritance has cases in which it is useful, but those cases are very narrow in applicability.
- We assume that the parent type and the child type are both private types, i.e., abstract data types, because that is the best practice. See the *Abstract Data Type idiom* (page 11) for justification and details.
- To provide the most general capabilities, we assume the parent type is visibly tagged.
- We're going to declare the child type in a distinct, dedicated package, following the *ADT idiom* (page 11). This package may or may not be a child of the parent package. This implementation's approach does not require a child package's special compile-time visibility, although a child package is often necessary for the sake of that visibility.
- Whether the child type is visibly derived will vary with the *inheritance idiom* (page 68) implementation.

To avoid unnecessary code duplication, we use the same parent type, declared as a simple tagged private type, in the examples for the two idiom implementations. The parent type could itself be derived from some other tagged type, but that changes nothing conceptually significant. We declare parent type in package P as follows:

```
package P is
   type Parent is tagged private;  -- visibly tagged
   --  primitive operations with type Parent as the
   --  controlling formal parameter
private
   type Parent is tagged record ... end record;
end P;
```

9.2.1 Subtype Inheritance

Recall that Subtype Inheritance requires clients to have compile-time visibility to the *is-a* relationship between the child and parent types. We can satisfy that requirement if we make the child visibly derived from the parent. Hence we declare the private type as a private extension in the visible part of the package:

```
with P; use P;
package Q is
  type Child is new Parent with private;
  --  implicit, inherited primitive Parent operations declared here,
  --  now for type Child
  --  additional primitives for Child explicitly declared, if any
private
  type Child is new Parent with record ... end record;
end Q;
```

The primitive operations from the parent type are implicitly declared immediately after the private extension declaration. That means those operations are in the visible part of the package, hence clients can invoke them. Any additional operations for the client interface will be explicitly declared in the visible part as well, as will any overriding declarations for those inherited operations that are to be changed.

For example, here is a basic bank account *ADT* (page 11) that we will use as the parent type in a derivation:

Listing 2: bank.ads

```
with Ada.Strings.Unbounded;  use Ada.Strings.Unbounded;
with Ada.Containers.Doubly_Linked_Lists;

package Bank is

   type Basic_Account is tagged private
     with Type_Invariant'Class => Consistent_Balance (Basic_Account);

   function Consistent_Balance (This : Basic_Account) return Boolean;

   type Currency is delta 0.01 digits 12;

   procedure Deposit (This   : in out Basic_Account;
                      Amount : Currency) with
     Pre'Class  => Open (This) and Amount > 0.0,
     Post'Class => Balance (This) = Balance (This)'Old + Amount;

   procedure Withdraw (This   : in out Basic_Account;
                       Amount : Currency) with
     Pre'Class  => Open (This) and Funds_Available (This, Amount),
     Post'Class => Balance (This) = Balance (This)'Old - Amount;

   function Balance (This : Basic_Account) return Currency
     with Pre'Class => Open (This);
```

(continues on next page)

(continued from previous page)

```

   procedure Report_Transactions (This : Basic_Account)
     with Pre'Class => Open (This);

   procedure Report (This : Basic_Account)
     with Pre'Class => Open (This);

   function Open (This : Basic_Account) return Boolean;

   procedure Open
     (This            : in out Basic_Account;
      Name            : String;
      Initial_Deposit : Currency)
   with Pre'Class => not Open (This),
        Post'Class => Open (This);

   function Funds_Available (This : Basic_Account;
                             Amount : Currency) return Boolean is
     (Amount > 0.0 and then Balance (This) >= Amount)
   with Pre'Class => Open (This);

private

   package Transactions is new
     Ada.Containers.Doubly_Linked_Lists (Element_Type => Currency);

   type Basic_Account is tagged record
      Owner           : Unbounded_String;
      Current_Balance : Currency := 0.0;
      Withdrawals     : Transactions.List;
      Deposits        : Transactions.List;
   end record;

   function Total (This : Transactions.List) return Currency is
     (This'Reduce ("+", 0.0));

end Bank;
```

We could then declare an interest-bearing bank account using Subtype Inheritance:

Listing 3: bank-interest_bearing.ads

```
package Bank.Interest_Bearing is

   type Account is new Basic_Account with private;

   overriding
   function Consistent_Balance (This : Account) return Boolean;

   function Minimum_Balance (This : Account) return Currency;

   overriding
   procedure Open
     (This            : in out Account;
      Name            : String;
      Initial_Deposit : Currency)
   with Pre => Initial_Deposit >= Minimum_Balance (This);

   overriding
   procedure Withdraw (This : in out Account;  Amount : Currency);

```

(continues on next page)

(continued from previous page)

```ada
   function Penalties_Accrued (This : Account) return Currency;
   function Interest_Accrued (This : Account) return Currency;

private

   type Account is new Basic_Account with record
      Penalties          : Transactions.List;
      Interest_Earned    : Transactions.List;
      Days_Under_Minimum : Natural := 0;
   end record;

end Bank.Interest_Bearing;
```

The new type `Bank.Interest_Bearing.Account` inherits all the `Basic_Account` operations in the package visible part. They are, therefore, available to clients. Some of those inherited operations are overridden so that their behavior can be changed. Additional operations specific to the new type are also declared in the visible part so they are added to the client API.

The package private part and the body of package `Bank.Interest_Bearing` have visibility to the private part of package Bank because the new package is a child of package Bank. That makes the private function `Bank.Total` visible in the child package, along with the components of the record type `Basic_Account`.

Note that there is no language requirement that the actual parent type in the private type's completion be the one named in the private extension declaration presented to clients. The parent type in the completion must only be in the same derivation class — be the same kind of type — so that it satisfies the *is-a* relationship stated to clients.

For example, we could start with a basic graphics shape:

```ada
package Graphics is
   type Shape is tagged private;
   --  operations for type Shape ...
   ...
end Graphics;
```

We could then declare a subcategory of Shape that allows translation in some 2-D space:

```ada
package Graphics.Translatable is
   type Translatable_Shape is new Graphics.Shape with private;
   procedure Translate (This : in out Translatable_Shape;  X, Y : in Float);
...
end Graphics.Translatable;
```

Given that, we could now declare another type visibly derived from Shape, but using `Translatable_Shape` as the actual parent type:

```ada
with Graphics;
private with Graphics.Translatable;
package Geometry is
   type Circle is new Graphics.Shape with private;
   --  operations for type Circle, inherited from Shape,
   --   and any new ops added ...
private
   use Graphics.Translatable;
   type Circle is new Translatable_Shape with record ... end record;
end Geometry;
```

In the type extension that completes type `Circle` in the package private part above, the extended parent type is not the one presented to clients, i.e., `Graphics.Shape`. Instead, the parent type is another type that is derived from type Shape. That substitution is legal and reasonable because Translatable_Shape necessarily can do anything that Shape can do. To understand why that is legal, it is helpful to imagine that there is a *contract* between the package public part and the private part regarding type Circle. As long as Circle can do everything promised to clients — i.e., inherited Shape operations — the contract is fulfilled. Circle inherits Shape operations because Translatable_Shape inherits those operations. The fact that Circle can do more than is contractually required by the client view is perfectly fine.

9.2.2 Implementation Inheritance

Recall that with Implementation Inheritance clients do not have compile-time visibility to the *is-a* relationship between the parent and child types. We meet that requirement by not making the child visibly derived from the parent. Therefore, we declare the child type as a simple tagged private type and only mention the parent in the child type's completion in the package private part:

```ada
with P; use P;
package Q is
  type Child is tagged private;
   --  explicitly declared primitives for Child
private
  type Child is new Parent with record ...
   --  implicit, inherited primitive operations with type Child
   --  as the controlling formal parameter
end Q;
```

The primitive operations from the parent type are implicitly declared immediately after the type extension, but these declarations are now located in the package private part. Therefore, the inherited primitive operations are not compile-time visible to clients. Hence clients cannot invoke them. These operations are only visible (after the type completion) in the package private part and the package body, for use with the implementation of the explicitly declared primitive operations.

For example, we might use a *controlled type* in the implementation of a tagged private type. These types have procedures Initialize and Finalize defined as primitive operations. Both are called automatically by the compiler. Clients generally don't have any business directly calling them so we usually use implementation inheritance with controlled types. But if clients did have the need to call them we would use Subtype Inheritance instead, to make them visible to clients.

For example, the following is a generic package providing an abstract data type for unbounded queues. As such, the Queue type uses dynamic allocation internally. This specific version automatically reclaims the allocated storage when objects of the Queue type cease to exist:

Listing 4: unbounded_sequential_queues.ads

```ada
with Ada.Finalization;
generic
   type Element is private;
package Unbounded_Sequential_Queues is

   type Queue is tagged limited private;

   procedure Insert (Into : in out Queue;  Item : Element) with
     Post => not Empty (Into) and
             Extent (Into) = Extent (Into)'Old + 1;
     --  may propagate Storage_Error

   procedure Remove (From : in out Queue;  Item : out Element) with
```

(continues on next page)

(continued from previous page)

```
     Pre  => not Empty (From),
     Post => Extent (From) = Natural'Max (0, Extent (From)'Old - 1);

   procedure Reset (This : in out Queue) with
      Post => Empty (This) and Extent (This) = 0;

   function Extent (This : Queue) return Natural;

   function Empty (This : Queue) return Boolean;

private

   type Node;

   type Link is access Node;

   type Node is record
      Data : Element;
      Next : Link;
   end record;

   type Queue is new Ada.Finalization.Limited_Controlled with
      record
         Count : Natural := 0;
         Rear  : Link;
         Front : Link;
      end record;

   overriding procedure Finalize (This : in out Queue) renames Reset;

end Unbounded_Sequential_Queues;
```

The basic operation of assignment usually does not make sense for an abstraction represented as a linked list, so we declare the private type as *limited*, in addition to tagged and private, and then use the language-defined limited controlled type for the type extension completion in the private part.

Procedures `Initialize` and `Finalize` are inherited immediately after the type extension. Both are null procedures that do nothing. We can leave `Initialize` as-is because initialization is already accomplished via the default values for the Queue components. On the other hand, we want finalization to reclaim all allocated storage so we cannot leave `Finalize` as a null procedure. By overriding the procedure, we can change the implementation. That change is usually accomplished by placing the corresponding procedure body in the package body. However, in this case we have an existing procedure named `Reset` that is part of the visible (client) API. `Reset` does exactly what we want `Finalize` to do, so we implement the overridden `Finalize` by saying that it is just another name for `Reset`. No completion body for `Finalize` is then required or allowed. This approach has the same semantics as if we explicitly wrote a body for `Finalize` that simply called `Reset`, but this is more succinct. Clients can call `Reset` whenever they want, but the procedure will also be called automatically, via `Finalize`, when any `Queue` object ceases to exist.

9.3 Pros

The two idioms are easily composed simply by controlling where in the enclosing package the parent type is mentioned: either in the declaration of the private child type in the package visible part or in the child type's completion in the package private type.

9.4 Cons

Although the inheritance expressions are simple by themselves, the many ancillary design choices can make the design effort seem more complicated than it really is.

9.5 Relationship With Other Idioms

We assume the *Abstract Data Type idiom* (page 11), so we are using private types throughout. That includes the child type, and, as we saw, allows us to control the compile-time visibility to the parent type.

CHAPTER

TEN

PROVIDING COMPONENT ACCESS TO ENCLOSING RECORD OBJECTS

10.1 Motivation

In some design situations we want to have a record component that is of a task or protected type. That in itself is trivially accomplished because task types and protected types can be used to declare record components. But there's more to this idiom.

We would want a task type or protected type record component when:

a) a task or protected object (PO) is required to implement part — but not all — of the record type's functionality, and

b) each such task or PO is intended to implement its functionality only for the object logically containing that specific task object or protected object. The record object and contained task/PO object pair is a functional unit, independent of all other such units.

This idiom applies to both enclosed task types and protected types, but for simplicity let's assume the record component will be of a protected type.

As part of a functional unit, the PO component will almost certainly be required to reference the other record components in the enclosing record object. That reference will allow the PO to read and/or update those other components. Note that these record components include discriminants, if any.

To be a functional unit, the record object referenced by a given PO in this relationship must be the same record object at run-time that contains that specific PO instance. That will allow the PO instance to implement the functionality for the specific record object containing that PO instance.

Unless we arrange it, that back-reference from the protected object to the record object isn't provided. Consider the following:

```
package Q is
   protected type P is ... end P;
   type R is record
     ...
     Y : P;
   end record;
end Q;
```

During execution, whenever an object of type `Q.R` is declared or allocated, at run-time we will have two objects, instances of two distinct types — the record object and the protected object. Let's say that a client declares an object `Obj` of type R. There is only one reference direction defined, from the record denoted by `Obj` to the component protected object denoted by `Obj.Y`. This idiom, however, requires a reference in the opposite direction, from `Oby.Y` to `Obj`.

This may seem like an unrealistic situation, but it is not. An IO device type that involves interrupt handling is just one real-world example, one that we will show in detail.

The idiom context is a type because there will often be multiple real-world entities being represented in software. Representing these entities as multiple objects declared of a single type is by far the most reasonable approach.

We assume the functional unit will be implemented as an *Abstract Data Type (ADT)* (page 11). Strictly speaking, the ADT idiom is not required here, but that is the best approach for defining major types, for the good reasons given in that idiom entry. There's no reason not to use an ADT in this case so we will.

10.2 Implementation(s)

As mentioned, the implementation approach applies to enclosed components of both task types and protected types. We will continue the discussion in terms of protected types.

The implementation has two parts:

1. An access discriminant on the PO type, designating the enclosing record's type. That part is straightforward.
2. A value given to that discriminant that designates the object of the enclosing record type, i.e., the record object that contains that PO. That part requires a relatively obscure language construct.

Given those two parts, the PO can then dereference its access discriminant to read or update the other components in the same enclosing record object.

Consider the following (very artificial) package declaration illustrating these two parts:

```ada
package P is
   type Device is tagged limited private;
private

   protected type Controller (Encloser : not null access Device) is
      -- Part 1

      procedure Increment_X;
   end Controller;

   type Device is tagged limited record
      X : Integer;  -- arbitrary type

      C : Controller (Encloser => ...);
      -- Part 2, not fully shown yet
   end record;

end P;
```

The record type named Device contains a component named X, arbitrarily of type **Integer**, and another component C that is of protected type `Controller`. Part #1 of the implementation is the access discriminant on the declaration of the protected type `Controller`:

```ada
protected type Controller (Encloser : not null access Device) is
```

Given a value for the discriminant `Encloser`, the code within the spec and body of type `Controller` can then reference some Device object via that discriminant.

But not just any object of type Device will suffice. For Part #2, we must give the `Encloser` discriminant a value that refers to the current instance of the record object containing this same PO object. In the package declaration above, the value passed to `Encloser` is elided.

The following is that code again, now showing just the declaration for Device, but also including the construct that is actually passed. This is where the subtlety comes into play:

```
type Device is tagged limited record
      ...
      C : Controller (Encloser => Device'Access);
end record;
```

The subtlety is the expression Device'Access. Within a type declaration, usage of the type name denotes the current instance of that type. The current instance of a type is the object of the type that is associated with the execution that evaluates the type name. For example, during execution, when an object of type Device is elaborated, the name Device refers to that object.

It isn't compiler-defined magic, the semantics are defined by the Ada standard so it is completely portable. (There are other cases for expressing the current instance of task types, protected types, and generics.)

Therefore, within the declaration of type Device, the expression Device'Access provides an access value designating the current instance of that type. This is exactly what we want and is the crux of the idiom expression. With that discriminant value, the enclosed PO spec and body can reference the other record components of the same object that contains the PO.

To illustrate that, here is the package body for this trivial example. Note the value referenced in the body of procedure Increment_X:

```
package body P is

   protected body Controller is

      procedure Increment_X is
      begin
         Encloser.X := Encloser.X + 1;
      end Increment_X;

   end Controller;

end P;
```

Specifically, the body of procedure Increment_X can use the access discriminant Encloser to get to the current instance's X component. (We could express it as Encloser.**all**.X but why bother. Implicit dereferencing is a wonderful thing.)

That's the implementation. Now for some necessary details.

Note that we declared type Device as a limited type, first in the visible part of the package:

```
type Device is tagged limited private;
```

and again in the type completion in the package private part:

```
type Device is tagged limited record ... end record;
```

We declare Device as a limited type because we want to preclude assignment statements for client objects of the type. Assignment of the enclosing record object would leave the PO Encloser discriminant designating the prior (right-hand side) enclosing object. If the PO is written with the assumption that the enclosing object is always the one identified during creation of the PO, that assumption will no longer hold. We didn't state it up-front, but that is the assumption underlying the idiom as described, and in fact, only limited types may have a component that uses the **Access** attribute in this way. Also note that any type that includes a protected or task object is limited, so a type like Device will necessarily be limited in any case.

The type need not be tagged for this approach, but it must be limited in both its partial view and its full view. More generally, a tagged type must be limited in both views if it is limited in either view.

For the idiom implementation to be legal, the type's completion in the private part must not merely be limited, but actually *immutably limited*, meaning that it is always truly limited. There are various ways to make that happen (see AARM22 7.5 (8.1/3)[39]) but the easiest way to is to include the reserved word **limited** in the type definition within the full view, as we did above. That is known as making the type *explicitly limited*. It turns out having a task or protected component also makes it immutably limited, so this requirement is naturally satisfied in this use case.

Why does the compiler require the type to be immutably limited?

Recall that a (non-tagged) private type need not be limited in both views. It can be limited in the partial client view but non-limited in its full view:

```ada
package Q is
   type T is limited private;
   -- the partial view for clients in package visible part
   ...
private
   type T is record -- the full view in the package private part
      ...
   end record;
end Q;
```

Clients must treat type Q.T as if it is limited, but Q.T isn't really limited because the full view defines reality. Clients simply have a more restricted view of the type than is really the case.

Types that are immutably limited are necessarily limited in all views. That's important because the current instance of the type given in type_name'Access must be aliased for 'Access to be legal. But if the type's view could change between limited and not limited, its current instance would be aliased in some contexts and not aliased in others. To prevent that complexity, the language requires the type to be immutably limited so that the current instance of the type will be aliased in every view. In practice, we're working with record types and type extensions, so just make the full type definition explicitly limited and all will be well:

```ada
package Q is
   type T is limited private;
   ...
private
   type T is limited record
      ...
   end record;
end Q;
```

Then, as mentioned, you can choose whether the type will also be tagged.

10.3 Real-World Example

For a concrete, real-world example, suppose we have a serial IO device on an embedded target board. The device can be either a UART or USART[40]. For the sake of brevity let's assume we have USARTs available.

Many boards have more than one USART resident, so it makes sense to represent them in software as instances of an ADT. This example uses the USART ADT supported in the

[39] http://www.ada-auth.org/standards/22aarm/html/AA-7-5.html
[40] https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

Ada Drivers Library (ADL)[41] that is named, imaginatively, USART. (We don't show package STM32.USARTs, but you will see it referenced in the example's context clauses.) Each of these USART devices can support either a polling implementation or an interrupt-driven implementation. We will first define a basic USART ADT, and then extend that to a new one that works with interrupts.

At the most basic level, to work with a given USART device we must combine it with some other hardware, specifically the IO pins that connect it to the outside world. That combination will be represented by a new ADT, the type Device defined in package Serial_IO.

Any given Serial_IO.Device object will be associated permanently with one USART. Therefore, type Device will have a discriminant named Transceiver designating that USART object.

There are some low-level operations that a Serial_IO.Device will implement, such as initializing the hardware and setting the baud rate and so forth. We can also implement the hardware-oriented input and output routines in this package because both are independent of the polling or interrupt-driven designs.

Here's the resulting package declaration for the serial IO device ADT. Parts of the package are elided for simplicity (the full code is *at the end of this idiom entry* (page 79)):

```
with STM32;         use STM32;
with STM32.GPIO;    use STM32.GPIO;
with STM32.USARTs;  use STM32.USARTs;
with HAL;  -- the ADL's Hardware Abstraction Layer

package Serial_IO is

   type Device (Transceiver : not null access USART) is tagged limited private;

   procedure Initialize
     (This           : in out Device;
      Tx_Pin         : GPIO_Point;
      Rx_Pin         : GPIO_Point;
      ...);

   procedure Configure (This : in out Device;  Baud_Rate : Baud_Rates;  ...);
   ...
   procedure Put (This : in out Device;  Data : HAL.UInt8) with Inline;
   procedure Get (This : in out Device;  Data : out HAL.UInt8) with Inline;

private

   type Device (Transceiver : not null access USART) is tagged limited record
      Tx_Pin : GPIO_Point;
      Rx_Pin : GPIO_Point;
      ...
   end record;

end Serial_IO;
```

When called, procedure Initialize does the hardware setup required, such as enabling power for the USART and pins. We can ignore those details for this discussion.

Given this basic Device type we can then use inheritance (type extension) to define distinct types that support the polling and interrupt-driven facilities. These types will themselves be ADTs. Let's focus on the new interrupt-driven ADT, named Serial_Port. This type will be declared in the child package Serial_IO.Interrupt_Driven.

When interrupts are used, each USART raises a USART-specific interrupt for sending and receiving. Each interrupt occurrence is specific to one device. Therefore, the interrupt

[41] https://github.com/AdaCore/Ada_Drivers_Library

handler code is specific to each Serial_Port object instance. We use protected objects as interrupt handlers in (canonical) Ada, hence each Serial_Port object will contain a dedicated interrupt handling PO as a record component.

As a controller and handler for a USART's interrupts, the PO will require a way to access the USART and pins being driven. Our idiom design provides that access.

Here is the client view of the ADT for the interrupt-driven implementation:

```
with Ada.Interrupts;      use Ada.Interrupts;
with HAL;
with System; use System;

package Serial_IO.Interrupt_Driven is

   type Serial_Port
     (Transceiver  : not null access USART;
      IRQ          : Interrupt_ID;
      IRQ_Priority : Interrupt_Priority)
   is new Serial_IO.Device with private;

   --  The procedures Initialize and Configure, among others, are
   --  implicitly declared here as operations inherited from
   --  Serial_IO.Device.

   overriding
   procedure Put (This : in out Serial_Port;  Data : HAL.UInt8)
     with Inline;

   overriding
   procedure Get (This : in out Serial_Port;  Data : out HAL.UInt8)
     with Inline;

private
   ...
end Serial_IO.Interrupt_Driven;
```

The declaration of type Serial_Port uses *Interface Inheritance* (page 61) to extend Serial_IO.Device with both visible and hidden components. The three visible extension components are the discriminants Transceiver, IRQ, and IRQ_Priority. Transceiver will designate the USART to drive (discussed in a moment). IRQ is the Interrupt_ID indicating the interrupt that the associated USART raises. IRQ_Priority is the priority for that interrupt. (*IRQ* in a common abbreviation for *Interrupt ReQuest*.) These two interrupt-oriented discriminants are used within the PO declaration to configure it for interrupt handling.

Clients will know which USART they are working with so they will be able to determine which interrupt ID and priority to specify, presumably by consulting the board documentation.

Now let's examine the Serial_Port type completion in the package's private part.

We've said we will use a PO interrupt handler as a component of the Serial_Port record type. This PO type, named IO_Manager, will include discriminants for the two interrupt-specific values it requires as an interrupt handler. It will also have a discriminant providing access to the enclosing Serial_Port record type.

```
protected type IO_Manager
  (IRQ          : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority;
   Port         : not null access Serial_Port)
with
   Interrupt_Priority => IRQ_Priority
is
   entry Put (Datum : HAL.UInt8);
```

(continues on next page)

(continued from previous page)

```
   entry Get (Datum : out HAL.UInt8);
private
   ...
   procedure IRQ_Handler with Attach_Handler => IRQ;
end IO_Manager;
```

Note how the first two discriminants are used within the type declaration to give the priority of the PO and to attach the interrupt handler procedure IRQ_Handler to the interrupt indicated by IRQ. The Port discriminant will be the back-reference to the enclosing record object.

We can then, finally, provide the Serial_Port type completion, in which the record object and protected object are connected whenever a Serial_Port object is declared:

```
type Serial_Port
  (Transceiver  : not null access USART;
   IRQ          : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority)
is new Serial_IO.Device (Transceiver) with record
   Controller : IO_Manager (IRQ, IRQ_Priority, Serial_Port'Access);
end record;
```

The type completion repeats the declaration in the public part, up to the point where the Serial_Port.Transceiver discriminant is passed to the Serial_IO.Device.Transceiver discriminant. Type Device must be constrained with a discriminant value here, so we just pass the discriminant defined for Serial_Port.

Why does Serial_Port also have a Transceiver discriminant? Just as Serial_IO.Device is a complete wrapper for the combination of a USART and IO pins, Serial_Port is a standalone wrapper for Serial_IO.Device. Hence Serial_Port also needs a discriminant designating a USART to be complete.

The full definition of type Serial_Port contains the declaration of the component named Controller, of the protected type IO_Manager. The two interrupt-oriented discriminants from Serial_Port are passed to the discriminants defined for this PO component. The third IO_Manager discriminant value, Serial_Port'Access, denotes the current instance of the Serial_Port type. Thus the idiom requirements are achieved.

Let's see that back-reference in use within the protected body.

As mentioned, there is one interrupt used for both sending and receiving, per USART. Strictly speaking, the device itself does use two dedicated interrupts, one indicating that a 9-bit value has been received, and one indicating that transmission for a single 9-bit value has completed. But these two are signaled to the software on one interrupt line, and that is the value indicated by IRQ.

Therefore, there is one interrupt handling protected procedure, named IRQ_Handler. In response to this interrupt, IRQ_Handler determines which event has occurred by checking one of the Transceiver status registers. The back-reference through Port makes that possible. Other Transceiver routines are also called via Port, and Port.**all** is passed to the Put and Get calls:

```
procedure IRQ_Handler is
begin
   --  check for data arrival
   if Port.Transceiver.Status (Read_Data_Register_Not_Empty) and then
       Port.Transceiver.Interrupt_Enabled (Received_Data_Not_Empty)
   then  -- handle reception
      -- call the Serial_IO.Device version:
      Get (Serial_IO.Device (Port.all), Incoming);
```

(continues on next page)

(continued from previous page)

```
      Await_Reception_Complete : loop
         exit when not Port.Transceiver.Status (Read_Data_Register_Not_Empty);
      end loop Await_Reception_Complete;
      Port.Transceiver.Disable_Interrupts (Received_Data_Not_Empty);
      Port.Transceiver.Clear_Status (Read_Data_Register_Not_Empty);
      Incoming_Data_Available := True;
   end if;

   --  check for transmission ready
   if Port.Transceiver.Status (Transmission_Complete_Indicated) and then
      Port.Transceiver.Interrupt_Enabled (Transmission_Complete)
   then  -- handle transmission
      -- call the Serial_IO.Device version:
      Put (Serial_IO.Device (Port.all), Outgoing);

      Port.Transceiver.Disable_Interrupts (Transmission_Complete);
      Port.Transceiver.Clear_Status (Transmission_Complete_Indicated);
      Transmission_Pending := False;
   end if;
end IRQ_Handler;
```

In this example, although the PO only accesses the `Transceiver` component in the enclosing record object, additional functionality might need to access more components, for this example perhaps using some of the inherited IO pin components.

10.4 Pros

The implementation is directly expressed, requiring only an access discriminant and the current instance semantics of `type_name'Access`.

Although the real-word example is complex — multiple discriminants are involved, and a type extension — the implementation itself requires little text. Interrupt handling is relatively complex in any language.

10.5 Cons

The record type must be truly a limited type, but that is not the severe limitation it was in earlier versions of Ada. Note that although access discriminants are required, there is no dynamic allocation involved.

10.6 Relationship With Other Idioms

This idiom is useful when we have a record type enclosing a PO or task object. If the *Abstract Data Machine (ADM)* (page 17) would instead be appropriate, the necessary visibility can be achieved without requiring this implementation approach because there would be no enclosing record type. But as described in the ADM discussion, the *ADT approach* (page 11) is usually superior.

10.7 Notes

As a wrapper abstraction for a USART, package `Serial_IO` is still more hardware-specific than absolutely necessary, as reflected in the parameters' types for procedure `Initialize` and the corresponding record component types. We could use the Hardware Abstraction

Layer (HAL) to further isolate the hardware dependencies, but that doesn't affect the idiom expression itself.

10.8 Full Source Code for Selected Units

We did not show some significant parts of the code discussed above, for the sake of not obscuring the points being made. Doing so, however, means that the interested reader cannot see how everything fits together and works, such as the actual IO using interrupts. The code below shows the relevant packages in their entirety. Note that the ADL STM32 hierarchy packages and the HAL (Hardware Abstraction Layer) package are in the Ada Drivers Library on GitHub[42].

First, the basic `Serial_IO` abstraction:

```
with STM32;         use STM32;
with STM32.GPIO;    use STM32.GPIO;
with STM32.USARTs;  use STM32.USARTs;
with HAL;

package Serial_IO is

   type Device (Transceiver : not null access USART) is tagged limited private;

   procedure Initialize
     (This           : in out Device;
      Transceiver_AF : GPIO_Alternate_Function;
      Tx_Pin         : GPIO_Point;
      Rx_Pin         : GPIO_Point;
      CTS_Pin        : GPIO_Point;
      RTS_Pin        : GPIO_Point);
   --  must be called before Configure

   procedure Configure
     (This      : in out Device;
      Baud_Rate : Baud_Rates;
      Parity    : Parities     := No_Parity;
      Data_Bits : Word_Lengths := Word_Length_8;
      End_Bits  : Stop_Bits    := Stopbits_1;
      Control   : Flow_Control := No_Flow_Control);

   procedure Set_CTS (This : in out Device; Value : Boolean) with Inline;
   procedure Set_RTS (This : in out Device; Value : Boolean) with Inline;

   procedure Put (This : in out Device;  Data : HAL.UInt8)     with Inline;
   procedure Get (This : in out Device;  Data : out HAL.UInt8) with Inline;

private

   type Device (Transceiver : not null access USART) is tagged limited record
      Tx_Pin  : GPIO_Point;
      Rx_Pin  : GPIO_Point;
      CTS_Pin : GPIO_Point;
      RTS_Pin : GPIO_Point;
   end record;

end Serial_IO;
```

And the package body:

[42] https://github.com/AdaCore/Ada_Drivers_Library

```
with STM32.Device; use STM32.Device;

package body Serial_IO is

   ----------------
   -- Initialize --
   ----------------

   procedure Initialize
     (This           : in out Device;
      Transceiver_AF : GPIO_Alternate_Function;
      Tx_Pin         : GPIO_Point;
      Rx_Pin         : GPIO_Point;
      CTS_Pin        : GPIO_Point;
      RTS_Pin        : GPIO_Point)
   is
      IO_Pins : constant GPIO_Points := Rx_Pin & Tx_Pin;
   begin
      This.Tx_Pin := Tx_Pin;
      This.Rx_Pin := Rx_Pin;
      This.CTS_Pin := CTS_Pin;
      This.RTS_Pin := RTS_Pin;

      Enable_Clock (This.Transceiver.all);

      Enable_Clock (IO_Pins);

      Configure_IO
        (IO_Pins,
         Config => (Mode_AF,
                    AF             => Transceiver_AF,
                    AF_Speed       => Speed_50MHz,
                    AF_Output_Type => Push_Pull,
                    Resistors      => Pull_Up));

      Enable_Clock (RTS_Pin & CTS_Pin);

      Configure_IO (RTS_Pin, Config => (Mode_In, Resistors => Pull_Up));

      Configure_IO
        (CTS_Pin,
         Config => (Mode_Out,
                    Speed       => Speed_50MHz,
                    Output_Type => Push_Pull,
                    Resistors   => Pull_Up));
   end Initialize;

   ---------------
   -- Configure --
   ---------------

   procedure Configure
     (This      : in out Device;
      Baud_Rate : Baud_Rates;
      Parity    : Parities     := No_Parity;
      Data_Bits : Word_Lengths := Word_Length_8;
      End_Bits  : Stop_Bits    := Stopbits_1;
      Control   : Flow_Control := No_Flow_Control)
   is
   begin
      This.Transceiver.Disable;
```

(continues on next page)

(continued from previous page)

```
      This.Transceiver.Set_Baud_Rate    (Baud_Rate);
      This.Transceiver.Set_Mode         (Tx_Rx_Mode);
      This.Transceiver.Set_Stop_Bits    (End_Bits);
      This.Transceiver.Set_Word_Length  (Data_Bits);
      This.Transceiver.Set_Parity       (Parity);
      This.Transceiver.Set_Flow_Control (Control);

      This.Transceiver.Enable;
   end Configure;

   -------------
   -- Set_CTS --
   -------------

   procedure Set_CTS (This : in out Device; Value : Boolean) is
   begin
      This.CTS_Pin.Drive (Value);
   end Set_CTS;

   -------------
   -- Set_RTS --
   -------------

   procedure Set_RTS (This : in out Device; Value : Boolean) is
   begin
      This.RTS_Pin.Drive (Value);
   end Set_RTS;

   ---------
   -- Put --
   ---------

   procedure Put (This : in out Device;  Data : HAL.UInt8) is
   begin
      This.Transceiver.Transmit (HAL.UInt9 (Data));
   end Put;

   ---------
   -- Get --
   ---------

   procedure Get (This : in out Device;  Data : out HAL.UInt8) is
      Received : HAL.UInt9;
   begin
      This.Transceiver.Receive (Received);
      Data := HAL.UInt8 (Received);
   end Get;

end Serial_IO;
```

Next, the interrupt-driven extension.

```
with Ada.Interrupts;      use Ada.Interrupts;
with HAL;
with System; use System;

package Serial_IO.Interrupt_Driven is
   pragma Elaborate_Body;

   type Serial_Port
     (Transceiver  : not null access USART;
```

(continues on next page)

(continued from previous page)

```
      IRQ          : Interrupt_ID;
      IRQ_Priority : Interrupt_Priority)
   is new Serial_IO.Device with private;
   --  A serial port that uses interrupts for I/O. Extends the Device
   --  abstraction that is itself a wrapper for the USARTs hardware.

   --  The procedures Initialize and Configure, among others, are implicitly
   --  declared here, as operations inherited from Serial_IO.Device

   overriding
   procedure Put (This : in out Serial_Port;  Data : HAL.UInt8)
     with Inline;
   --  Non-blocking, ie the caller can return before the Data goes out,
   --  but does block until the underlying UART is not doing any other
   --  transmitting. Does no polling. Will not interfere with any other I/O
   --  on the same device.

   overriding
   procedure Get (This : in out Serial_Port;  Data : out HAL.UInt8)
     with Inline;
   --  Blocks the caller until a character is available! Does no polling.
   --  Will not interfere with any other I/O on the same device.

private

   --  The protected type defining the interrupt-based I/O for sending and
   --  receiving via the USART attached to the serial port designated by
   --  Port. Each serial port object of the type defined by this package has
   --  a component of this protected type.
   protected type IO_Manager
     (IRQ          : Interrupt_ID;
      IRQ_Priority : Interrupt_Priority;
      Port         : not null access Serial_Port)
   --  with
      --  Interrupt_Priority => IRQ_Priority   -- compiler bug :-(
   is
      pragma Interrupt_Priority (IRQ_Priority);

      entry Put (Datum : HAL.UInt8);

      entry Get (Datum : out HAL.UInt8);

   private

      Outgoing : HAL.UInt8;
      Incoming : HAL.UInt8;

      Incoming_Data_Available : Boolean := False;
      Transmission_Pending    : Boolean := False;

      procedure IRQ_Handler with Attach_Handler => IRQ;

    end IO_Manager;

   type Serial_Port
     (Transceiver  : not null access USART;
      IRQ          : Interrupt_ID;
      IRQ_Priority : Interrupt_Priority)
   is
      new Serial_IO.Device (Transceiver) with
   record
```

(continues on next page)

(continued from previous page)

```
      Controller : IO_Manager (IRQ, IRQ_Priority, Serial_Port'Access);
      --  Note that Serial_Port'Access provides the Controller with a view
      --  to the current instance's components, including the discriminant
      --  components
   end record;

end Serial_IO.Interrupt_Driven;
```

And the package body:

```
with STM32.Device; use STM32.Device;

package body Serial_IO.Interrupt_Driven is

   ---------
   -- Put --
   ---------

   overriding
   procedure Put (This : in out Serial_Port;  Data : HAL.UInt8) is
   begin
      This.Controller.Put (Data);
   end Put;

   ---------
   -- Get --
   ---------

   overriding
   procedure Get (This : in out Serial_Port;  Data : out HAL.UInt8) is
   begin
      This.Transceiver.Enable_Interrupts (Received_Data_Not_Empty);
      This.Controller.Get (Data);
   end Get;

   ----------------
   -- IO_Manager --
   ----------------

   protected body IO_Manager is

      -----------------
      -- IRQ_Handler --
      -----------------

      procedure IRQ_Handler is
      begin
         --  check for data arrival
         if Port.Transceiver.Status (Read_Data_Register_Not_Empty) and then
            Port.Transceiver.Interrupt_Enabled (Received_Data_Not_Empty)
         then  -- handle reception
            -- call the Serial_IO.Device version:
            Get (Serial_IO.Device (Port.all), Incoming);

            Await_Reception_Complete : loop
               exit when not
                 Port.Transceiver.Status (Read_Data_Register_Not_Empty);
            end loop Await_Reception_Complete;
            Port.Transceiver.Disable_Interrupts (Received_Data_Not_Empty);
            Port.Transceiver.Clear_Status (Read_Data_Register_Not_Empty);
            Incoming_Data_Available := True;
```

(continues on next page)

(continued from previous page)

```
         end if;

         --  check for transmission ready
         if Port.Transceiver.Status (Transmission_Complete_Indicated) and then
            Port.Transceiver.Interrupt_Enabled (Transmission_Complete)
         then  -- handle transmission
            -- call the Serial_IO.Device version:
            Put (Serial_IO.Device (Port.all), Outgoing);

            Port.Transceiver.Disable_Interrupts (Transmission_Complete);
            Port.Transceiver.Clear_Status (Transmission_Complete_Indicated);
            Transmission_Pending := False;
         end if;
      end IRQ_Handler;
      ---------
      -- Put --
      ---------

      entry Put (Datum : HAL.UInt8) when not Transmission_Pending is
      begin
         Transmission_Pending := True;
         Outgoing := Datum;
         Port.Transceiver.Enable_Interrupts (Transmission_Complete);
      end Put;

      ---------
      -- Get --
      ---------

      entry Get (Datum : out HAL.UInt8) when Incoming_Data_Available is
      begin
         Datum := Incoming;
         Incoming_Data_Available := False;
      end Get;

   end IO_Manager;

end Serial_IO.Interrupt_Driven;
```

CHAPTER

ELEVEN

INTERRUPT HANDLING

11.1 Motivation

Recall that, in Ada, protected procedures are the standard interrupt-handling mechanism. The canonical interrupt handling and management model is defined in the Systems Programming Annex, section C.3 of the Reference Manual[43]. We assume that this optional annex is supported, and indeed effectively all compilers do support it. Likewise, we assume that the Real-Time Annex, annex D[44], is supported (which would require Annex C[45] to be supported anyway). Finally, we assume that either the Ravenscar or the Jorvik usage profile is applied. These two profiles define configurations of the two annexes that are appropriate for typical embedded systems that handle interrupts.

The definition of a canonical model mitigates differences imposed by the target, but some remain. For example, the number of different priority values, including interrupt priorities, differs with the targets involved. The model supports blocking of those interrupts at a lower priority than the currently executing interrupt handler, but the hardware might not support that behavior, although many do. None of these variations affect the expression of the idioms themselves.

The response to interrupts is often arranged in logical levels. The first level is the protected procedure handler itself. In some cases, everything required to handle the interrupt is performed there. However, some applications require more extensive, asynchronous processing of the data produced by the first level interrupt handler. In this case a second-level response can be defined, consisting of a task triggered by the first level. For example, the interrupt handler could respond to the first arrival of a character on a USART[46], poll for the remainder (or not), and then notify a task to perform analysis of the entire string received.

But even if no second-level interrupt processing is required, the interrupt handler may be required to notify the application that the event has occurred. Because interrupts are asynchronous, and logically concurrent with the application code, the association of an application task to a given interrupt-driven event is convenient and common.

Hence a task is often involved. How the handler procedure notifies the task leads to a couple of different idiom implementations. In both cases notification amounts to releasing the previously suspended task for further execution.

In the following section we show how to express these three idioms: one for using protected procedures alone, and two in which a protected procedure handler notifies a task.

[43] http://www.ada-auth.org/standards/12rm/html/RM-C-3.html
[44] http://www.ada-auth.org/standards/12rm/html/RM-D.html
[45] http://www.ada-auth.org/standards/12rm/html/RM-C.html
[46] https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

11.2 Implementation(s)

11.2.1 First Level Handler Alone

In this approach the interrupt handler protected procedure does everything necessary and does not require a second-level handler.

An interrupt handler that simply copies data from one location to another is a good example of a necessary and sufficient first-level handler. The enclosing application assumes the copying is occurring as required and needs no explicit notification. If the copying isn't happening the failure will be obvious.

So, given that, why discuss such a scenario? Two reasons: to show how it is done in general, and especially, to show how double-buffering can be implemented very elegantly with interrupts.

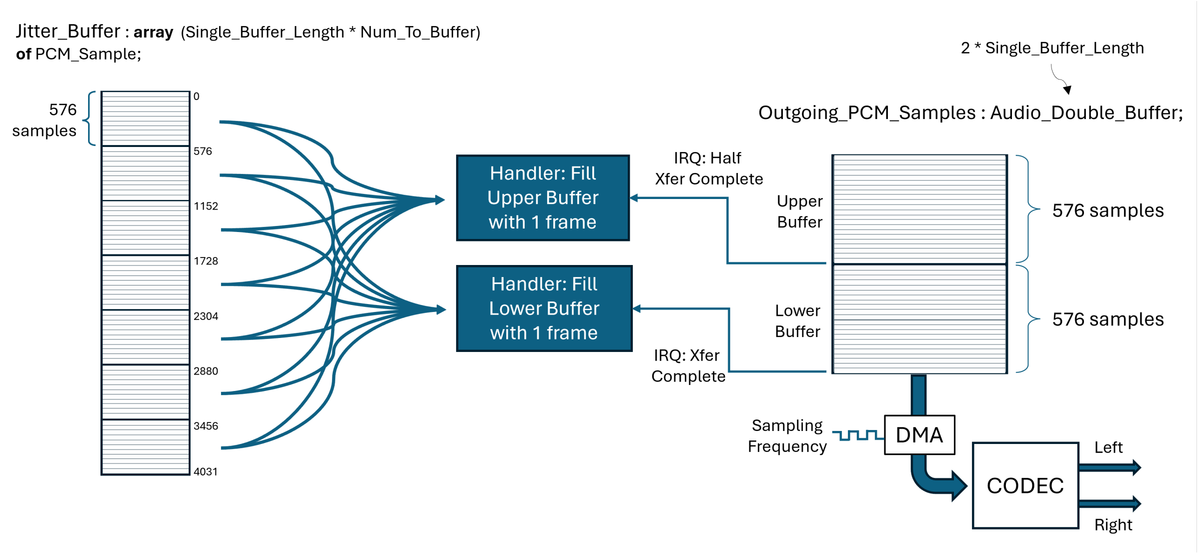
For a concrete example, consider an audio streaming device that takes PCM samples from Ethernet via incoming UDP packets and transfers them to an audio CODEC device[47] on the target board. The CODEC output is physically connected to a high-quality amplifier and speakers. No upper-level application thread requires notification of the copying: if the transfer is working the audio output occurs, otherwise it does not.

In our implementation the CODEC device is *fed* from a buffer named `Outgoing_PCM_Samples`. The buffer must always have new samples available when the CODEC is ready for them, because delays or breaks would introduce audible artifacts. The timing is determined by the sampling rate used by the audio source, prior to transmission. To match that rate and to provide it efficiently, we use DMA to transfer the data from the buffer to the CODEC. In addition, `Outgoing_PCM_Samples` is double-buffered to help ensure the samples are always available upon demand.

However, the incoming UDP packets don't arrive at exact intervals. Because of this *jitter* in the arrival times, we cannot directly insert the PCM samples from these incoming packets into the `Outgoing_PCM_Samples` buffer. The delays would be audible. Therefore, we use a *jitter buffer* to deal with the arrival time variations. This jitter buffer holds the PCM samples as they arrive in the UDP packets, in sufficient amounts to de-couple the arrival time jitter from the outgoing data. A jitter buffer can do much more than this, such as correcting the order of arriving packets, but in this specific case the additional functionality is not required.

We use two DMA interrupts to copy data from the jitter buffer to the `Outgoing_PCM_Samples` buffer. The rationale for using two interrupts, rather than one, is given momentarily. The figure below illustrates the overall approach, with the jitter buffer on the left, the two interrupt handlers in the middle, and the `Outgoing_PCM_Samples` buffer on the right.

[47] https://en.wikipedia.org/wiki/Audio_codec



Each UDP packet contains 576 PCM samples, used as the *single buffer length* for the double-buffered `Outgoing_PCM_Samples` and the `Jitter_Buffer`.

The advantage of double-buffering is that the producer can be filling one buffer while the consumer is removing data from the other. These directions switch when the current output buffer becomes empty. The result is a fast, continuous output stream. Many audio and video devices use double-buffering for that reason.

To express double-buffering you could use two physically distinct array objects, switching between them when the DMA controller signals that the current outgoing buffer is empty. That would require keeping track of which buffer is being filled and which is being emptied. There is an elegant, simpler alternative that uses two different DMA interrupts instead of one. (The DMA device must support this approach directly.)

In this alternative, there is one physical array (`Outgoing_PCM_Samples`), containing twice the number of components as a single physical buffer would contain. We can then use the two interrupts to treat the one physical array as two logical buffers.

The two DMA interrupts are triggered as the DMA transfer consumes the content within this single array. One interrupt is triggered when the transfer reaches the physical half-way point in the array. The other interrupt is triggered when the transfer reaches the physical end of the array. Therefore, because the array is twice the size of a single buffer, each interrupt corresponds to one of the two logical buffers becoming empty.

Furthermore, the DMA device generating these interrupts is configured so that it does not stop. After triggering the *half transfer complete* interrupt the DMA continues reading, now from the second logical buffer. After triggering the *transfer complete* interrupt the DMA device starts over at the beginning of the array, reading from the first logical buffer again.

Therefore, we have two distinct interrupt handlers, one for each of the two interrupts. When the *half transfer complete* handler is invoked, the upper logical buffer is now empty, so the handler for that half fills it. Likewise, the *transfer complete* interrupt handler fills the lower logical buffer at the bottom half of the array. There's no need to keep track of which buffer is being filled or emptied. It's all being emptied, and the handlers always fill the same upper or lower halves of the array. As long as each handler completes filling their half before the DMA transfer begins reading it, all is well.

Here's the declaration of the protected object containing the DMA interrupt handling code.

```
protected DMA_Interrupt_Controller with
  Interrupt_Priority => DMA_Interrupt_Priority
is
private
```

(continues on next page)

(continued from previous page)

```
   procedure DMA_IRQ_Handler with
      Attach_Handler => STM32.Board.Audio_Out_DMA_Interrupt;
end DMA_Interrupt_Controller;
```

A few points are worth highlighting.

First, DMA_Interrupt_Priority is an application-defined constant. The actual value isn't important to this discussion. The handler procedure is attached to an interrupt that is specific to the target board, so it is defined in the package STM32.Board in the Ada Drivers Library. Each target board supported by the library has such a package, always with the same package name. This particular STM32 board has dedicated audio DMA support, along with the CODEC.

Second, there's nothing declared in the visible part of the PO. More to the point, everything is declared in the optional private part. That placement is a matter of style, but it's good style. No software client should ever call the protected procedure — only the hardware should call it, via the runtime library — so we make it impossible for any client to call it accidentally. That placement also informs the reader of our intent.

Third, we said there are two interrupts, but only one interrupt handler procedure is declared and attached. There's nothing inherently wrong with one routine handling multiple interrupts, although conceptually it is not ideal. In this case it is necessary because on this target both device interrupts arrive at the MCU on one external interrupt line. Therefore, the one protected procedure handler handles both device interrupts, querying the DMA status flags to see which interrupt is active. This approach is shown below. Note that there must be an enclosing package, with multiple context clauses, but we do not show them so that we can focus on the interrupt handler itself.

```
protected body DMA_Interrupt_Controller is

   procedure DMA_IRQ_Handler is
      use STM32.Board; -- for the audio DMA
   begin
      if Status (Audio_DMA,
                 Audio_DMA_Out_Stream,
                 DMA.Half_Transfer_Complete_Indicated)
      then
         --  The middle of the double-buffer array has been reached by the
         --  DMA transfer, therefore the "upper half buffer" is empty.
         Fill_Logical_Buffer (Outgoing_PCM_Samples,
                              Starting_Index => Upper_Buffer_Start);
         Clear_Status (Audio_DMA,
                       Audio_DMA_Out_Stream,
                       DMA.Half_Transfer_Complete_Indicated);
      end if;

      if Status (Audio_DMA,
                 Audio_DMA_Out_Stream,
                 DMA.Transfer_Complete_Indicated)
      then
         --  The bottom of the double-buffer array has been reached by the
         --  DMA transfer, therefore the "lower half buffer" is empty.
         Fill_Logical_Buffer (Outgoing_PCM_Samples,
                              Starting_Index => Lower_Buffer_Start);
         Clear_Status (Audio_DMA,
                       Audio_DMA_Out_Stream,
                       DMA.Transfer_Complete_Indicated);
      end if;
   end DMA_IRQ_Handler;

end DMA_Interrupt_Controller;
```

In both cases Fill_Logical_Buffer is called to insert samples from the jitter buffer into one of the logical buffers. The difference is the value passed to the formal parameter Starting_Index. This is the array index at which filling should begin within Outgoing_PCM_Samples. Upper_Buffer_Start corresponds to Outgoing_PCM_Samples'First, and Lower_Buffer_Start is Outgoing_PCM_Samples'First + Single_Buffer_Length.

That's all the software has to do. Offloading work to the hardware, in this case the DMA controller, is always a good idea, but that's especially true for less powerful targets, e.g., microcontrollers. Note that the availability of the *half transfer complete* interrupt varies across different DMA devices.

The implementation of Fill_Logical_Buffer is straightforward and need not be shown. However, the procedure declares a local variable named Incoming_PCM_Samples that has ramifications worth noting. In particular, the representation may require altering and rebuilding the underlying Ada run-time library.

The object Incoming_PCM_Samples is declared within Fill_Logical_Buffer like so:

```
Incoming_PCM_Samples : Jitter_Buffer.Sample_Buffer_Slice;
```

The alteration might be required because Fill_Logical_Buffer executes entirely in the interrupt handler procedure's context. Hence the storage used by the procedure's execution comes from the interrupt handler stack. Interrupt handlers typically do relatively little, and, as a result, a relatively small stack allocation is typically defined for them. The storage for Incoming_PCM_Samples might exceed that allocation.

Specifically, we said that Fill_Logical_Buffer fills an entire half of the double-buffer, i.e., it works in terms of Single_Buffer_Length. If Sample_Buffer_Slice is an actual array, the required storage might be considerable.

The interrupt stack allocation is set by the run-time library source code in GNAT, as is common. You could increase the allocation and rebuild the run-time.

On the other hand, Sample_Buffer_Slice need not be an actual array. It could be a record type containing a (read-only) pointer to the jitter buffer array and an index indicating where in that array the *slice* to be transferred begins. That representation would obviously require much less stack space, obviating the run-time library change and rebuild. Moreover, that representation would allow Fill_Logical_Buffer to copy directly from the jitter buffer into the final destination, i.e., Outgoing_PCM_Samples. If Incoming_PCM_Samples is an array, we'd have to copy from the jitter buffer into Incoming_PCM_Samples, and then again from there to Outgoing_PCM_Samples. That's an extra copy operation we can avoid.

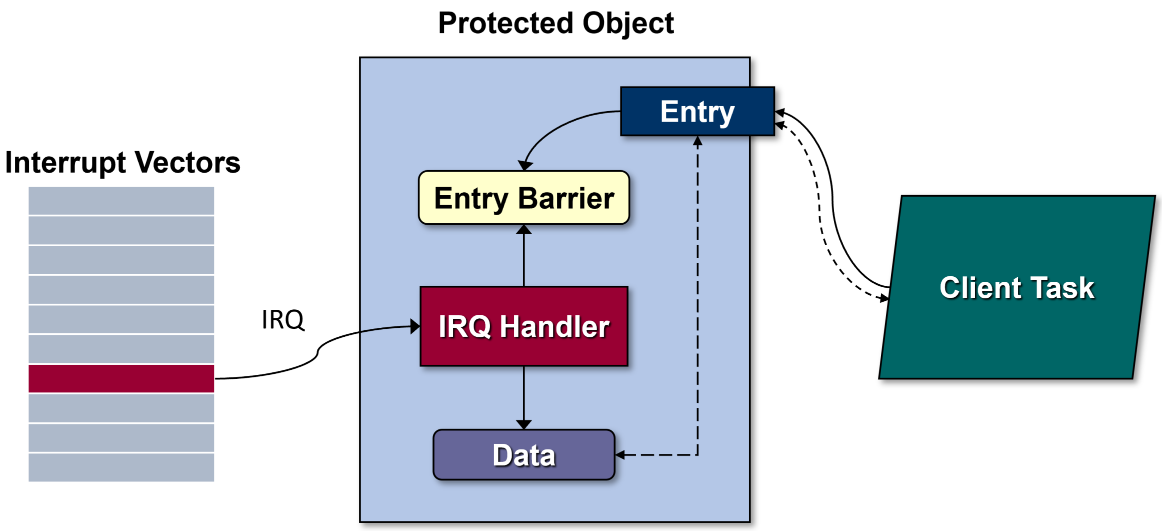
A related issue, perhaps requiring a run-time change, is the *secondary stack* allocation for interrupt handlers. The secondary stack is a common approach to implementing calls to functions that return values of unconstrained subtypes (usually, unconstrained array types, such as **String**). Because the result size is not known at the point of the call, using the primary call stack for holding the returned value is messy. The function's returned value would follow the stack space used for the call itself. But on return, only the call space is popped, leaving a *hole* in the stack because the value returned from the function remains on the stack. Therefore, another separate stack is commonly used for these functions. (GNAT does so.) The interrupt handler code could exhaust this allocation as well. The allocation amount is also specified in the run-time library source code. But, as with the situation above, the source code can be changed, in this case to avoid calling functions with unconstrained result types. The trade-off is whether that change is more costly than changing and rebuilding the run-time, as well as maintaining the change.

11.2.2 Task Notification Introduction

The first idiom implementation did not require notifying a task, but these next idiom implementations will do so. As we mentioned earlier, how the interrupt handler achieves this notification leads to two distinct idioms. Ultimately the difference between them is whether or not the interrupt handler must communicate with the task. In both cases the handler synchronizes with the task because of the notification required.

11.2.3 Task Notification With Communication

In this implementation the interrupt handler releases a task but also communicates with it when doing so. Therefore, a protected entry is used, and the entry parameters are the communication medium. The approach is depicted in the figure below:



The interrupt handler stores data within the PO and only enables the entry barrier when ready to either produce it or consume it via the entry parameters. The dashed lines in the figure represent this data flow.

By coincidence, this is the notification approach used in the idiom entry *Providing Component Access to Enclosing Record Objects* (page 71). In that implementation, client tasks call two entries to Put and Get single characters, so the data stored in the PO consists of those characters. We did not mention it there because we were focused on that other idiom, i.e., how to give visibility within a PO/task component to an enclosing record object.

Be sure to understand the code for the other idiom before exploring this one. We will repeat elided parts of the code and only discuss the parts relevant for this current idiom. Because we are focused now on the interrupt handling task notification, here is the full interrupt handler PO type declaration — `IO_Manager` — within the elided package declaration:

```
package Serial_IO.Interrupt_Driven is

   type Serial_Port ... is new Serial_IO.Device with private;

   overriding
   procedure Put (This : in out Serial_Port;  Data : HAL.UInt8)
     with Inline;

   overriding
   procedure Get (This : in out Serial_Port;  Data : out HAL.UInt8)
     with Inline;

private

   protected type IO_Manager
     (IRQ          : Interrupt_ID;
      IRQ_Priority : Interrupt_Priority;
      Port         : not null access Serial_Port)
```

(continues on next page)

(continued from previous page)

```
   with
      Interrupt_Priority => IRQ_Priority
   is

      entry Put (Datum : HAL.UInt8);

      entry Get (Datum : out HAL.UInt8);

   private

      Outgoing : HAL.UInt8;
      Incoming : HAL.UInt8;

      Incoming_Data_Available : Boolean := False;
      Transmission_Pending    : Boolean := False;

      procedure IRQ_Handler with Attach_Handler => IRQ;

    end IO_Manager;

   ...

end Serial_IO.Interrupt_Driven;
```

A protected object of type IO_Manager is given a discriminant value that designates the enclosing Serial_Port object because that Serial_Port has the USART device required to do the actual I/O. The other two discriminants are required for configuring the interrupt handler and attaching it to the interrupt hardware.

The two octets Outgoing and Incoming are the values sent and received via the interrupt handler's manipulation of the USART. (A USART doesn't receive characters, as such, and we're ignoring the fact that it may work with a 9-bit value instead.)

The two Boolean components Incoming_Data_Available and Transmission_Pending are used for the two barrier expressions. Their purpose is explained below.

The bodies of visible procedures Put and Get (shown below) call through to the interrupt manager's protected entries, also named Put and Get. Those entries block the callers until the interrupt manager is ready for them, via the entry barriers controlled by the interrupt handler.

```
with STM32.Device; use STM32.Device;

package body Serial_IO.Interrupt_Driven is

   ---------
   -- Put --
   ---------

   overriding
   procedure Put (This : in out Serial_Port;  Data : HAL.UInt8) is
   begin
      This.Controller.Put (Data);
   end Put;

   ---------
   -- Get --
   ---------

   overriding
   procedure Get (This : in out Serial_Port;  Data : out HAL.UInt8) is
```

(continues on next page)

(continued from previous page)

```
   begin
      This.Transceiver.Enable_Interrupts (Received_Data_Not_Empty);
      This.Controller.Get (Data);
   end Get;

   ----------------
   -- IO_Manager --
   ----------------

   protected body IO_Manager is

      -----------------
      -- IRQ_Handler --
      -----------------

      procedure IRQ_Handler is
      begin
         --  check for data arrival
         if Port.Transceiver.Status (Read_Data_Register_Not_Empty) and then
            Port.Transceiver.Interrupt_Enabled (Received_Data_Not_Empty)
         then  -- handle reception
            Get (Serial_IO.Device (Port.all), Incoming);
            -- call the Serial_IO.Device version!
            Await_Reception_Complete : loop
               exit when not
                 Port.Transceiver.Status (Read_Data_Register_Not_Empty);
            end loop Await_Reception_Complete;
            Port.Transceiver.Disable_Interrupts (Received_Data_Not_Empty);
            Port.Transceiver.Clear_Status (Read_Data_Register_Not_Empty);
            Incoming_Data_Available := True;
         end if;

         --  check for transmission ready
         if Port.Transceiver.Status (Transmission_Complete_Indicated) and then
            Port.Transceiver.Interrupt_Enabled (Transmission_Complete)
         then  -- handle transmission
            Put (Serial_IO.Device (Port.all), Outgoing);
            -- call the Serial_IO.Device version!
            Port.Transceiver.Disable_Interrupts (Transmission_Complete);
            Port.Transceiver.Clear_Status (Transmission_Complete_Indicated);
            Transmission_Pending := False;
         end if;
      end IRQ_Handler;

      ---------
      -- Put --
      ---------

      entry Put (Datum : HAL.UInt8) when not Transmission_Pending is
      begin
         Transmission_Pending := True;
         Outgoing := Datum;
         Port.Transceiver.Enable_Interrupts (Transmission_Complete);
      end Put;

      ---------
      -- Get --
      ---------

      entry Get (Datum : out HAL.UInt8) when Incoming_Data_Available is
      begin
```

(continues on next page)

(continued from previous page)

```
         Datum := Incoming;
         Incoming_Data_Available := False;
      end Get;

   end IO_Manager;

end Serial_IO.Interrupt_Driven;
```

Note how `IRQ_Handler` checks for which interrupt is active, possibly both, does whatever is necessary for that to be handled, and then sets the entry barriers accordingly. The barrier expression `Transmission_Pending` blocks `Put` callers until the current transmission, if any, completes. The barrier `Incoming_Data_Available` blocks `Get` callers until a character has been received and can be provided to the caller. The entry bodies copy the entry formal parameters to/from the internally stored characters and likewise set the entry barriers.

Note too how the body of procedure `Get` first enables the *received data available* interrupt before calling the entry. The body of the entry `Put` does something similar. They both work in concert with the handler procedure to manage the interrupts as required.

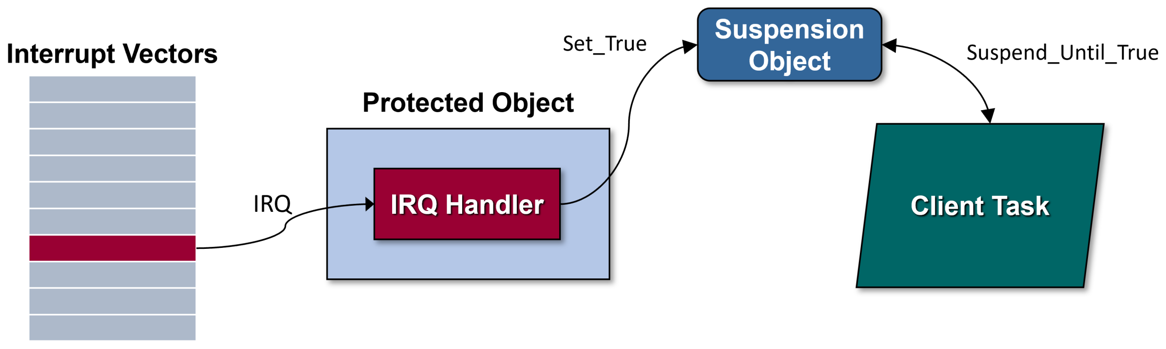
Using protected entries is ideal for this case because, after all, that is exactly what they are designed to do. Note that declaring multiple protected entries in a single protected type/object requires the Jorvik usage profile to be applied.

11.2.4 Task Notification Without Communication

In this implementation, the interrupt handler procedure is not required to communicate with the task. It only needs to synchronize with it, to release it.

Therefore, we can use a `Suspension_Object`: a language-defined, thread-safe *binary flag* type defined in package `Ada.Synchronous_Task_Control`. Objects of this type have two values: **`True`** and **`False`**, with **`False`** as the default initial value. There are two primary primitive operations: procedures `Suspend_Until_True` and `Set_True`. Procedure `Set_True` does just what you think it does. Procedure `Suspend_Until_True` suspends the caller (task) until the value of the specified argument becomes **`True`**, at which point the suspended task is allowed to continue execution. (Of course, if it was already **`True`** when `Suspend_Until_True` was called, the caller returns without suspending.) Critically, procedure `Suspend_Until_True` also sets the argument back to **`False`** before returning. As a result, those are the only two routines you're likely to need, although there are others.

The interrupt handler procedure in this approach simply calls `Set_True` for a `Suspension_Object` (an object of that type) visible to both the handler and the task. This arrangement is illustrated by the following figure:



The language requires the run-time library implementation to allow calls to `Set_False` and `Set_True` during any protected action, even one that has its ceiling priority in the `Interrupt_Priority` range, so this approach will work for interrupt handlers as well as tasks.

For our example we implement a facility for sending and receiving *messages* over a serial port, using interrupts. The design is similar to the implementation we just explored, and thus to the *Providing Component Access to Enclosing Record Objects* (page 71) idiom. In that implementation, however, only single characters were sent and received, whereas messages will consist of one or more characters. Although there are differences, we assume that you are familiar enough with that idiom's approach that we don't need to go into all the details of the serial I/O, the USART, or the interrupt handler within a PO. We'll focus instead of the differences due to this idiom.

In this version we want to notify a task when an entire message has been sent or received, not just a single character. We'll define a message as a `String` that has a message-specified logical terminator character, e.g., the *nul* character. Transmission will cease when the terminator character is encountered when sending a message object. Similarly, a message is considered complete when the terminator character is received. (The terminator is not stored in message content.)

In a sense the interrupt handler is again communicating with tasks, but not directly, so entry parameters aren't applicable. Therefore, a `Suspension_Object` component is appropriate. But instead of one `Suspension_Object` variable, each `Message` object will contain two: one for notification of new content receipt, and one for notification of successful content transmission.

For the sake of the Separation of Concerns principle, the type `Message` should be an ADT of its own, in a dedicated package:

```
with Serial_IO;                    use Serial_IO;
with Ada.Synchronous_Task_Control;  use Ada.Synchronous_Task_Control;

package Message_Buffers is

   type Message (Physical_Size : Positive) is tagged limited private;

   function Content (This : Message) return String;
   function Length (This : Message) return Natural;
   procedure Set (This : in out Message;  To : String) with
     Pre  => To'Length <= This.Physical_Size,
     Post => Length (This) = To'Length and Content (This) = To;
   ...
   function Terminator (This : Message) return Character;
   procedure Await_Transmission_Complete (This : in out Message);
   procedure Await_Reception_Complete (This : in out Message);
   procedure Signal_Transmission_Complete (This : in out Message);
   procedure Signal_Reception_Complete (This : in out Message);

private

   type Message (Physical_Size : Positive) is tagged limited record
      Content               : String (1 .. Physical_Size);
      Length                : Natural := 0;
      Reception_Complete    : Suspension_Object;
      Transmission_Complete : Suspension_Object;
      Terminator            : Character := ASCII.NUL;
   end record;

end Message_Buffers;
```

In essence, a `Message` object is just the usual *variable length string* abstraction with a known terminator and ways to suspend and resume clients using them. Note the two `Suspension_Object` components.

In this example the tasks to be notified are application tasks rather than second-level interrupt handlers. Client tasks can suspend themselves to await either transmission completion or reception completion. The `Message` procedures simply call the appropriate routines for the parameter's Suspension_Object components:

```
procedure Await_Transmission_Complete (This : in out Message) is
begin
   Suspend_Until_True (This.Transmission_Complete);
end Await_Transmission_Complete;
```

and likewise:

```
procedure Await_Reception_Complete (This : in out Message) is
begin
   Suspend_Until_True (This.Reception_Complete);
end Await_Reception_Complete;
```

The client task could look like the following, in this case the main program's environment task:

```
procedure Demo_Serial_Port_Nonblocking is

   Incoming : aliased Message (Physical_Size => 1024);  -- arbitrary size
   Outgoing : aliased Message (Physical_Size => 1024);  -- arbitrary size

   procedure Send (This : String) is
   begin
      Set (Outgoing, To => This);
      Start_Sending (COM, Outgoing'Unchecked_Access);
      Outgoing.Await_Transmission_Complete;
   end Send;

begin
   Initialize_Hardware (COM);
   Configure (COM, Baud_Rate => 115_200);

   Incoming.Set_Terminator (ASCII.CR);
   Send ("Enter text, terminated by CR.");
   loop
      Start_Receiving (COM, Incoming'Unchecked_Access);
      Incoming.Await_Reception_Complete;
      Send ("Received : " & Incoming.Content);
   end loop;
end Demo_Serial_Port_Nonblocking;
```

We don't show all the context clauses, for brevity, but one of the packages declares COM as the serial port. This demo doesn't exploit the nonblocking aspect because it does not perform any other actions before suspending itself after initiating sending and receiving. But it could do so, while the I/O is happening, only later suspending to await completion of the requested operation.

The interrupt handler procedure can signal both transmission and reception completion using the two other procedures:

```
procedure Signal_Transmission_Complete (This : in out Message) is
begin
   Set_True (This.Transmission_Complete);
end Signal_Transmission_Complete;

procedure Signal_Reception_Complete (This : in out Message) is
begin
   Set_True (This.Reception_Complete);
end Signal_Reception_Complete;
```

In this version of the Serial IO facility, the interrupt handler's enclosing protected type is the type Serial_Port itself, rather than a PO enclosed by a record type:

```
protected type Serial_Port
  (Device       : not null access Peripheral_Descriptor;
   IRQ          : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority)
with
    Interrupt_Priority => IRQ_Priority
is

   procedure Start_Sending (Msg : not null access Message);
   procedure Start_Receiving (Msg : not null access Message);

private

   Next_Out     : Positive;
   Outgoing_Msg : access Message;
   Incoming_Msg : access Message;

   procedure Handle_Transmission with Inline;
   procedure Handle_Reception    with Inline;
   procedure ISR with Attach_Handler => IRQ;

end Serial_Port;
```

Procedure ISR (*Interrupt Service Routine*) is the handler.

The two visible protected procedures, `Start_Sending` and `Start_Receiving`, are given non-null arguments when called (indirectly) by client tasks. Each argument is an access value designating a `Message` object declared by clients. The pointers are copied into the internal components, i.e., `Outgoing_Msg` and `Incoming_Msg`, for use by the interrupt handler procedure.

As with the earlier idiom above, there are multiple device interrupts, but they are all delivered on one external interrupt line. The handler procedure checks the status flags to see which interrupts are active and calls dedicated internal procedures accordingly. We don't need to see this infrastructure code again, so we can focus instead on one, the internal `Handle_Reception` procedure. The routine for transmitting is similar.

```
procedure Handle_Reception is
   Received_Char : constant Character :=
     Character'Val (Current_Input (Device.Transceiver.all));
begin
   if Received_Char /= Incoming_Msg.Terminator then
      Incoming_Msg.Append (Received_Char);
   end if;
   if Received_Char = Incoming_Msg.Terminator or else
      Incoming_Msg.Length = Incoming_Msg.Physical_Size
   then -- reception complete
      loop
         --  wait for device to clear the status
         exit when not Status (Device.Transceiver.all,
                               Read_Data_Register_Not_Empty);
      end loop;
      Disable_Interrupts (Device.Transceiver.all,
                          Source => Received_Data_Not_Empty);
      Incoming_Msg.Signal_Reception_Complete;
      Incoming_Msg := null;
   end if;
end Handle_Reception;
```

Note the call to `Signal_Reception_Complete` for the current Message object being received, designated by `Incoming_Msg`.

The alternative to a `Suspension_Object` is a parameterless protected entry that a task calls to suspend itself. That certainly works in general, but we would need two entries, hence Jorvik. But also, the `Suspension_Object` approach can have a little better performance because it does not have the functionality that a protected entry has.

Note that type `Suspension_Object` might very well be implemented as a protected type. On a uniprocessor target, protected object mutual exclusion can be implemented via priorities, so it won't make much difference. (GNAT's bare-board run-times use that mutual exclusion implementation approach, as well as the PO implementation of type `Suspension_Object`.)

11.3 Pros

In all three idioms, the approach is directly expressed, meets the requirements, and hides the implementation details. The implementations are efficient relative to their requirements, the only reasonable metric. In particular, `Suspension_Objects` are expected to be faster than protected entries, but only support synchronization, and only with one caller at a time — there's no queue. Nor do they support communication. Protected entries have no such restrictions and are reasonably efficient given their considerable additional capabilities.

11.4 Cons

None.

11.5 Relationship With Other Idioms

The idiom showing how to connect a PO or task to an enclosing record object was illustrated by an interrupt handler PO, but that idiom is not necessary. Indeed, we used a protected type directly in the last implementation.

11.6 What About Priorities?

The idiom expressions do not determine the actual priorities assigned to the protected objects containing the handler procedures, nor those of the notified tasks.

The language standard requires the priorities for interrupt handler POs to be in the range defined by the subtype `System.Interrupt_Priority`. Under the Ravenscar and Jorvik profiles they must also satisfy the Ceiling Priority Protocol requirements.

The target's interrupt hardware may dictate the specific handler priorities, or at least their floor values. You may be able to control those hardware priorities via the target board startup code.

But usually we have some freedom to choose, so what priorities should be assigned?

Often the values are arbitrary. However, a more rigorous approach may be required. A good guideline is that if you need to do a timing (schedulability) analysis for the application tasks' deadlines, you need to do it for the interrupt handlers' deadlines too. The same analyses can be used, i.e., response-time analysis, and the same priority assignment schemes, i.e., a shorter period gets a higher priority. (The interrupt *period* is the minimum interval between the interrupt occurrences.)

In addition, ensuring interrupt handler deadlines are met is part of ensuring the tasks meet their deadlines. That's because the interrupt handlers release the associated sporadic (event-driven) tasks for execution. A sporadic task triggered by a device (say) usually will have a deadline no greater than the next occurrence of the sensor-generated interrupt, that is, the interrupt period. The priority of the task will be set according to that period.

11.7 Notes

1. The *traditional* expression for an interrupt handler, i.e., a procedure, is allowed by the language as a vendor-defined extension. However, there will likely be language-oriented restrictions applied to those procedures, due to the context. That's true of other languages as well.
2. You shouldn't assign interrupt handler (PO) priorities by semantic importance, just as you shouldn't do so for task priorities. More *important* interrupt handlers shouldn't necessarily be assigned more urgent priorities.

CHAPTER

TWELVE

REDUCING OBJECT CODE FROM GENERIC PACKAGE INSTANTIATIONS

12.1 Motivation

Generic unit instantiations are often, but not always, implemented by an Ada compiler as a form of *macro expansion*. In this approach, the compiler produces separate, dedicated object code for every instantiation. The macro expansion approach can produce better run-time performance but can result in large total object code size for the executable when there are many instances, especially when the generic packages instantiated contain a lot of unit declarations. For example, the generic I/O packages contained within package `Ada.Text_IO` are themselves relatively large.

The alternative compiler implementation approach is *code-sharing*, in which distinct instantiations of a given generic unit are implemented with shared object code in a single module.

Clearly, sharing the object code can reduce the total size, but code-sharing can be very complicated to implement, depending on the generic unit itself. For a trivial example, consider the following package:

```
generic
package P is
   Error : exception;
end P;
```

The semantics of the language require that every instantiation of generic package P be a distinct package, as if each instance was instead written explicitly as a non-generic package (at the point of instantiation) with the instance name. As a result, each package instance declares an exception, and these exceptions must be treated as distinct from each other. A code-sharing implementation must maintain that distinction with one object code module.

In the example above, there are no generic formal parameters, nor other declarations within the package declarative part besides the exception, because they are not necessary for that example. However, generic formal parameters can be a problem for code-sharing too. For example, consider this generic package:

```
generic
   Formal_Object : in out Integer;
package P is
   --  ...
end P;
```

This generic package has a generic formal object parameter with mode **`in out`**. (We chose type **`Integer`** purely for convenience.) That specific mode can cause a similar problem as seen in the exception example, because the mode allows the generic package to update the generic actual object passed to it. The shared object code must keep track of which object is updated during execution.

Therefore, when writing the application source code that instantiates generic packages, developers should do so in a manner that minimizes the amount of object code that might result.

12.2 Implementation(s)

The application source code should be written in a manner that shares the instantiations themselves, when possible, thereby reducing the number of instantiations that exist.

For example, let's say that several units in the application code require the ability to do I/O on some floating-point type. For simplicity, let's say that this is a type named Real, declared in a package named Common. Here is a declaration for an example package body that requires the I/O capability:

```
with Ada.Text_IO, Common;
package body User1 is
    package Real_IO is new Ada.Text_IO.Float_IO (Common.Real);
    -- ...
end User1;
```

That's certainly legal, and works, but we've said that several units require I/O for type Real. Let's say there are in fact twenty such units. They all do something similar:

```
with Ada.Text_IO, Common;
package body User20 is
    package Real_IO is new Ada.Text_IO.Float_IO (Common.Real);
    --  ...
end User20;
```

As a result, the application has twenty instantiations (at least) of Ada.Text_IO.Float_IO. There will be instances named User1.Real_IO, User2.Real_IO, and so on, up to User20. Real_IO. The fact that the local names are all Real_IO is irrelevant.

If the compiler happens to use the macro-expansion implementation, that means the application executable will have twenty copies of the object code defined by the generic Float_IO. For example, GNAT performs some internal restructuring to avoid this problem for these specific language-defined generic units, but not for application-defined generics.

Instead, we can simply instantiate the generic at the library level:

```
with Ada.Text_IO, Common;
package Real_IO is new Ada.Text_IO.Float_IO (Common.Real);
```

Because the instantiation occurred at the library level, the resulting instance is declared at the library level, and can therefore be named in a "with_clause" like any other library package.

```
with Real_IO;
package body User1 is
    --  ...
end User1;
```

Each client package can use the same instance via the with_clause, and there's only one instance so there's only one copy of the object code.

12.3 Pros

The total object code size is reduced, compared to the alternative of many local instantiations.

12.4 Cons

What would otherwise be an implementation detail hidden from clients can now become visible to them because a (public) library unit can be named in with_clause by any other unit. As a result, this approach should not be used in all cases, not even as a default design approach. Restricting the visibility of the instance may be more important than the amount of object code it contributes. Hiding implementation artifacts allows more freedom to change the implementation without requiring changes to client code.

12.5 Relationship With Other Idioms

None.

12.6 Notes

1. The reader should understand that this issue is not about the number of subprograms within any given package, whether or not the package is a generic package. In the past, some linkers included the entire object code for a given package (instance or not), regardless of the number of subprograms actually used from that package in the application code. That was an issue with reusable library code, for example packages providing mathematical functions. Modern linkers can be told not to include those subprograms not called by the application. For example, with gcc, the compiler can be told to put each subprogram in a separate section, and then the linker can be told to only include in the executable those sections actually referenced. (Data declarations can be reduced that way as well.)

CHAPTER

THIRTEEN

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

13.1 Motivation

In order for the expected semantics to be obtained, some types require clients to follow a specific protocol when calling the type's operations. Furthermore, failing to follow the protocol can cause system-wide ill effects.

For example, concurrency abstractions such as mutexes provide the mutually exclusive access necessary to prevent the race conditions that arise when competing concurrent threads access shared resources. These mutex objects must be 1) both acquired and released, 2) by every thread accessing that shared resource, 3) at the right places in the source code, and 4) in the proper order. Failure to acquire the mutex prior to accessing the shared resource leads to race conditions, and failure to release it can lead to deadlocks. Ensuring the mutex is released is complicated by the possibility of exceptions raised after the lock is acquired.

Although concurrency is a prime example, the issue is general in nature. We will continue with the concurrency context for the sake of discussion.

Like the classic *monitor* concept ([50],[51],[52]) on which they are based, Ada defines a protected object (PO) as a concurrency construct that is higher-level and more robust than mutexes and semaphores. Those advantages accrue because the bodies of the protected operations are only responsible for implementing the functional requirements. The underlying run-time library is responsible for implementing the mutually exclusive access, and also thread management. As a result, the source code is much simpler and is robust even in the face of exceptions.

(In the works cited above, Hoare's contribution[52] was equally important, but Hansen's contributions[50],[51] were reified in Concurrent Pascal[48], a concrete programming language.)

However, a protected object is not always appropriate. Consider an existing sequential program that makes calls to visible procedures provided by a package:

```ada
package P is

   procedure Operation_1;

   procedure Operation_2;

   --  ...

end P;
```

Inside the package body are one or more state variables that are manipulated by the procedures (i.e., as in an *Abstract Data Machine* (page 17)):

[50] P. B. Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977.
[51] P. B. Hansen. *Operating System Principles*. Prentice-Hall, 1973.
[52] C. A. R. Hoare. Monitors: an operating system structuring concept. *Comm. ACM*, 17(10):549–557, 1974.
[48] https://en.wikipedia.org/wiki/Concurrent_Pascal

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

package body P is

   State : Integer := 0;

   procedure Operation_1 is
   begin
      State := State + 1;  -- for example...
      Put_Line ("State is now " & State'Image);
   end Operation_1;

   procedure Operation_2 is
   begin
      State := State - 1;  -- for example...
      Put_Line ("State is now " & State'Image);
   end Operation_2;

   --  ...

end P;
```

This design is reasonable in a strictly sequential caller context. But if new application requirements are such that multiple tasks will be calling these procedures asynchronously, there is a problem. The package-level variable `State` will be subject to race conditions because it is (indirectly) shared among the calling tasks. Race conditions tend to be Heisenbugs because they are timing-dependent, so they can be exceedingly difficult to identify and expensive to debug.

In response to the new requirements, we could declare a protected object within the package body and move the declaration of `State` into that PO. In addition, we would declare two protected procedures corresponding to `Operation_1` and `Operations_2`. The two new protected procedure bodies would do what the original procedures did, including accessing and updating `State`. The original procedures — still presented to clients — would now call these new protected procedures:

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

package body P is

   protected Threadsafe is
      procedure Operation_1;
      procedure Operation_2;
   private
      State : Integer := 0;
   end Threadsafe;

   protected body Threadsafe is

      procedure Operation_1 is
      begin
         State := State + 1;  -- for example...
         Put_Line ("State is now " & State'Image);
      end Operation_1;

      procedure Operation_2 is
      begin
         State := State - 1;  -- for example...
         Put_Line ("State is now " & State'Image);
      end Operation_2;
```

(continues on next page)

(continued from previous page)

```
   end Threadsafe;

   procedure Operation_1 is
   begin
      Threadsafe.Operation_1;
   end Operation_1;

   procedure Operation_2 is
   begin
      Threadsafe.Operation_2;
   end Operation_2;

   --  ...

end P;
```

As a result, there can be no race conditions on State, and the source code is both simple and robust.

We could put the PO in the package spec and then clients could call the protected object's operations directly, but changing all the clients could be expensive.

However, this new design is not portable because the two Threadsafe protected procedure bodies both call a potentially blocking operation, in this case Ada.Text_IO.Put_Line. Erroneous execution is the result. It might work as intended when executed, or it might do something else, or, if detected, Program_Error will be raised. On a run-time library built on top of an operating system, it may work as intended because the OS may provide thread locking mechanisms that the run-time library can use. In that case a blocking operation just suspends the caller thread's execution temporarily without releasing the PO lock. Although the blocking operation would allow some other caller task to be dispatched, no other caller could acquire that same PO lock, so race conditions are prevented within that PO. When the blocking operation returns, the protected procedure body can continue executing, still holding the lock. However, on a run-time library that does not use locks for mutual exclusion — it can use priorities, in particular — another caller to that same PO could access the enclosed variables while the first caller is blocked, thus breaking the mutually exclusive access guarantee.

Calling an I/O operation is not all that strange here, and those are not the only potentially blocking operation defined by the language.

Note that moving the calls to Put_Line out of the PO procedure bodies, back to the regular procedure bodies that call those PO procedures, would solve the portability problem but would not work functionally. There would be no guarantee that, during execution, the call to Put_Line would immediately follow the execution of the protected procedure called immediately before it in the source code. Hence the printed text might not reflect the current value of the State variable.

As a consequence, we must fall back to manually acquiring and releasing an explicit lock. For example, we could declare a lock object at the package level, as shown below, and have each operation acquire and release it:

```
with GNAT.Semaphores;  use GNAT.Semaphores;
with Ada.Text_IO;      use Ada.Text_IO;

package body P is

   subtype Mutual_Exclusion is Binary_Semaphore
    (Initially_Available => True,
     Ceiling             => Default_Ceiling);

   Lock : Mutual_Exclusion;
```

(continues on next page)

(continued from previous page)

```ada
   State : Integer := 0;

   procedure Operation_1 is
   begin
      Lock.Seize;
      State := State + 1;  -- for example...
      Put_Line ("State is now" & State'Img);
      Lock.Release;
   exception
      when others =>
         Lock.Release;
         raise;
   end Operation_1;

   procedure Operation_2 is
   begin
      Lock.Seize;
      State := State - 1;  -- for example...
      Put_Line ("State is now" & State'Img);
      Lock.Release;
   exception
      when others =>
         Lock.Release;
         raise;
   end Operation_2;

end P;
```

The subtype `Mutual_Exclusion` is just a binary semaphore with the discriminant set so that any object of the subtype is initially available. You can assume it is a protected type with classic binary semaphore semantics. See package `GNAT.Semaphores` for the details. The ceiling discriminant isn't important here, but we must set them all if we set any of them.

This design works, but the resulting code is clearly more complex and less robust than the PO approach.

13.2 Implementation

Our implementation uses an explicit global lock (a mutex), as above, but reintroduces automatic lock acquisition and release.

To achieve that automation, we leverage the language-defined object *lifetime* rules. These rules specify that an object is initialized when it is created and finalized when it is about to be destroyed. Initialization and finalization may be null operations, and thus absent from the object code, but application developers can define explicit initialization and finalization operations. When defined, these operations are called automatically by the underlying implementation, during the object's lifetime.

We will use the object initialization operation to seize the global lock and the object finalization operation to release it. The object lifetime rules will ensure that the lock's operations are called at the necessary times, thereby providing the required mutually exclusive access. In addition, the rules will ensure that the lock will be released even if an exception is raised in the bracketed application code.

Developers may be familiar with this approach under the name Resource Acquisition Is Initialization[49]. Another name for this technique is *Scope-Bound Resource Management* because of the initialization and finalization steps invoked upon scope entry and exit.

[49] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Therefore, we will create a new type with user-defined initialization and finalization operations. We name this new type `Lock_Manager` because the type provides a wrapper for locks, rather than being a lock directly. Object creation and destruction will invoke the initialization and finalization routines, automatically.

Because they are wrappers for locks, each object of this type will reference a distinct lock object so that the initialization and finalization operations can manipulate that lock object. We use an access discriminant to designate that lock. By doing so, we decouple the new type from the specific lock, and thus from the application code. Otherwise, the new facility would not be reusable.

The resulting relationship between the global shared lock and the local object will be as follows:

```ada
Lock : Mutual_Exclusion;

procedure Op is
   LM : Lock_Manager (<pointer to Lock>)
   --  <initialization automatically called for LM>
begin
   --  ... sequence of statements for Op
   --  <finalization called for LM>
end Op;
```

The language rules specify that a subprogram's local declarative part is elaborated prior to the execution of that subprogram's sequence of statements. During that elaboration, objects are created and initialized. The object creation for `LM` precedes the sequence of statements in the procedure body for `Op`, so the designated lock will be acquired prior to the shared resource use within that body.

Similarly, the rules specify that finalization occurs when an object is about to cease to exist, in this case because the local object `LM` goes out of scope. That won't happen until the end of the sequence of statements is reached for `Op`, in the normal case, so finalization will ensure that the lock is released after any possible reference in `Op`'s statement sequence. The run-time will also invoke finalization in the face of exceptions because exceptions also cause the scope to be exited.

To define the `Lock_Manager` type, we declare it in a separate package as a tagged limited private type with a discriminant designating a `Mutual_Exclusion` object:

```ada
type Lock_Manager (Lock : not null access Mutual_Exclusion) is
   tagged limited private;
```

We make it a limited type because copying doesn't make sense semantically for `Lock_Manager` objects.

In addition, during optimization the compiler is allowed to remove unreferenced objects of non-limited types. As you saw above in procedure `Op`, there will be no explicit references to the object `LM`, so making the type limited prevents that unwanted optimization.

Only *controlled* types support user-defined initialization and finalization operations (as of Ada 2022). Therefore, in the package private part the type is fully declared as a controlled type derived from `Ada.Finalization.Limited_Controlled`, as shown below. We hide the fact that the type will be controlled because we don't intend `Initialize` and `Finalize` to be called manually by clients.

```ada
type Lock_Manager (Lock : not null access Mutual_Exclusion) is
   new Ada.Finalization.Limited_Controlled with null record;
```

No additional record components are required, beyond the access discriminant.

Immediately following the type declaration, we declare overriding versions of the inherited procedures `Initialize` and `Finalize`:

```
overriding procedure Initialize (This : in out Lock_Manager);
overriding procedure Finalize   (This : in out Lock_Manager);
```

These are the operations called automatically by the implementation.

The full package spec is as follows:

```
with Ada.Finalization;
with GNAT.Semaphores; use GNAT.Semaphores;

package Lock_Managers is

  subtype Mutual_Exclusion is Binary_Semaphore
    (Initially_Available => True,
     Ceiling             => Default_Ceiling);

   type Lock_Manager (Lock : not null access Mutual_Exclusion) is
      tagged limited private;

private

   type Lock_Manager (Lock : not null access Mutual_Exclusion) is
      new Ada.Finalization.Limited_Controlled with null record;

   overriding procedure Initialize (This : in out Lock_Manager);
   overriding procedure Finalize (This : in out Lock_Manager);

end Lock_Managers;
```

The fact that there are no visible primitive operations tells the reader that this is a somewhat different *ADT* (page 11). The most useful thing a client can do with such a type is to declare objects, but that's exactly what we want.

Each overridden procedure simply references the lock designated by the formal parameter's Lock discriminant:

```
package body Lock_Managers is

   ----------------
   -- Initialize --
   ----------------

   overriding procedure Initialize (This : in out Lock_Manager) is
   begin
      This.Lock.Seize;
   end Initialize;

   --------------
   -- Finalize --
   --------------

   overriding procedure Finalize (This : in out Lock_Manager) is
   begin
      This.Lock.Release;
   end Finalize;

end Lock_Managers;
```

The resulting user code is almost unchanged from the original sequential code:

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

(continues on next page)

(continued from previous page)

```
package body P is

   State : Integer := 0;

   Lock : aliased Mutual_Exclusion;

   procedure Operation_1 is
      LM : Lock_Manager (Lock'Access) with Unreferenced;
   begin
      State := State + 1;  -- for example...
      Put_Line ("State is now" & State'Img);
   end Operation_1;

   procedure Operation_2 is
      LM : Lock_Manager (Lock'Access) with Unreferenced;
   begin
      State := State - 1;  -- for example...
      Put_Line ("State is now" & State'Img);
   end Operation_2;

end P;
```

The aspect `Unreferenced` tells the compiler that no references in the source code are expected. That has two effects during compilation. First, warnings about the lack of references in the source code are disabled. Ordinarily we'd want those warnings because an unreferenced object usually indicates a coding error. That warning would be *noise* for objects of this type. But by the same token, the compiler will issue a warning if some explicit reference is present, perhaps added much later in the project lifetime.

13.3 Pros

Race conditions are precluded, the client code is simpler than direct manual calls, and the code is robust, especially concerning exceptions. These advantages are significant, given the cost in engineering time to debug the errors this design prevents.

13.4 Cons

The lock is global, so all calls go through it. Hence all calls are sequential, even if some could run concurrently. In the above example that's exactly as required, but in other situations it might be unnecessarily limiting.

Compared to the manual call approach, the run-time cost for keeping track of objects to be finalized could be non-trivial. That's likely true in any language.

13.5 Relationship With Other Idioms

None.

13.6 Notes

- The name for a similar type in the C++ Boost library is Scoped_Lock, as is the Ada type in the GNAT library package `GNATColl.Locks`. I used Scope_Lock in AdaCore's

Gem #70[53].

- I didn't invent the name `Scope_Lock` or the Ada implementation, but I don't recall where I first saw it many years ago. My apologies to that author.
- I consider the name `Lock_Manager` or something similar to be better, since objects of the type are wrappers for locks, not locks themselves. Indeed, in C++ 2011 the name is `lock_guard`.

[53] P. Rogers. *Gem #70: The Scope Locks Idiom*. https://www.adacore.com/gems/gem-70, 2009.

CHAPTER

FOURTEEN

USING STREAMS FOR API FLEXIBILITY

14.1 Motivation

Software interfaces for hardware devices usually support only primitive data types. Communications devices such as serial I/O ports and network adapters are good examples. Their device drivers provide an API for sending and receiving individual 8-bit or 9-bit numeric quantities, or sequences of these. Software clients — either the application, or higher-level layers of device interfaces — may want to send and receive more complex data types. If so, how can the device driver support them?

We certainly don't want the clients to do unchecked conversions everywhere. That's error-prone, it prevents the compiler from checking the usage, and makes clients responsible for what should be an internal implementation detail.

This is a general issue, not specific to communications hardware, but we will discuss it in that context because of the familiarity of such devices.

The traditional approach is for the device driver's I/O routines to have two parameters for this purpose: an address and a length. The address indicates the first byte of the client value to be sent or received, and the length indicates how many bytes are involved. Values of any type can be sent or received using this interface, but it's a very unsafe / unrobust approach. Developers could pass the wrong starting address, or specify the wrong length, thus potentially transmitting only part of the intended value or including part of some wholly unrelated object. Moreover, developers could pass the address and length of some object that is not of the type expected on the other end of the connection. After all, Ada allows us to take the address of just about anything. The effect on the receiver would be difficult to predict. These mistakes are very expensive to locate, and the compiler cannot help.

We need a type-safe approach for sending and receiving higher-level types so that the compiler can catch our mistakes. After all, preventing coding errors is much cheaper than fixing them later.

14.2 Implementation

We will explore the possibilities using a concrete USART (Universal Synchronous/Asynchronous Receiver Transmitter)[54] defined in the Ada Drivers Library (ADL)[55]. A USART is the physical communications device underlying what is commonly referred to as a *serial port*. That name reflects the fact that the device transmits and receives data serially, as opposed to in parallel.

The ADL provides packages representing specific microcontrollers as well as their on-chip peripherals, including USARTs, timers, DMA controllers, and so forth. Each kind of peripheral is represented by a dedicated *Abstract Data Type (ADT)* (page 11). The following is the elided ADT declaration for USARTs on STM32 microcontrollers:

[54] https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter
[55] https://github.com/AdaCore/Ada_Drivers_Library

```
--  ...

package STM32.USARTs is

   type USART is ... private;

   --  ...
   procedure Receive  (This : USART;         Data : out UInt9);
   procedure Transmit (This : in out USART;  Data : UInt9);

   function Tx_Ready (This : USART) return Boolean;
   function Rx_Ready (This : USART) return Boolean;

   --  ...

private
   --  ...
end STM32.USARTs;
```

Note the formal parameter data type for both Receive and Transmit, i.e., a single 9-bit unsigned numeric value. Although a client might actually want to send and receive such values directly, that's probably not the case.

Our implementation is structured as a package hierarchy rooted at package Serial_IO. (This implementation is part of an example in the ADL — see *Note #1* (page 120) below.) This root package declares a record type and routines that are common to any implementation. The type, named Peripheral_Descriptor, contains a component named Transceiver that will designate the actual on-chip USART device being driven. The other record components are required for connecting that device to the external world.

Type Peripheral_Descriptor is hardware-specific and is therefore not defined as an ADT. Instead, the package uses the *Groups of Related Program Units* (page 8) idiom.

```
with STM32;         use STM32;
with STM32.GPIO;    use STM32.GPIO;
with STM32.USARTs;  use STM32.USARTs;

package Serial_IO is

   type Peripheral_Descriptor is record
      Transceiver    : not null access USART;
      Transceiver_AF : GPIO_Alternate_Function;
      Tx_Pin         : GPIO_Point;
      Rx_Pin         : GPIO_Point;
   end record;

   procedure Initialize_Hardware (Device : Peripheral_Descriptor);
   --  enable clocks, configure GPIO pins, etc.

   procedure Configure
     (Device    : access USART;
      Baud_Rate : Baud_Rates;
      Parity    : Parities     := No_Parity;
      Data_Bits : Word_Lengths := Word_Length_8;
      End_Bits  : Stop_Bits    := Stopbits_1;
      Control   : Flow_Control := No_Flow_Control);

   --  ...

end Serial_IO;
```

Procedure Configure is a convenience routine, provided because a specific sequence of driver calls is required in order to set the individual parameters.

Clients will call these procedures directly to set up the STM32 on-chip USART device.

Our implementation will consist of an ADT named `Serial_Port`, and a means for sending and receiving values of higher-level types via `Serial_Port` objects. Type `Serial_Port` will be a wrapper for the device driver's USART type. Therefore, the type `Serial_Port` will have an access discriminant designating a USART:

```
type Serial_Port (Device : not null access USART) is ...
```

Using this `Device` discriminant a `Serial_Port` object can reference the wrapped physical USART in order to send and receive values via that hardware device.

With that introduction in place, we can consider the possible approaches to sending and receiving values of higher-level types via `Serial_Port` objects.

The canonical Ada approach consists of a generic package with a generic formal parameter type. That formal type represents a client-specific type to be sent or received. Clients instantiate the generic for every client-defined type necessary.

The generic package would look like the following:

```
generic
   type Client_Data (<>) is limited private;
package Client_IO is

   procedure Send
     (This     : in out Serial_Port;
      Outgoing : Client_Data);

   procedure Receive
     (This     : in out Serial_Port;
      Incoming : out Client_Data);

end Client_IO;
```

In the procedure bodies, the values of the `Incoming` or `Outgoing` parameters would be converted to or from bytes as necessary and sent or received via the USART designated by `This.Device`.

This approach supports as many client-level types as required, including limited types. It is type-safe so the compiler can catch errors in the type(s) being sent and received. In addition, the low-level implementation details, such as unchecked conversions, are hidden inside the generic package body.

Moreover, the approach is independent of other design considerations, such as whether callers wait for completion of the invoked I/O operation. The bodies of the generic procedures can be implemented to provide the expected behavior.

However, this approach is somewhat heavy because the generic package must be instantiated for every client type to be supported. If there are many such types, there will be many instantiations. Not only is that more lines of source code, but also more object code because most Ada implementations do not support code-sharing for generic instantiations. But that said, the need for a given client to send and receive values of many different types is not typical.

However, there is a more concise approach possible, for both the driver and client source code. This alternative approach leverages the flexibility of streams and stream attributes. Using streams allows the wrapper to support an unlimited number of distinct client types, with no additional source code required per type.

Recall that the stream attributes are callable routines whose first parameter is an access value designating some stream object. The formal parameter type is access-to-class-wide, so any stream object is allowed. For example, the notional specification for the 'Output attribute looks like this, for some subtype S of type T:

```
procedure S'Output
  (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T);
```

Therefore, the fundamental approach will be to declare the Serial_Port ADT as a stream type. Clients can then send and receive values simply by invoking the stream attributes, passing (access to) Serial_Port objects as the first parameter to the invocations.

The Serial_Port ADT will be defined in package Serial_IO.Streaming. Given that, a client can declare a Serial_Port object like so:

```
with STM32.Device;
with Serial_IO.Streaming; use Serial_IO;

package Peripherals_Streaming is

   --  the USART selection is arbitrary but the AF number and the
   --  pins must be those required by that USART
   Peripheral : constant Serial_IO.Peripheral_Descriptor :=
                  (Transceiver    => STM32.Device.USART_1'Access,
                   Transceiver_AF => STM32.Device.GPIO_AF_USART1_7,
                   Tx_Pin         => STM32.Device.PB6,
                   Rx_Pin         => STM32.Device.PB7);

   COM : aliased Streaming.Serial_Port (Peripheral.Transceiver);

end Peripherals_Streaming;
```

In the above, Peripheral is an object that describes a specific USART on the SMT32 Discovery Board microcontroller, along with the values necessary to connect that specific USART to the external environment. The Peripheral variable will be passed to a call to Initialize_Hardware. Similarly, COM is an object that wraps the USART designated by Peripheral.

Because Serial_Port will be a stream type (it will be in the derivation class rooted at Root_Stream_Type), COM will be a streaming object that we can pass to invocations of the stream attributes. For example, to send a **String** value via COM we could write:

```
String'Output (COM'Access, "Hello World");
```

To send an **Integer** value:

```
Integer'Write (COM'Access, 42);
```

To receive an **Integer** value into the **Integer** object X:

```
Integer'Read (COM'Access, X);
```

That's all clients must do to send and receive values via the USART wrapped by COM. They could do the same for floating-point types, record types, and so on. Objects of any type with the streaming attributes defined can be sent or received, and in any order.

To make Serial_Port a stream type, the declaration visibly extends Ada.Streams.Root_Stream_Type:

```
type Serial_Port (Device : not null access USART) is
  new Ada.Streams.Root_Stream_Type with private;
```

This is *Interface Inheritance* (page 61) so that clients can treat Serial_Port as a stream type. The private extension hides implementation details that we'll describe momentarily.

As a concrete extension of Root_Stream_Type we must declare overridings for procedures Read and Write:

```
overriding
procedure Read
  (This   : in out Serial_Port;
   Buffer : out Ada.Streams.Stream_Element_Array;
   Last   : out Ada.Streams.Stream_Element_Offset);

overriding
procedure Write
  (This   : in out Serial_Port;
   Buffer : Ada.Streams.Stream_Element_Array);
```

Stream_Element_Array is an unconstrained array type with Stream_Element as the array component. Stream_Element is an unsigned numeric type corresponding to a machine storage element, e.g., a byte.

Procedure Write inserts these array components into the designated stream. Procedure Read consumes the array components from the stream and includes a parameter indicating the index of the last component assigned.

These two procedures are called by the various stream attributes' implementations, not by clients. For example, consider again the call to 'Output:

```
String'Output (COM'Access, "Hello World");
```

The call dispatches to our overriding of procedure Write because the first parameter designates our specific stream object.

For both procedures the array components hold the serialized representation of the value read from, or to be written to, the stream. For procedure Write, the array contains the stream-oriented representation of the client value, e.g., the "Hello World!" passed to **String**'Output. For procedure Read, the array contains the value consumed from the stream that will be converted into the client type, e.g., type **Integer** for a call to **Integer**'Read, and loaded into the client variable.

Note that the two procedures are only responsible for reading or writing the array components from/to the specified stream. Conversions between the types in the clients' attribute invocations and type Stream_Element_Array are not their responsibility. That metamorphosis is handled automatically by the language-defined attributes' implementations. It's a nice separation of concerns.

Here then is the full package declaration for the Serial_Port ADT:

```
with Ada.Streams;
with Ada.Real_Time; use Ada.Real_Time;

package Serial_IO.Streaming is
   pragma Elaborate_Body;

   type Serial_Port (Device : not null access USART) is
     new Ada.Streams.Root_Stream_Type with private;

   procedure Set_Read_Timeout
     (This : in out Serial_Port;
      Wait : Time_Span);
   --  Stream attributes that call Read (below) can either wait
   --  indefinitely or can be set to return any current values
   --  received after a given interval. If the value Time_Span_Last
   --  is passed to Wait, the effect is essentially to wait forever,
   --  i.e., blocking. That is also the effect if this routine is
   --  never called.
```

(continues on next page)

(continued from previous page)

```
   overriding
   procedure Read
     (This   : in out Serial_Port;
      Buffer : out Ada.Streams.Stream_Element_Array;
      Last   : out Ada.Streams.Stream_Element_Offset);

   overriding
   procedure Write
     (This   : in out Serial_Port;
      Buffer : Ada.Streams.Stream_Element_Array);

private

   type Serial_Port (Device : access USART) is
     new Ada.Streams.Root_Stream_Type with record
       Timeout : Time_Span := Time_Span_Last;
     end record;

   procedure Await_Send_Ready (This : access USART) with Inline;

   procedure Await_Data_Available
     (This      : access USART;
      Timeout   : Time_Span := Time_Span_Last;
      Timed_Out : out Boolean)
   with Inline;

   use Ada.Streams;

   function Last_Index
     (First : Stream_Element_Offset;
      Count : Long_Integer)
      return Stream_Element_Offset
   with Inline;

end Serial_IO.Streaming;
```

Prior to procedures Read and Write, the package declares a procedure for controlling a timeout associated with a Serial_Port stream. This timeout controls how long procedure Read should wait for input to be available in the stream. The default is to wait for what amounts to forever. Note that the timeout applies both to the case of some input received, and none received.

In the package private part we see that the type extension contains the record component named Timeout, with the initial value providing the default. That's the only other record component required, besides the discriminant.

Additional implementation-oriented routines are also declared there, rather than in the package body, for the sake of any child packages that might be declared in the future. Note in particular the two that await I/O completion, as this is a blocking implementation.

Here is the package body:

```
with HAL;

package body Serial_IO.Streaming is

   ----------------------
   -- Set_Read_Timeout --
   ----------------------
```

(continues on next page)

(continued from previous page)

```
   procedure Set_Read_Timeout
     (This : in out Serial_Port;
      Wait : Time_Span)
   is
   begin
      This.Timeout := Wait;
   end Set_Read_Timeout;

   ----------------------
   -- Await_Send_Ready --
   ----------------------

   procedure Await_Send_Ready (This : access USART) is
   begin
      loop
         exit when This.Tx_Ready;
      end loop;
   end Await_Send_Ready;

   --------------------------
   -- Await_Data_Available --
   --------------------------

   procedure Await_Data_Available
     (This      : access USART;
      Timeout   : Time_Span := Time_Span_Last;
      Timed_Out : out Boolean)
   is
      Deadline : constant Time := Clock + Timeout;
   begin
      Timed_Out := True;
      while Clock < Deadline loop
         if This.Rx_Ready then
            Timed_Out := False;
            exit;
         end if;
      end loop;
   end Await_Data_Available;

   ----------------
   -- Last_Index --
   ----------------

   function Last_Index
     (First : Stream_Element_Offset;
      Count : Long_Integer)
      return Stream_Element_Offset
   is
   begin
      if First = Stream_Element_Offset'First and then Count = 0 then
         --  although we intend to return First - 1, we cannot
         raise Constraint_Error;  -- per AI95-227
      else
         return First + Stream_Element_Offset (Count) - 1;
      end if;
   end Last_Index;

   ----------
   -- Read --
   ----------
```

(continues on next page)

(continued from previous page)

```
   overriding
   procedure Read
     (This   : in out Serial_Port;
      Buffer : out Ada.Streams.Stream_Element_Array;
      Last   : out Ada.Streams.Stream_Element_Offset)
   is
      Raw       : HAL.UInt9;
      Timed_Out : Boolean;
      Count     : Long_Integer := 0;
   begin
      Receiving : for K in Buffer'Range loop
         Await_Data_Available (This.Device, This.Timeout, Timed_Out);
         exit Receiving when Timed_Out;
         This.Device.Receive (Raw);
         Buffer (K) := Stream_Element (Raw);
         Count := Count + 1;
      end loop Receiving;
      Last := Last_Index (Buffer'First, Count);
   end Read;

   -----------
   -- Write --
   -----------

   overriding
   procedure Write
     (This   : in out Serial_Port;
      Buffer : Ada.Streams.Stream_Element_Array)
   is
   begin
      for Next of Buffer loop
         Await_Send_Ready (This.Device);
         This.Device.Transmit (HAL.UInt9 (Next));
      end loop;
   end Write;

end Serial_IO.Streaming;
```

Procedure Read polls the wrapped USART device, continually, until a byte becomes available or the timeout is reached. Procedure `Await_Data_Available` performs this timed polling. Polling without relinquishing the processor is extremely questionable on a main CPU, but in a device driver on a dedicated microcontroller it is not necessarily a poor choice. But if polling is a problem, there is nothing preventing a non-blocking, interrupt-based implementation with a stream-based client API.

The function `Last_Index` is a convenience function called by procedure Read. It is used to compute `Read.Last`, the index of the last array component assigned in `Read.Buffer`. The function result is `Buffer'First - 1` when no components are assigned, except when that would be less than the lowest possible array index value.

Here is a demonstration procedure to be run on the STM32 F4 Discovery Board:

```
with Last_Chance_Handler;  pragma Unreferenced (Last_Chance_Handler);
with Serial_IO;
with Peripherals_Streaming; use Peripherals_Streaming;

procedure Demo_Serial_Port_Streaming is
begin
   Serial_IO.Initialize_Hardware (Peripheral);
   Serial_IO.Configure (COM.Device, Baud_Rate => 115_200);
   --  This baud rate selection is entirely arbitrary. Note that you may
```

(continues on next page)

(continued from previous page)

```
   --  have to alter the settings of your host serial port to match this
   --  baud rate, or just change the above to match whatever the host
   --  serial port has set already. An application such as TerraTerm
   --  or RealTerm is helpful.

   loop
      declare
         --  await the next msg from the serial port
         Incoming : constant String := String'Input (COM'Access);
      begin
         --  echo the received msg content
         String'Output (COM'Access, "You sent '" & Incoming & "'");
      end;
   end loop;
end Demo_Serial_Port_Streaming;
```

The specific USART on the STM32 F4 Discovery Board must be connected to a serial port on the host computer. With that connection in place the embedded ARM board and the host computer can communicate over the two serial ports. This demonstration iteratively receives a string sent from the host, prepends some text, and sends that back, in effect echoing the host sender's text.

The stream attributes `String'Output` and `String'Input` write and read the bounds as well as the characters. As a consequence, you will need to use a program on the host that handles those bounds. A good way to do that is to use a host program that also uses streams to send and receive `String` values. Note that the ADL serial port examples include a host application that you can build and run for this purpose.

14.3 Pros

The stream-based approach has all the advantages of the generic-based approach without requiring generic instantiations. There is no limit to the number and kinds of client types supported, including limited types if they have the attributes defined. It is type-safe by default, because the compiler will verify that the type used to invoke a stream attribute is the same type as the value involved. In addition, the low-level implementation details are hidden inside the package body.

Furthermore, the approach is independent of other design considerations, such as whether callers wait for completion of the invoked I/O operation.

Because it is maximally flexible and concise, we consider it the best implementation for this idiom. The generic-based approach remains a good one, however.

14.4 Cons

Limited types do not support the stream I/O attributes by default, but developers can define them. Note that this is not a problem for the generic-based approach, because we declared the generic formal type as `limited` and wouldn't need to do anything within the generic that would contradict that. The client's generic actual type can then be either a limited type or not.

When multiple types are being sent and received, the sender and receiver must be coordinated so that the next value consumed from the stream is of the type expected by the receiver. For example, the next value in the stream might have been written by the sender as a floating-point value, via `Float'Write (...)`. The receiver must use `Float'Read(...)` to consume that value from the stream. Arguably, this is not really a *con* because it's true for any stream when multiple types are involved. Even if we used the generic-based approach, developers could instantiate the generic multiple times with different types and send their values via the same port. With streams this approach is as type-safe as it can be. However, see *Note #2* (page 120) below for a possible mitigation.

14.5 Relationship With Other Idioms

As stated above, we use *Interface Inheritance* (page 61) to visibly extend the root stream type.

14.6 Notes

1. You can find the streams-based approach, and others, in the *serial_ports* example in the ADL, for an STM32 F4 Discovery Board, found in Ada_Drivers_Library/examples/STM32F4_DISCO/*[56]. You can build and run them using the GNAT project file named serial_ports_f4disco.gpr[57] located in that directory. See the Ada_Drivers_Library/examples/shared/serial_ports/README.md[58] file for how to run them, including the special cable required for connecting the target board to the host computer. Note that a non-blocking, interrupt-drive approach is included there, although it is not stream-based.
2. In the *Cons section* (page 119) above, we mentioned the coordination issue that arises when values of multiple types are inserted and retrieved from a given stream. A possible alternative would be to send and receive only tagged types in a given class hierarchy. The receiver could then use the language-defined `Generic_Dispatching_Constructor` to dynamically dispatch to constructors for the values received from the stream. Thus, the receiver would not need to know in advance what specific types of values are incoming.

[56] https://github.com/AdaCore/Ada_Drivers_Library/tree/master/examples/STM32F4_DISCO
[57] https://github.com/AdaCore/Ada_Drivers_Library/blob/master/examples/STM32F4_DISCO/serial_ports_f4disco.gpr
[58] https://github.com/AdaCore/Ada_Drivers_Library/tree/master/examples/shared

CHAPTER

FIFTEEN

DEALING WITH SILENT TASK TERMINATION

15.1 Motivation

A task completes abnormally when an exception is raised in its sequence of statements and is not handled. Even if the task body has a matching exception handler and it executes, the task still completes after the handler executes, although this time it completes normally. Similarly, if a task is aborted the task completes, again abnormally.

Whatever the cause, once completed a task will (eventually) terminate, and it does this silently — there is no notification or logging of the termination to the external environment. A vendor could support notification via their run-time library[59], but the language standard does not require it and most vendors — if not all — do not.

Nevertheless, applications may require some sort of notification of the event that caused the termination. Assuming the developer is responsible for implementing it, how can the requirement best be met?

15.2 Implementation

For unhandled exceptions, the simplest approach to silent termination is to define the announcement or logging response as an exception handler located in the task body exception handler part:

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
-- ...
   task body Worker is
   begin
      --  ...
   exception
      when Error : others => -- last wishes
         Put_Line ("Task Worker terminated due to " & Exception_Name (Error));
   end Worker;
```

A handler at this level expresses the task's *last wishes* prior to completion, in this case printing the names of the task and the active exception to `Standard_Output`. (We could print the associated exception message too, if desired.) The **others** choice covers all exceptions not previously covered, so in the above it covers all exceptions. Specific exceptions also could be covered, but the **others** choice should be included (at the end) to ensure no exception occurrence can be missed.

You'll probably want this sort of handler for every application task if you want it for any of them. That's somewhat inconvenient if there are many tasks in the application, but feasible. Possible mitigation includes the use of a task type. In that case you need only define the handler once, in the task type's body. You could even declare such a task type

[59] The Verdix Ada Development System did so.

inside a generic package, with generic formal subprograms for the normal processing and the exception handler's processing. That would make the task type reusable. But that's a bit heavy, and it can be awkward. See the *Notes* (page 133) section below for details.

Alternatively, we could prevent unhandled exceptions from causing termination in the first place. We can do that by preventing task completion, via some additional constructs that prevent reaching the end of the task's sequence of statements. We will show these constructs incrementally for the sake of clarity.

Before we do, note that many tasks are intended to run until power is removed, so they have an infinite loop at the outer level as illustrated in the code below. For the sake of clarity and realism, we name the loop `Normal` and call some procedures to show the typical structure, along with the last wishes handler:

```ada
task body Worker is
begin
   Initialize_State;
   Normal : loop
      Do_Actual_Work;
   end loop Normal;
exception
   when Error : others => -- last wishes
      Put_Line ("Task Worker terminated due to " &
                Exception_Name (Error));
end Worker;
```

In the above, the procedures' names indicate what is done at that point in the code. The steps performed may or may not be done by actual procedure calls.

Strictly speaking, the optional exception handler part of the task body is the very end of the task's sequence of statements (the *handled sequence of statements*). We want to prevent the thread of control reaching that end — which would happen if any handler there ever executed — because the task would then complete.

Therefore, we first wrap the existing code inside a block statement. The task body's exception handler section becomes part of the block statement rather than at the top level of the task:

```ada
task body Worker is
begin
   begin
      Initialize_State;
      Normal : loop
         Do_Actual_Work;
      end loop Normal;
   exception
      when Error : others =>  -- last wishes
         Put_Line ("Task Worker terminated due to " &
                   Exception_Name (Error));
   end;
end Worker;
```

Now any exception raised within `Initialize_State` and `Do_Actual_Work` will be caught in the block statement's handler, not the final part of the task's sequence of statements. Nothing else changes, semantically. The task will still complete because the block statement exits after the handler executes, and so far there's nothing after that block statement. We need to make one more addition.

The second (and final) addition prevents reaching the end of the sequence of statements after a handler executes, and hence the task from completing. This is accomplished by wrapping the new block statement inside a new loop statement. We name this outermost loop Recovery:

```
task body Worker is
begin
   Recovery : loop
      begin
         Initialize_State;
         Normal : loop
            Do_Actual_Work;
         end loop Normal;
      exception
         when Error : others =>
            Put_Line (Exception_Name (Error) & " handled in task Worker");
      end;
   end loop Recovery;
end Worker;
```

When the block exits after the exception handler prints the announcement, normal execution resumes and the end of the `Recovery` loop is reached. The thread of control then continues at the top of the loop. Of course, absent an unhandled exception reaching this level, the `Normal` loop is never exited in the first place.

These two additions ensure that — with one caveat — `Worker` never terminates due to an unhandled exception raised during execution of the task's sequence of statements.

Note that an exception raised during elaboration of the task body's declarative part is not handled by the approach, or any other approach at this level, because the exception is propagated immediately to the master of the task. Such a task never reaches the handled sequence of statements in the first place.

The caveat concerns the language-defined exception `Storage_Error`. This exception requires special consideration because the reasons for raising it include exhausting the storage required for execution itself.

There are a couple of scenarios to consider.

The first scenario is task activation, i.e., creation. Initial task activation involves execution (in the tasking part of the run-time library) before the sequence of steps is reached. Hence task activation for the `Worker` task could fail due to an insufficient initial storage allocation. But because that failure happens before the block statement is entered it doesn't really apply to the caveat above.

The second scenario involves execution within the task's actual sequence of statements. Therefore it does apply to the caveat above. Here's why.

When called, the execution of a given subprogram requires a representation in storage, often known as a *frame*. Because subprogram calls and their returns can be seen as a series of stack pushes and pops, the representation for execution is typically via a stack of these frames. Calls cause stack frame pushes, creating new frames on the stack, and returns cause stack pops, reclaiming the frames. On exit, execution returns to the caller, so the previous top of the stack is now the active frame. Representation as a stack of frames works well so it is very common. (Functions returning values of unconstrained types are problematic because the size of the result isn't known at the point of the call, so the required frame size isn't known when the push occurs. Solutions vary, but that's a topic for another day.)

Now, suppose the task's sequence of statements includes a long series of subprogram calls, in which one subprogram calls another, and that one calls another, and so on, and none of these calls has yet returned. Eventually, of course, the dynamic call chain will end because the calls will return, at least in normal code. But let's suppose that the call chain is long and the most recent call has not yet returned to the caller.

In that case it is possible for one more call to exhaust the storage available for that task's execution. You can easily construct such a chain by calling an infinitely recursive procedure or function:

```
procedure P;

procedure P is
begin
   P;
end P;
```

When executing on a host OS it might take a very long time for a call to P to exhaust available storage, maybe longer than you'd be willing to wait. But on an embedded system, where physical storage is limited and there's no virtual memory, it might not take long at all.

Now, you might think that you don't use recursion, much less infinitely recursive routines, so this problem doesn't apply to you. But recursion is just an easy illustration. How long a call chain is too long? It depends on the memory resources available.

Moreover, exhaustion is not due only to the storage required for the call/return semantics. Frames include the representation of the local objects declared within the subprograms' declarative parts, if any.

```
procedure P;

procedure P is
   Local : Integer;
begin
   Local := 0;
   P;
end P;
```

Each execution of a call to P creates a semantically distinct instance of `Local`. A new frame containing the storage for each call's copy of `Local` implements that requirement nicely.

Of course, different subprograms usually declare different local objects, if they declare any at all. Because the storage required for these declarations varies, the corresponding frame sizes vary.

We can use that fact to reduce the length of the dynamic call chain required to illustrate storage exhaustion. The called subprograms will declare very large objects within their declarative parts. Hence each frame is correspondingly larger than if the subprogram declared nothing locally. Continuing the infinitely recursive subprogram example:

```
procedure P;

procedure P is
   type Huge_Component is array (Long_Long_Integer) of Long_Float;
   type Huge_Array is array (Long_Long_Integer) of Huge_Component;
   Local : Huge_Array;
begin
   Local := (others => (others => 0.0));
   P;
end P;
```

The size of the frame for an individual call to P will be very large indeed, if it is representable at all. Fewer calls will be required before `Storage_Error` is raised.

Now, with all that said, let's get back to this approach to silent termination. Here's the code again:

```
task body Worker is
begin
   Recovery : loop
      begin
         Initialize_State;
```

(continues on next page)

(continued from previous page)

```
         Normal : loop
            Do_Actual_Work;
         end loop Normal;
      exception
         when Error : others =>
            Put_Line (Exception_Name (Error) & " handled in task Worker.");
      end;
   end loop Recovery;
end Worker;
```

At this point you might be thinking that `Storage_Error` would be caught by the **others** choice anyway, so this (long-winded) talk about stack frames and dynamic call chains is irrelevant. That's where the caveat comes into play.

Specifically, if there's insufficient storage remaining for execution to continue, how how do we know there's enough storage remaining to execute the exception handler? For that matter, how do we even know there's enough storage available for the run-time library to find the handler in the first place? Absent a storage analysis, we can't know with certainty.

Therefore, if the application matters, perform a worst-case storage analysis per task, including the exception handlers, and explicitly specify the tasks' stacks accordingly. For example:

```
task Worker with Storage_Size => System_Config.Worker_Storage;
```

We've defined the value as a constant named `Worker_Storage` declared in an application-defined package `System_Config`. All such values are declared in that package, for the sake of centralizing all the application's configuration parameters. We'd declare all the tasks' priorities there too.

Finally, although this approach works, the state initialization requires some thought.

As shown above, full initialization is performed again when the `Recovery` loop circles back around to the top of the loop. As a result, the *normal* processing in `Do_Actual_Work` must be prepared for suddenly encountering completely different state, i.e., a restart to the initial state. If that is not feasible the call to `Initialize_State` could be moved outside, prior to the start of the `Recovery` loop, so that it only executes once. Perhaps a different initialization procedure could be called after the exception handler to do partial initialization. Whether or not that will suffice depends on the application.

However, these approaches do not address task termination due to task abort statements.

Aborting tasks is both messy and expensive at run-time. If a task is updating some object and is aborted before it finishes the update, that object is potentially corrupted. That's the messiness. If an aborted task has dependent tasks, all the dependents are aborted too, transitively. A task in a rendezvous with the aborted task is affected, as are those queued waiting to rendezvous with it, and so on. That's part of the expensiveness when aborts are actually used. Worse, even if never used, abort statements impose an expense at run-time. The language semantics requires checks for an aborted task at certain places within the run-time library. Those checks are executed even if no task abort statement is ever used in the application. To avoid that distributed cost, you would need to apply a tasking profile disallowing abort statements and build the executable with a correspondingly reduced run-time library implementation.

As a consequence, aborting a task should be very rarely done. Regardless, the task abort statement exists. How can we express a *last wishes* response for that cause too?

Fortunately, Ada provides a facility that addresses all possible causes: normal termination, termination due to task abort, and termination due to unhandled exceptions.

With this facility developers specify procedures that are invoked automatically by the run-time library during task finalization. These procedures express the last wishes for the task, but do not require any source code within the task, unlike the exception handler in each task body described earlier. These response procedures are known as *handlers.*

During execution, handlers can be applied to an individual task or to groups of related tasks. Handlers can also be removed from those tasks or replaced with other handlers. Because procedures are not first-class entities in Ada, handlers are assigned and removed by passing access values designating them.

The facility is defined by package `Ada.Task_Termination`. The package declaration for this language-defined facility follows, with slight changes for easier comprehension.

```
with Ada.Task_Identification;  use Ada.Task_Identification;
with Ada.Exceptions;           use Ada.Exceptions;

package Ada.Task_Termination
   --  ...
is
   type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);

   type Termination_Handler is access protected procedure
     (Cause : in Cause_Of_Termination;
      T     : in Task_Id;
      X     : in Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler
     (Handler : in Termination_Handler);

   function Current_Task_Fallback_Handler
     return Termination_Handler;

   procedure Set_Specific_Handler
     (T       : in Task_Id;
      Handler : in Termination_Handler);

   function Specific_Handler (T : Task_Id) return Termination_Handler;
end Ada.Task_Termination;
```

As shown, termination handlers are actually protected procedures, with a specific parameter profile. Therefore, the type `Termination_Handler` is an access-to-protected-procedure with that signature. The compiler ensures that any designated protected procedure matches the parameter profile.

Termination handlers apply either to a specific task or to a group of related tasks, including potentially all tasks in the partition. Each task has one, both, or neither kind of handler. By default none apply. (Unless a partition is part of a distributed program, a single partition constitutes an entire Ada program.)

Clients call procedure `Set_Specific_Handler` to apply the protected procedure designated by `Handler` to the task with the specific `Task_Id` value T. These are known as *specific* handlers. The use of a `Task_Id` to specify the task, rather than the task name, means that we can set or remove a handler without having direct visibility to the task in question.

Clients call procedure `Set_Dependents_Fallback_Handler` to apply the protected procedure designated by `Handler` to the task making the call, i.e., the current task, and to all tasks that are dependents of that task. These handlers are known as *fall-back* handlers.

Handlers are invoked automatically, with the following semantics:

1. If a specific handler is set for the terminating task, it is called and then the response finishes.
2. If no specific handler is set for the terminating task, the run-time library searches for a fall-back handler. The search is recursive, up the hierarchy of task masters, including, ultimately, the environment task. If no fall-back handler is found no handler calls are

made whatsoever. If a fall-back handler is found it is called and then the response finishes; no further searching or handler calls occur.

As a result, at most one handler is called in response to any given task termination.

The following client package illustrates the approach. Package `Obituary` declares protected object `Obituary.Writer`, which declares two protected procedures. Both match the profile specified by type `Termination_Handler`. One such procedure would suffice, we just provide two for the sake of illustrating the flexibility of the dynamic approach.

Listing 5: obituary.ads

```ada
with Ada.Exceptions;          use Ada.Exceptions;
with Ada.Task_Termination;    use Ada.Task_Termination;
with Ada.Task_Identification; use Ada.Task_Identification;

package Obituary is

   protected Writer is

      procedure Note_Passing
         (Cause    : Cause_Of_Termination;
          Departed : Task_Id;
          Event    : Exception_Occurrence);
      --  Written by someone who's read too much English lit

      procedure Dissemble
         (Cause    : Cause_Of_Termination;
          Departed : Task_Id;
          Event    : Exception_Occurrence);
      --  Written by someone who may know more than they're saying

   end Writer;

end Obituary;
```

Clients can choose among these protected procedures to set a handler for one or more tasks.

The two protected procedures display messages corresponding to the cause of the termination. One procedure prints respectful messages, in the style of someone who's read too much Old English literature. The other prints rather dissembling messages, as if written by someone who knows more than they are willing to say. The point of the difference is that more than one handler can be available to clients, and their choice is made dynamically at run-time.

The package body is structured as follows:

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

package body Obituary is

   protected body Writer is
      procedure Note_Passing () is ...
      procedure Dissemble () is ...
   end Writer;

begin -- optional package executable part
   Set_Dependents_Fallback_Handler (Writer.Note_Passing'Access);
end Obituary;
```

In addition to defining the bodies of the protected procedures, the package body has an executable part. That part is optional, but in this case it is convenient. This executable part calls procedure `Set_Dependents_Fallback_Handler` to apply one of the two handlers.

Because this call happens during library unit elaboration, it sets the fall-back handler for all the tasks in the partition (the program). The effect is global to the partition because library unit elaboration is invoked by the *environment task,* and the environment task is the ultimate master of all application tasks in a partition. Therefore, the fall-back handler is applied to the top of the task dependents hierarchy, and thus to all tasks. The application tasks need not do anything in their source code for the handler to apply to them.

The call to `Set_Dependents_Fallback_Handler` need not occur in this particular package body, or even in a package body at all. But because we want it to apply to all tasks in this specific example, including library tasks, placement in a library package's elaboration achieves that effect.

The observant reader will note the with-clause for `Ada.Text_IO`, included for the sake of references to `Put_Line`. We'll address the ramifications momentarily. Here are the bodies for the two handlers:

Listing 6: obituary.adb

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

package body Obituary is

   protected body Writer is

      procedure Note_Passing
        (Cause    : Cause_Of_Termination;
         Departed : Task_Id;
         Event    : Exception_Occurrence)
      is
      begin
         case Cause is
            when Normal =>
               Put_Line (Image (Departed) &
                         " went gently into that good night");
            when Abnormal =>
               Put_Line (Image (Departed) & " was most fouly murdered!");
            when Unhandled_Exception =>
               Put_Line (Image (Departed) &
                         " was unknit by the much unexpected " &
                         Exception_Name (Event));
         end case;
      end Note_Passing;

      procedure Dissemble
        (Cause    : Cause_Of_Termination;
         Departed : Task_Id;
         Event    : Exception_Occurrence)
      is
      begin
         case Cause is
            when Normal =>
               Put_Line (Image (Departed) & " died, naturally.");
               Put_Line ("We had nothing to do with it.");
            when Abnormal =>
               Put_Line (Image (Departed) & " had a tragic accident.");
               Put_Line ("We're sorry it had to come to that.");
            when Unhandled_Exception =>
               Put_Line (Image (Departed) &
                         " was apparently fatally allergic to " &
                         Exception_Name (Event));
         end case;
      end Dissemble;
```

(continues on next page)

(continued from previous page)

```ada

   end Writer;

begin -- optional package executable part
   Set_Dependents_Fallback_Handler (Writer.Note_Passing'Access);
end Obituary;
```

Now, about those calls to `Ada.Text_IO.Put_Line`. Procedure `Put_Line` is a *potentially blocking* operation. Consequently, a call within a protected operation is a bounded error (see RM 9.5.1(8)) and the resulting execution is not portable. For example, the `Put_Line` calls will likely work as expected on a native OS. However, their execution may do something else on other targets, including raising `Program_Error` if detected. The GNAT bare-metal targets, for example, raise `Program_Error`.

For a portable approach, we move these two blocking calls to a new dedicated task and revise the protected object accordingly. That's portable because a task can make blocking calls.

First, we change `Obituary.Writer` to have a single protected procedure and a new entry. The protected procedure will be used as a termination handler, as before, but does not print the messages. Instead, when invoked by task finalization, the handler enters the parameter values into an internal data structure and then enables the entry barrier on the protected entry. The dedicated task waits on that entry barrier and, when enabled, retrieves the stored values describing a termination. The task can then call `Put_Line` to print the announcement with those values.

Here's the updated `Obituary` package declaration:

Listing 7: obituary.ads

```ada
with Ada.Exceptions;            use Ada.Exceptions;
with Ada.Task_Termination;      use Ada.Task_Termination;
with Ada.Task_Identification;   use Ada.Task_Identification;
with Ada.Containers.Vectors;

package Obituary is

   pragma Elaborate_Body;

   Comment_On_Normal_Passing : Boolean := True;
   --  Do we say anything if the task completed normally?

   type Termination_Event is record
      Cause    : Cause_Of_Termination;
      Departed : Task_Id;
      Event    : Exception_Id;
   end record;

   package Termination_Events is new Ada.Containers.Vectors
     (Positive, Termination_Event);

   protected Writer is

      procedure Note_Passing
         (Cause    : Cause_Of_Termination;
          Departed : Task_Id;
          Event    : Exception_Occurrence);

      entry Get_Event (Next : out Termination_Event);

   private
```

(continues on next page)

(continued from previous page)

```
      Stored_Events : Termination_Events.Vector;
   end Writer;

end Obituary;
```

As a minor refinement we add the option to not print announcements for normal completions, for those applications that allow task completion.

We must declare the generic container instantiation outside the protected object, an unfortunate limitation of protected objects. We would prefer that clients have no compile-time visibility to it, since it is an implementation artifact.

The updated package body is straightforward:

Listing 8: obituary.adb

```
package body Obituary is

   protected body Writer is

      ------------------
      -- Note_Passing --
      ------------------

      procedure Note_Passing
         (Cause    : Cause_Of_Termination;
          Departed : Task_Id;
          Event    : Exception_Occurrence)
      is
      begin
         if Cause = Normal and then
            not Comment_On_Normal_Passing
         then
            return;
         else
            Stored_Events.Append
              (Termination_Event'(Cause,
                                  Departed,
                                  Exception_Identity (Event)));
         end if;
      end Note_Passing;

      ---------------
      -- Get_Event --
      ---------------

      entry Get_Event (Next : out Termination_Event)
      when
         not Stored_Events.Is_Empty
      is
      begin
         Next := Stored_Events.First_Element;
         Stored_Events.Delete_First;
      end Get_Event;

   end Writer;

begin -- optional package executable part
   Set_Dependents_Fallback_Handler (Writer.Note_Passing'Access);
end Obituary;
```

In the body of Note_Passing, we store the Exception_Id for the exception occurrence indicated by Event. That exception occurrence need not be active by the time the task reads the Id for that occurrence.

A new child package declares the task that prints the termination information:

Listing 9: obituary-output.ads

```ada
package Obituary.Output is
   pragma Elaborate_Body;

   task Printer;

end Obituary.Output;
```

In the package body, the task body iteratively suspends on the call to `Writer.Get_Event`, waiting for a termination handler to make the termination data available. Once it returns from the call, if ever, it simply prints the information and awaits further events:

Listing 10: obituary-output.adb

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

package body Obituary.Output is

   -------------
   -- Printer --
   -------------

   task body Printer is
      Next : Termination_Event;
   begin
      loop
         Writer. Get_Event (Next);
         case Next.Cause is
            when Normal =>
                Put_Line (Image (Next.Departed) & " died, naturally.");
                --  What a difference that comma makes!
                Put_Line ("We had nothing to do with it.");
            when Abnormal =>
               Put_Line (Image (Next.Departed) &
                         " had a terrible accident.");
               Put_Line ("We're sorry it had to come to that.");
            when Unhandled_Exception =>
               Put_Line (Image (Next.Departed) &
                         " reacted badly to " &
                         Exception_Name (Next.Event));
               Put_Line ("Really, really badly.");

         end case;
      end loop;
   end Printer;

end Obituary.Output;
```

We declared this task in a child package because one can view the `Printer` and the `Writer` as parts of a single subsystem, but that structure isn't necessary. An unrelated application task could just as easily retrieve the information stored by the protected `Writer` object.

Here is a sample demonstration main procedure, a simple test to ensure that termination due to task abort is captured and displayed:

Listing 11: demo_fallback_handler_abort.adb

```ada
with Obituary.Output; pragma Unreferenced (Obituary.Output);
--  otherwise neither package is in the executable

procedure Demo_Fallback_Handler_Abort is

   task Worker;
   task body Worker is
   begin
      loop  --  ensure not already terminated when aborted
         delay 0.0;  --  yield the processor
      end loop;
   end Worker;

begin
   abort Worker;
end Demo_Fallback_Handler_Abort;
```

Note that the nested task would not be accepted under the Ravenscar or Jorvik profiles because those profiles require tasks to be declared at the library level, but that can easily be addressed.

When this demo main is run, the output looks like this:

```
worker_00000174BC68A570 had a terrible accident.
We're sorry it had to come to that.
```

The actual string representing the task identifier will vary with the implementation.

You'll have to use control-c (or whatever is required on your host) to end the program because the `Printer` task in `Obituary.Output` runs forever. Many applications run *forever* so that isn't necessarily a problem. That could be addressed if need be.

15.3 Pros

The facility provided by package `Ada.Task_Termination` allows developers to respond in any way required to task termination. The three causes, normal completion, unhandled exceptions, and task abort are all supported. Significantly, no source code in application tasks is required for the termination support to be applied, other than the isolated calls to set the handlers.

15.4 Cons

On a bare metal target there may be restrictions that limit the usefulness of the facility. For example, on targets that apply the Ravenscar or Jorvik profiles, task abort is not included in the profile and tasks are never supposed to terminate for any reason, including normally. Independent of the profiles, some run-time libraries may not support exception propagation, or even any exception semantics at all.

15.5 Relationship With Other Idioms

None.

15.6 Notes

If you did want to use a generic package to define a task type that is resilient to unhandled exceptions, you could do it like this:

Listing 12: resilient_workers.ads

```ada
with System;
with Ada.Exceptions; use Ada.Exceptions;

generic
   type Task_Local_State is limited private;
   with procedure Initialize (This : out Task_Local_State);
   with procedure Update (This : in out Task_Local_State);
   with procedure Respond_To_Exception
          (Current_State : in out Task_Local_State;
           Error         : Exception_Occurrence);
package Resilient_Workers is

   task type Worker
     (Urgency : System.Priority := System.Default_Priority)
   with
      Priority => Urgency;

end Resilient_Workers;
```

Listing 13: resilient_workers.adb

```ada
package body Resilient_Workers is

   task body Worker is
      State : Task_Local_State;
   begin
      Recovery : loop
         begin
            Initialize (State);
            Normal : loop
               Update (State);
               --  The call above is expected to return, ie
               --  this loop is meant to iterate
            end loop Normal;
         exception
            when Error : others =>
               Respond_To_Exception (State, Error);
         end;
      end loop Recovery;
   end Worker;

end Resilient_Workers;
```

Although this code looks useful, in practice it has issues.

First, in procedure `Initialize`, the formal parameter mode may be a problem. You might need to change the parameter mode from mode out to mode in-out instead, because recovery from unhandled exceptions will result in another call to `Initialize`. Mode out makes sense for the first time `Initialize` is called, but does it make sense for all calls after that? It depends on the application's procedures. The behavior of `Update` may be such that local state should only partially be reset in subsequent calls to `Initialize`.

Furthermore, if `Initialize` must only perform a partial initialization on subsequent calls, the procedure must keep track of the number of calls. That requires a variable declared external to the body of `Initialize`. The additional complexity is unfortunate. We could perhaps mitigate this problem by having two initialization routines passed to the instantiation: one for full initialization, called only once with mode out for the state, and one for partial initialization, called on each iteration of the `Recovery` loop with mode in-out for the state:

Listing 14: resilient_workers.adb

```ada
package body Resilient_Workers is

   task body Worker is
      State : Task_Local_State;
   begin
      Fully_Initialize (State);

      Recovery : loop
         begin
            Normal : loop
               Update (State);
               --  The call above is expected to return, i.e.
               --  this loop is meant to iterate
            end loop Normal;
         exception
            when Error : others =>
               Respond_To_Exception (State, Error);
         end;

         Partially_Initialize (State);
      end loop Recovery;
   end Worker;

end Resilient_Workers;
```

If both application initialization routines happen to do the same thing, we'd like the developer to be able to pass the same application procedure to both generic formal procedures `Fully_Initialize` and `Partially_Initialize` in the instantiation. But that wouldn't compile because the parameter modes don't match.

Then there's the question of the nature of the task. Is it periodic, or sporadic, or free running? If it is periodic, we need a delay statement in the `Normal` loop to suspend the task for the required period. The generic's task body doesn't do that. The actual procedure passed to `Update` could do the delay, but now, like a single version of `Initialize` required to do both partial and full initialization, it needs additional state declared external to the procedure body (for the `Time` variable used by the absolute delay statement).

Finally, the single generic formal type used to represent the task's local state can be awkward. Having one type for a task's total state is unusual, and aggregating otherwise unrelated types into one isn't good software engineering and doesn't reflect the application domain. Nor is it necessarily trivial to create one type representing a set of distinct variables. For example, some of these stand-alone variables could be objects of indefinite types. Different task objects of a given task type might not agree on those objects' constraints. Furthermore, that awkwardness extends to the procedures that use that single object, in that every procedure except for `Initialize` will likely ignore parts of it.

In summary, the problems are likely more problematic than this generic is worth.

CHAPTER

SIXTEEN

IDIOMS FOR PROTECTED OBJECTS

Note

Parts of this chapter were originally published as the Blog Post: On the Benefits of Families ... (Entry Families)[60].

First, a bit of background regarding synchronization in concurrent programming, protected objects, and the requeue statement. If you already know this material, feel free to skip ahead to the *Motivation* (page 137) section for this chapter.

Concurrent programming is more complex (and more fun) than purely sequential programming. The cause of this complexity is two-fold: 1) the executing threads' statements are likely interleaved at the assembly language level, and 2) the order of that interleaving is unpredictable. As a result, developers cannot know, in general, where in its sequence of statements any given thread is executing. Developers can only assume that the threads are making finite progress when they execute.

A consequence of unpredictable interleaving is that the bugs specific to this type of programming are timing-dependent. Such bugs are said to be *Heisenbugs* because they "go away when you look at them," i.e., changing the code — adding debugging statements or inserting debugger breakpoints — changes the timing. The bug might then disappear entirely, or simply appear elsewhere in time. We developers must reason about the possible effects of interleaving and design our code to prevent the resulting bugs. (That's why this is fun.) Such bugs are really design errors.

One of these errors is known as a *race condition*. A race condition is possible when multiple threads access some shared resource that requires mutually exclusive access. If we accidentally forget the finite progress assumption we may incorrectly assume that the threads sharing that resource will access it serially. Unpredictable execution interleaving cannot support that assumption.

Race conditions on memory locations are the most common, but the issue is general in nature, including for example hardware devices and OS files. Hence the term "resource."

For example, suppose multiple threads concurrently access an output display device. This device can be ordered to move its cursor to arbitrary points on the display by writing a specific sequence of bytes to it, including the numeric values for X and Y coordinates. A common use is to send the "move cursor to X, Y" sequence and then send the text intended to appear at coordinates X and Y.

Clearly, this device requires each client thread to have mutually exclusive access to the device while sending those two byte sequences. Otherwise, uncoordinated interleaving could result in one thread preempting another thread in the middle of sending those two sequences. The result would be an intermingled sequence of bytes sent to the device. (On a graphics display the chaotic output can be entertaining to observe.)

[60] https://www.adacore.com/blog/on-the-benefits-of-families

Memory races on variables are less obvious. Imagine two threads, Thread1 and Thread2, that both increment a variable visible to both (an integer, let's say).

Suppose that the shared integer variable has a value of zero. Both threads increment the variable, so after they do so the new value should be two. The compiler will use a hardware register to hold and manipulate the variable's value because of the increased performance over memory accesses. Each thread has an independent copy of the registers, and will perform the same assembly instructions:

1. load a register's value from the memory location containing the variable's value
2. increment the register's value
3. store the register's new value back to the variable's memory location.

The two threads' executions may be interleaved in these three steps. It is therefore possible that Thread1 will execute step 1 and step 2, and then be preempted by the execution of Thread2. Thread2 also executes those two steps. As a result, both threads' registers have the new value of 1. Finally, Thread1 and Thread2 perform the third step, both storing a value of 1 to the variable's memory location. The resulting value of the shared variable will be 1, rather than 2.

Another common design bug is assuming that some required program state has been achieved. For example, for a thread to retrieve some data from a shared buffer, the buffer must not be empty. Some other thread must already have inserted data. Likewise, for a thread to insert some data, the buffer must not be full. Again, the finite progress assumption means that we cannot know whether either of those two states are achieved.

Therefore, interacting threads require two forms of synchronization: *mutual exclusion* and *condition synchronization*. These two kinds of synchronization enable developers to reason rigorously about the execution of their code in the context of the finite progress assumption.

Mutual exclusion synchronization prevents threads that access some shared resource from doing so at the same time, i.e., it provides mutually exclusive access to that resource. The effect is achieved by ensuring that, while any given thread is accessing the resource, that execution will not be interleaved with the execution of any other thread's access to that shared resource.

Condition synchronization suspends a thread until the condition — an arbitrary Boolean expression — is `True`. Only when the expression is (or becomes) `True` can the caller thread be allowed to continue.

A thread-safe bounded buffer is a good example for these two kinds of synchronization. Some threads, the producers, will insert items into the buffer. Other threads, the consumers, will concurrently remove items. The array object representing the buffer contents, as well as the indexes into the array, require mutually exclusive access for both producers and consumers. Furthermore, producers must be blocked (suspended) as long as the given buffer is full, and consumers must be blocked as long as the given buffer is empty.

Concurrent programming languages support mechanisms providing the two forms of synchronization. In some languages these are explicit constructs; other languages take different approaches. In any case, developers can apply these mechanisms to enforce assumptions more specific than simple finite progress.

Ada uses the term `task` rather than thread so we will use that term from here on.

The protected procedures and protected entries declared by a protected object (PO) automatically execute with mutually exclusive access to the entire protected object. No other caller task can be executing these operations at the same time, so execution of the procedure or entry body statements will not be interleaved. (Functions are special because they have read-only access to the data in the PO.) Therefore, there can be no race conditions on the data encapsulated within it. Even if the protected object has no encapsulated data, these operations always execute with mutually exclusive access. During such execution we can say that the PO is locked, speaking informally, because all other caller tasks are held pending.

Protected entries are much like protected procedures, except that entry bodies include a barrier condition that is used to express condition synchronization. The condition is an arbitrary Boolean expression, although there are some restrictions on the content for implementation reasons. Only when the barrier condition is **True** will a caller to the entry be allowed to execute the entry body. Once the body completes, the caller exits and can continue execution outside of the PO. (We'll say more about that later.) For example, entry barriers can express whether a bounded buffer is full or empty, thereby enabling and disabling buffer insertion and removal.

Under some circumstances, an entry may execute a requeue statement to reroute the caller to some other entry, for reasons that will be explained shortly, but from the caller task's point of view there is only one call being made.

The requeue statement may not be familiar to many readers. To explain its semantics we first need to provide its rationale.

Ada synchronization constructs are based on *avoidance synchronization*, meaning that:

1. the user-written controls that enable/disable the execution of protected entry bodies and task entry accept statements enable them only when they can actually provide the requested service, and
2. that determination is based on information known prior to the execution of the entry body or accept statement.

For example, at runtime, if a bounded buffer is full, that fact can be determined from the buffer's state: is the count of contained items equal to the capacity of the backing array? If so, the user-defined controls disable the operation to insert another value. Likewise, if the buffer is empty, the removal operation is disabled. When we write the buffer implementation we know beforehand what the operations will try to do, so we can write the controls to disallow them at runtime until they can succeed. Most of the time that's sufficient, but not always. If we can't know precisely what the operations will do when we write the code, avoidance synchronization won't suffice.

The **requeue** statement is employed when avoidance synchronization is not sufficient. A task calling an entry that executes a requeue statement is much like a person calling a large company on the telephone. Calling the main number connects you to a receptionist (if you're lucky and don't get an annoying menu). If the receptionist can answer your question, they do so and then you both hang up. Otherwise, the receptionist forwards the call to the person they determine that you need to speak with. After doing so, the receptionist hangs up, because from their point of view the call is complete. The call is not complete from your point of view, though, until you finish your conversation with the new person. And of course you may have to wait to speak to that person.

Like the receptionist, the first entry called must take (execute) the call without knowing what the request will be, because the entry barrier cannot reference the entry parameters. The parameter values are only known once the entry body executes. Therefore, the first entry may or may not be able to provide the requested service and allow the caller to return from the call. If not, it requeues the call and finishes, leaving the call still pending on the requeue target, i.e., the second entry.

A requeue statement is not required in all cases but, as you will see, sometimes it is essential. Note that protected procedures cannot execute requeue statements, only protected entries can do so. Protected procedures are appropriate when only mutual exclusion is required (to update encapsulated data).

16.1 Motivation

Of the several highly significant features added to the Ada language over the years, protected objects are one of the most important.

One of the reasons for this prominence is that protected objects make efficient asynchronous task interactions possible. Many, if not most, task interactions are asynchronous, but early Ada had only a synchronous mechanism for communication and synchronization, known informally as the rendezvous. The rendezvous is a high-level, very robust mechanism providing communication and synchronization for two tasks at a time. This mechanism isn't a problem in itself. If the application requires what amounts to an atomic action with two task participants, then the rendezvous meets this requirement nicely.

But, as a synchronous mechanism, the rendezvous is far too expensive when only an asynchronous mechanism (involving only one task) is required. Older mechanisms used for asynchronous interactions, such as semaphores, mutexes, and condition variables, are nowhere near as simple and robust for clients, but are much faster.

In addition, the rendezvous is only available between tasks, meaning that abstractions requiring mutual exclusion and condition synchronization had to be implemented as tasks too. Inserting and removing from a thread-safe buffer, for example, involved expensive task switching between the buffer task, the producer task, and the consumer task. This was the primary source of comparative inefficiency.

There was a non-standard notion of a *passive task* that wasn't actually a thread of control, and therefore did not require task switching, but it was not widely adopted. In that same vein, Ada 80 had a built-in `Semaphore` task type, intended to be implemented efficiently and used as the name suggests, but mixing the higher-level rendezvous with the much lower-level semaphore abstraction was considered poor language design. It did not survive to the ANSI and first ISO standards. Ultimately, the designers of the first version of Ada thought that processors would become so much faster in the future that the relative inefficiency and semantic mismatch wouldn't matter. Processors did get faster, but the problems still mattered.

Another reason that protected objects are so important is that they are applicable to a wide range of programming domains. Protected objects are critical to concurrent programming, real-time programming, and embedded systems programming with Ada. We've already highlighted their high level, robust support for asynchronous interactions in concurrent programming. For real-time programming, systems of any significant complexity will map deadlines to tasks. Consequently, in such systems the programming model is concurrent programming with the addition of predictability. In these systems protected objects have additional semantics (e.g., priorities) but supply the same benefits as in concurrent programming. For embedded systems programming, protected objects are used to express interrupt handlers, again with added semantics.

Their most important contribution, however, goes beyond direct client use of their automatic mutual exclusion and condition synchronization semantics. Developers can use protected objects to create just about any synchronization and communication protocol imaginable. These include application independent abstractions such as atomic actions, readers-writers locks, mutexes, and so on, but also schemes based on application-specific protocols and data structures. When combined with other language features, such as requeue, task identifiers, and the overall composition capabilities of the language, the result is a flexible, powerfully expressive facility.

16.2 Implementation

Protected objects are primarily utilized in two ways. We will refer to these two ways as idioms, for the sake of consistency with the rest of this course, although the other idioms in this course are much more narrow in scope. In the first, protected objects encapsulate and manipulate application-specific data. In the second, protected objects are used to create developer-defined synchronization and communication abstractions.

16.2.1 First Idiom Category: Application-Specific Protected Objects

In the first idiom, a given protected object implements all the application-specific functionality for the shared resources it encapsulates. We declare the shared data in the PO private part and declare the protected entries and procedures that manipulate that data in the visible part, as well as functions for reading that data if needed. The compiler won't allow any direct references to the hidden data from outside of the PO; the visible operations must be called by client tasks to manipulate the data. Readers familiar with the classic *monitor* construct will recognize it as the conceptual foundation for protected objects used this way.

For example, let's say we want to protect a product serial number variable from concurrent manipulation by multiple caller tasks. These tasks need to get the next sequential serial number, which entails incrementing the current value each time a task requests the next number. We must prevent the increment from occurring concurrently, otherwise the resulting race condition could occasionally provide incorrect values to the callers. Therefore, the increment will be done inside a protected procedure that provides the current value via parameter and also increments the value before returning. We declare the protected object like so:

```ada
protected Serial_Number is
   procedure Get_Next (Number : out Positive);
private
   Value : Positive := 1;
end Serial_Number;


protected body Serial_Number is

   procedure Get_Next (Number : out Positive) is
   begin
      Number := Value;
      Value  := Value + 1;
   end Get_Next;

end Serial_Number;
```

Whenever any task calls `Serial_Number.Get_Next`, the task will block until it has mutually exclusive access to the PO, and consequently to the `Serial_Number.Value` component. At that point, `Value` is assigned to the formal parameter and then incremented. Once the procedure returns, the caller task can continue execution with their unique serial number copy. No race conditions are possible and the shared serial number value increments safely each time `Get_Next` is called.

Note the robust nature of a protected object's procedural interface: clients simply call the protected procedures, entries, or functions. The called procedure or entry body, when it executes, will always do so with mutually exclusive access. (Functions can have some additional semantics that we can ignore here.) There is no explicit lower level synchronization mechanism for the client to acquire and release. The semantics of protected objects are implemented by the underlying Ada run-time library, hence all error cases are also covered. This procedural interface, with automatic implementation for mutual exclusion, is one of the significant benefits of the monitor construct on which protected objects are based.

16.2.2 Second Idiom Category: Developer-Defined Concurrency Abstractions

In the second idiom, data may be declared in the protected object private part, but they are not application data. Likewise, the protected operations do nothing application specific.

Instead, the PO provides some synchronization (and perhaps communication) protocol that we want to make available to client tasks. These tasks call the protected object's operations in order to get the protocol's semantics applied to their execution. The data declared in the private part, if any, exist purely for the sake of implementing the intended protocol.

In particular, the protected operations block and release the caller tasks per the new abstraction's semantics. We are using the term *block* loosely here, meaning the caller task is not allowed to return from the call until some condition holds.

These abstractions are frequently declared as protected types rather than anonymously typed protected objects like the `Serial_Number` PO. Protected types are especially preferred when the protocol is application independent and hence reusable. Declaration as a type also provides all the flexibility of types, including the ability to declare as many objects of the type as required and the ability to compose other types using them. Types also allow parameterization via discriminants, if necessary.

The synchronization abstractions may be classic mechanisms long known to the concurrent programming community, for example semaphores, or they may be wholly novel, perhaps based on application-specific contexts and data structures. Very sophisticated abstractions can be expressed, such as atomic actions involving an arbitrary number of tasks. The possibilities are endless.

For example, we could have a protected type that implements the Readers-Writers synchronization protocol. In this protocol only one task at a time can write (update) the state of the shared objects, and writers must wait until there are no readers, but multiple simultaneous readers are allowed as long as there is no writer active. Such a protected object would have multiple protected operations, some to block callers until appropriate for the given read or write action requested, and some to signal the end of the read or write operation so that a pending request (if any) can be granted.

This second idiomatic application of protected objects is extremely useful and therefore common. However, there is also a situation in which we are forced to use it, for the sake of portability. That happens when statements that would otherwise be within a protected operation include a potentially blocking operation. This is a term defined by the language for those constructs that may cause a currently executing caller task to yield the processor to another task. As such, they are not allowed within protected operations, neither directly nor indirectly. To understand why, you need to understand the underlying system approaches available for implementing the mutually exclusive access that protected operations provide automatically.

16.2.2.1 System Implementation of PO Mutual Exclusion

The underlying run-time library implements the mutual exclusion and thread management semantics for protected objects. Two approaches are known.

One implementation approach, typical when executing on an operating system, uses an explicit locking mechanism provided by the OS. The run-time library code implementing the protected operations first acquires a dedicated OS lock and then later releases it when exiting.

But another approach is available that does not use explicit locks. Instead, mutual exclusion is implemented via priorities, both task priorities and PO priorities. Note that this implementation requires priorities to be defined, execution on a uniprocessor, and the `Ceiling_Locking` policy to have been specified via the `Locking_Policy` pragma.

Specifically, developers assign a priority to each protected object. Each PO priority must be the highest (the *ceiling*) of all the priorities of those tasks that actually call the operations provided by the PO. Consequently, for any given PO, no task that calls that PO will have a higher priority than the PO priority. Because caller tasks inherit the PO priority (immediately), their calls execute with the highest priority of any caller task for that specific PO. Therefore, no other caller task can preempt any current caller executing within the PO. The current caller may be preempted, but not by a task that would also call that same PO. Thus, mutually exclusive access is provided automatically, and very efficiently. This approach has other benefits as well that are not pertinent here.

However, the priority-based implementation cannot work reliably if blocking is allowed within a protected operation. If the current caller could yield the processor inside a protected operation, some other task could then be allowed to continue execution, including possibly a task making a call to that same PO. In that case mutual exclusion would not be provided for that PO.

As a result, the language defines a number of potentially blocking operations and disallows them within protected operations. Any I/O operation is potentially blocking, for example, as are delay statements, but there are others as well. See the Ada RM, section 9.5{34} for the full list.

For example, in the *Dealing with Silent Task Termination idiom* (page 121) idiom we had an initial implementation of a protected procedure body that called `Ada.Text_IO.Put_Line`:

```
procedure Dissemble
  (Cause    : in Cause_Of_Termination;
   Departed : in Task_Id;
   Event    : in Exception_Occurrence)
is
begin
   case Cause is
      when Normal =>
         Put_Line (Image (Departed) & " died, naturally.");
         Put_Line ("We had nothing to do with it.");
      when Abnormal =>
         Put_Line (Image (Departed) & " had a tragic accident.");
         Put_Line ("We're sorry it had to come to that.");
      when Unhandled_Exception =>
         Put_Line (Image (Departed) &
                   " was apparently fatally allergic to " &
                   Exception_Name (Event));
   end case;
end Dissemble;
```

As described in that idiom entry, the above might work, but it is not portable.

As a consequence, we may find ourselves with some statement (e.g., the call to `Put_Line`) that would have been within a protected operation for the sake of mutually exclusive access, but that cannot be included there if the code is to be portable. The statement must be written outside of a PO, not within a protected object's operations, and not in anything called by those protected operations.

16.2.2.2 Examples for Second Idiom Category

We will use the approach to potentially blocking operations as the first example.

Suppose we are implementing a message logging facility. Any given task executing in the application can write a log message by calling a procedure named `Enter`, defined in package Log. The actual messages are values of type **`String`**:

```
package Log is
   procedure Enter (Log_Entry : String);
end Log;
```

Messages are written to an external file so that they will persist. That file is declared in the package body. Therefore, the package design is an *Abstract Data Machine* (page 17):

```
with Ada.Text_IO;  use Ada.Text_IO;
package body Log is

   Log_File : File_Type;

```

(continues on next page)

(continued from previous page)

```
   procedure Enter (Log_Entry : String) is
   begin
      Put_Line (Log_File, Msg);
      Flush (Log_File);
   end Enter;

begin
   Create (Log_File, Out_File, "log.txt");
end Log;
```

Unfortunately, this won't work reliably in a concurrent program. Multiple tasks may call procedure `Enter` simultaneously, indirectly making the `Log_File` object a shared resource. Race conditions are therefore possible when updating the `Log_File` object via `Put_Line`. We could employ a protected object to prevent the race condition, but as we saw with protected procedure `Dissemble` above, placing the call to `Put_Line` within a protected operation is not portable. We need some other way to ensure mutually exclusive access to the shared file object.

In some programming languages, a *mutex* is used to provide mutually exclusive access (hence the name) to some set of objects that are shared among multiple competing threads. All these threads must follow the same usage pattern:

1. before accessing the shared resource, a thread calls a routine on the mutex in order to block until it is appropriate to continue,
2. upon return from the call that thread executes an arbitrary sequence of statements accessing the resource,
3. after that sequence, the thread calls another operation on the same mutex to signal that some other thread can now be allowed to return from their call in step one.

A mutex must be implemented so that, for any given mutex object, only one caller at a time is allowed to return from the call in step one. Therefore, step one is said to acquire or seize the mutex object, and step three releases it. The result is that only one thread at a time will execute the statements in step two, hence with mutually exclusive access to the manipulated resources.

We can create a protected type providing a basic mutex abstraction. The protected operations will consist of two routines: one to acquire the mutex (step one) and one to release it (step three). Calls to these two PO operations can then bracket an application-specific sequence of statements that manipulate objects requiring mutually exclusive access (step two). But now this bracketed code can include some potentially blocking operations.

```
protected type Mutex is
   entry Acquire;
   procedure Release;
private
   Available : Boolean := True;
end Mutex;


protected body Mutex is

   entry Acquire when Available is
   begin
      Available := False;
   end Acquire;

   procedure Release is
   begin
      Available := True;
   end Release;
```

(continues on next page)

(continued from previous page)

```
end Mutex;
```

Tasks that want to exclusively acquire an object of the Mutex type will call entry Acquire on that PO. Similarly, tasks call protected procedure Release to signal that the logical hold on the PO is no longer required. The component Available is declared within the protected private part, and exists only to implement the mutex semantics.

The gist of the implementation is that acquiring the Mutex object amounts to allowing a caller task to exit their call to the entry Acquire, with any other caller tasks held pending. The entry barrier condition expresses the logic of whether the caller is allowed to continue, via the internal Boolean component Available. There is no actual lock in view here, just the effect of a lock. That effect is achieved via condition synchronization that ensures only one task at a time can return from the Acquire call. All other callers to Acquire are held, suspended, in the entry's queue. When Release is called that Mutex protected object becomes available for locking again.

The following code fragment illustrates using the Mutex type for the sake of controlling access to a shared variable, in this case the file object in the message logging package. Here is the pertinent part of the logging facility's package body:

```
Log_Lock : Mutex;
Log_File : File_Type;

procedure Enter (Log_Entry : String) is
begin
   Log_Lock.Acquire;
   Put_Line (Log_File, Log_Entry);
   Flush (Log_File);
   Log_Lock.Release;
end Enter;
```

The body of procedure Enter first calls Log_Lock.Acquire. The call is not allowed to return until the caller task exclusively holds the logical lock associated with the Log_Lock object. Therefore, every subsequent statement executes with mutual exclusion relative to the Log_Lock object. In this case, there are two such statements, the one that writes the string to the single shared output file and one that flushes any internal buffers associated with the file. They are both potentially blocking operations, but we're not in a protected operation so that's not a problem. Finally, procedure Enter calls Log_Lock.Release to relinquish the current caller task's hold on the Log_Lock mutex. If some other task was waiting to hold the Log_Lock object, that task can now return from its call to Acquire and can execute its update to the log file.

There are issues unaddressed in the three-step client protocol illustrated by the code above, especially error cases. For example, even if an exception is raised in step two, we need to ensure that Release is called with exactly-once semantics. There are other abstractions that address these client usage issues, namely scope-based locking, but we'll ignore them here. See the *Resource Acquisition Is Initialization* (page 103) idiom for the Scope_Lock type.

The implementation of type Mutex above doesn't have quite the full canonical semantics. So far it is really just that of a binary semaphore. In particular, a mutex should only be released by the same task that previously acquired it, i.e., the current owner. We can implement that consistency check in a fuller illustration of this example, one that raises an exception if the caller to Release is not the current owner of the Mutex object.

The new version of type Mutex is declared as follows. The difference is the additional component of type Ada.Task_Identification.Task_Id named Current_Owner. (Assume a use-clause for that package.)

```
protected type Mutex is
   entry Acquire;
   procedure Release;
private
   Available     : Boolean := True;
   Current_Owner : Task_Id := Null_Task_Id;
end Mutex;
```

The updated protected body is as follows:

```
protected body Mutex is

   entry Acquire when Available is
   begin
      Available := False;
      Current_Owner := Acquire'Caller;
   end Acquire;

   procedure Release is
   begin
      if Current_Owner = Current_Task then
         Available := True;
         Current_Owner := Null_Task_Id;
      else
         raise Protocol_Error;
      end if;
   end Release;

end Mutex;
```

Note how entry `Acquire`, when granting the logical lock and releasing the caller, also captures the identity of that caller. Procedure `Release` can verify that identity when it is eventually called, using function `Current_Task` declared in package `Ada.Task_Identification`.

We can build on that version of the type `Mutex` to make a variation named `Reentrant_Mutex`. This type allows a given task to re-acquire a `Reentrant_Mutex` object if that same task is the current owner, i.e., has returned from a previous call to `Acquire` and has not yet called `Release` a matching number of times:

```
protected type Reentrant_Mutex is
   entry Acquire;
   procedure Release;
private

   entry Retry;
   --  Internal target of requeue when the mutex is already owned.

   Depth : Natural := 0;
   --  Number of calls to Seize for a given holder. A value of zero
   --  corresponds no task currently holding the mutex.

   Current_Owner : Task_Id := Null_Task_Id;
   --  The current holder of the mutex, initially none.

end Reentrant_Mutex;
```

We still have the `Current_Owner` component, but we've added a new component to keep track of the *depth* of the current owner's calls. The depth test replaces the simple Boolean test of being available, so the `Available` component is gone. Instead, when the depth is zero the corresponding protected object is available, but it is also available if the current caller of `Acquire` is the current owner from a previous call.

```
protected body Reentrant_Mutex is

   entry Acquire when True is
   begin
      if Current_Owner = Null_Task_Id then
         Current_Owner := Acquire'Caller;
         Depth := 1;
      elsif Current_Owner = Acquire'Caller then
         Depth := Depth + 1;
      else -- held already, but not by current caller
         requeue Retry with abort;
      end if;
   end Acquire;

   procedure Release is
   begin
      if Current_Owner = Current_Task then
         Depth := Integer'Max (0, Depth - 1);
         if Depth = 0 then
            Current_Owner := Null_Task_Id;
         end if;
      else
         raise Protocol_Error;
      end if;
   end Release;

   entry Retry when Depth = 0 is
   begin
      Depth := 1;
      Current_Owner := Retry'Caller;
   end Retry;

end Reentrant_Mutex;
```

The barrier for entry `Acquire` is always set to **True** because the test for availability is not possible until the body begins to execute. If the PO is not available, the caller task is requeued onto the `Retry` entry. (A barrier set to **True** like this is a strong indicator of a requeue operation.) The `Retry` entry will allow a caller to return — from the caller's point of view, a return from the call to `Acquire` — only when no other caller currently owns the PO.

The examples so far exist primarily for providing mutual exclusion to code that includes potentially blocking operations. By no means, however, are these the only examples. Much more sophisticated abstractions are possible.

For example, let's say we want to have a notion of *events* that application tasks can await, suspending until the specified event is *signaled*. At some point, other tasks will signal that these events are ready to be handled by the waiting tasks. Understand that events don't have any state of their own, they either have happened or not, and may happen more than once.

For the sake of discussion let's declare an enumeration type representing four possible events:

```
type Event is (A, B, C, D);
```

These event names are not very meaningful, but they are just placeholders for those that applications would actually define. Perhaps a submersible's code would have events named `Hatch_Open`, `Hatch_Closed`, `Umbilical_Detached`, and so on.

Client tasks can suspend, waiting for an arbitrary event to be signaled, and other tasks can signal the occurrence of events, using a *event manager* that the two sets of tasks reference.

Here's the declaration of the manager type:

```
type Manager is limited private;
```

Type Manager will be fully implemented in the package private part as a protected type.

The type is limited because it doesn't make sense for clients to assign one Manager object to another, nor to compare them via predefined equality. There's another reason you'll see shortly. The type is private because that's the default for good software engineering, and there's no reason to override that default to make the implementation visible to clients. Our API will provide everything clients require, when combined with the capabilities provided by any limited type (e.g., declaring objects, and passing them as parameters).

Tasks can wait for a single event to be signaled, or they can wait for one of several. Similarly, tasks can signal one or more events at a time. Such groups of events are easily represented by an unconstrained array type:

```
type Event_List is array (Positive range <>) of Event;
```

We chose **Positive** as the index subtype because it allows a very large number of components, far more than is likely ever required, and has an intuitive default lower bound of 1. An aggregate value of the array type can then represent multiple events, for example:

```
Event_List'(A, C, D)
```

Given these three types we can define a useful API:

```
procedure Wait
  (This         : in out Manager;
   Any_Of_These :        Event_List;
   Enabler      :    out Event);

procedure Wait
  (This     : in out Manager;
   This_One : Event);

procedure Signal
  (This         : in out Manager;
   All_Of_These : Event_List);

procedure Signal
  (This     : in out Manager;
   This_One : Event);
```

Here's a task that waits for either event A or B, using a global Controller variable of the Manager type:

```
task body A_or_B_Processor is
   Active : Event;
begin
   loop
      Wait (Controller,
            Any_Of_These => Event_List'(A, B),
            Enabler => Active);
      Put_Line ("A_or_B_Processor responding to event " &
                Active'Image);
   end loop;
end A_or_B_Processor;
```

When the call to Wait returns, at least one of either A or B has been signaled. One of those signaled events was selected and returned in the Enabler parameter. That selected event is no longer signaled when the call returns and will stay that way until another call to procedure Signal changes it. The other event in the list is not affected, whether or not it was also signaled.

A signaling task could use the API to signal one event:

```
Signal (Controller, This_One => B);
```

or just:

```
Signal (Controller, B);
```

To signal multiple events:

```
Signal (Controller, All_Of_These => Event_List'(A, C, D));
```

Now let's consider the Manager implementation. As this is a concurrent program, we need it to be thread-safe. We've declared the Manager type as limited, so either a task type or a protected type would be allowed as the type's completion. (That's the other reason the type is limited.) There's no need for this manager to do anything active, it just suspends some tasks and resumes others when invoked. Therefore, a protected type will suffice, rather than a task's active thread of control.

Clearly, tasks that await events must block until a requested event has been signaled, assuming it was not already signaled when the call occurred, so a protected procedure won't suffice. Protected procedures only provide mutual exclusion, whereas protected entries can suspend a caller on a condition. Therefore, we'll use a protected entry for the waiters to call. As you will see later, there is another reason to use protected entries here.

Inside the Manager protected type we need a way to represent whether events have been signaled. We can use an array of Boolean components for this purpose, with the events as the indexes. For any given event index value, if the corresponding array component is **True** that event has been signaled, otherwise it has not.

```
type Event_States is array (Event) of Boolean;

Signaled : Event_States := (others => False);
```

Thus, for example, if Signaled (B) is **True**, a task that calls Wait for B will be able to return. Otherwise, that task will be blocked and cannot return from the call. Later another task will set Signaled (B) to **True**, and then the waiting task can be unblocked.

Since an aggregate can also contain only one component if desired, we can use a single set of protected routines for waiting and signaling in the Manager protected type. We don't need one set of routines for waiting and signaling a single event, and another set of routines for waiting and signaling multiple events. Here then is the visible part:

```
protected type Manager is

   entry Wait
     (Any_Of_These : Event_List;
      Enabler      : out Event);

   procedure Signal (All_Of_These : Event_List);

private
   ...
end Manager;
```

Both the entry and the procedure take an argument of the array type, indicating one or more client events. The entry, called by waiting tasks, also has an output argument, Enabler, indicating which specific event enabled the task to resume, i.e., which event was found signaled and was selected to unblock the task. We need that parameter because the task may have specified that any one of several events would suffice, and more than one could have been signaled.

The bodies of our API routines are then just calls into the protected Manager that is passed as the first argument. For example, here are two of the four:

```
procedure Wait
  (This         : in out Manager;
   Any_Of_These :        Event_List;
   Enabler      :    out Event)
is
begin
   This.Wait (Any_Of_These, Enabler);
end Wait;

procedure Signal
  (This     : in out Manager;
   This_One : Event)
is
begin
   This.Signal (Event_List'(1 => This_One));
end Signal;
```

Now let's examine the implementation of the protected type. The visible part is repeated here:

```
protected type Manager is

   entry Wait
     (Any_Of_These : Event_List;
      Enabler      : out Event);

   procedure Signal (All_Of_These : Event_List);

private
   ...
end Manager;
```

The entry Wait suspends callers until one of the requested events is signaled, as specified by the argument Any_Of_These. Normally we'd expect to use the entry barrier to express this behavior by querying the events' state array. If one of the requested events is **True** the barrier would allow the call to execute and complete. However, barriers do not have compile-time visibility to the entry parameters, so the parameters cannot be referenced in the barriers. This situation calls for a requeue statement.

Because Wait always takes a call, the entry barrier is just hard-coded to **True**. (As mentioned earlier, that's always a strong indication that requeue is involved.) Even though this barrier always allows a call, much like a protected procedure, we must use an entry because only protected entries can requeue callers.

Inside the entry body the specified events' states are checked, looking for one that is **True**. If one is found, the entry body completes and the caller returns to continue further, responding to the found event. If no requested event is **True**, though, we cannot let the caller continue. We block it by requeueing the caller on to another entry. Eventually that other entry will allow the caller to return, when an awaited event finally becomes **True** via Signal.

Here then is the full declaration for the protected type Manager, including the array type declaration that cannot be internal to the protected type:

```
type Event_States is array (Event) of Boolean;
```

(continues on next page)

(continued from previous page)

```
protected type Manager is

   entry Wait
     (Any_Of_These : Event_List;
      Enabler      : out Event);

   procedure Signal (All_Of_These : Event_List);

private

   Signaled          : Event_States := (others => False);
   Prior_Retry_Calls : Natural := 0;

   entry Retry
     (Any_Of_These : Event_List;
      Enabler      : out Event);

end Manager;
```

The private part contains the event states component, a management component, and the other entry, `Retry`, onto which we will requeue when necessary. Note that this other entry is only meant to be called by a requeue from the visible entry `Wait`, so we declare it in the private part to ensure there are no other calls to it. That informs the reader, but also the maintainer, who in the future might be tempted to call it in some other context.

The routine that checks for an existing signaled event is internal to the protected type and is declared as follows:

```
procedure Check_Signaled
  (These   : Event_List;
   Enabler : out Event;
   Found   : out Boolean);
```

The procedure examines the events specified in the formal parameter These to see if any of them are currently signaled, i.e., have a value of **True**. If it finds one, `Enabler` is set to that event value and Found is set to **True**. Otherwise, Found is set to **False** and `Enabler` is set to the value `Event'First`. The value assigned to `Enabler` in that case is arbitrary, but the assignment itself is important. Assigning a value prevents the actual parameter from becoming undefined upon return. `Enabler` will only be evaluated when Found returns **True** so the arbitrary value will be ignored.

Here's the body of the entry `Wait`, containing a call to `Check_Signaled` and the requeue statement. Note that the formal parameter `Wait.Enabler` is passed as the actual parameter to `Check_Signaled.Enabler`.

```
entry Wait
  (Any_Of_These : Event_List;
   Enabler      : out Event)
when
   True
is
   Found_Awaited_Event : Boolean;
begin
   Check_Signaled (Any_Of_These, Enabler, Found_Awaited_Event);
   if not Found_Awaited_Event then
      requeue Retry;
   end if;
end Wait;
```

The hard-coded entry barrier (**when True**) always allows a caller to execute, subject to the mutual exclusion requirement. If `Check_Signaled` doesn't find one of the specified events signaled, we requeue the caller to the Retry entry. (The Wait entry parameters go to the Retry entry, transparently.) On the other hand, if Check_Signaled did find a specified event signaled, we just exit the entry, the formal parameter Enabler having been set already by the call to the internal procedure.

Eventually, presumably, an awaited **False** event will become **True**. That happens when Signal is called:

```
procedure Signal (All_Of_These : Event_List) is
begin
   for C of All_Of_These loop
      Signaled (C) := True;
   end loop;
   Prior_Retry_Calls := Retry'Count;
end Signal;
```

After setting the specified events' states to **True**, Signal captures the number of queued callers waiting on Retry. (The component Prior_Retry_Calls is an internal component declared in the protected type. The value is never presented to callers, but is, instead, used only to manage callers.)

At long last, here's the body of Retry:

```
entry Retry
  (Any_Of_These : Event_List;
   Enabler      : out Event)
when
   Prior_Retry_Calls > 0
is
   Found_Signaled_Event : Boolean;
begin
   Prior_Retry_Calls := Prior_Retry_Calls - 1;
   Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
   if not Found_Signaled_Event then
      requeue Retry;
   end if;
end Retry;
```

When a protected procedure or entry completes their sequence of statements, the run-time system re-evaluates all the object's entry barriers, looking for an open (**True**) barrier with a caller queued, waiting. If one is found, that entry body is allowed to execute on behalf of that caller. On exit, the evaluation / execution process repeats. This process is known as a protected action and is one reason protected objects are so expressive and powerful. The protected action continues iterating, executing enabled entry bodies on behalf of queued callers, until either no barriers are open or no open barriers have callers waiting. Note that one of these entries may enable the barrier condition of some other entry in that same PO.

Therefore, when procedure Signal sets Prior_Retry_Calls to a value greater than zero and then completes, the protected action allows Retry to execute. Furthermore, Retry continues to execute, attempting to service all the prior callers in the protected action, because its barrier becomes **False** only when all those prior callers have been serviced.

For each caller, Retry attempts the same thing Wait did: if a requested event is **True** the caller is allowed to return from the call. Otherwise, the caller is requeued onto Retry. So yes, Retry requeues the caller onto itself! Doing so is not inherently a problem, but in this particular case a caller would continue to be requeued indefinitely when the requested event is **False**, unless something prevents that from happening. That's the purpose of the count of prior callers. Only that number of callers are executed by the body of Retry in the protected action. After that the barrier is closed by Prior_Retry_Calls becoming zero, the protected action ceases when the entry body exits, and any unsatisfied callers remain queued.

All well and good, but have you noticed the underlying assumption? The code assumes that unsatisfied callers are placed onto the entry queue at the end of the queue, i.e., in FIFO order. Consequently, they are not included in the value of the Prior_Retry_Calls count and so do not get executed again until Signal is called again. But suppose we have requested elsewhere that entry queues (among other things) are ordered by caller priority? We'd want that for a real-time system. But then a requeued caller would not go to the back of the entry queue and could, instead, execute all over again, repeatedly, until the prior caller count closed the entry.

If priority queuing might be used, we must change the internal implementation so that the queuing policy is irrelevant. We'll still have Wait do a requeue when necessary, but no requeue will ever go to the same entry executing the requeue statement. Therefore, the entry queuing order won't make a difference. An entry family facilitates that change, and rather elegantly, too.

An entry family is much like an array of entries, each one identical to the others. To work with any one of the entries we specify an index, as with an array. For example, here's a requeue to Retry as a member of an entry family, with Active_Retry as the index:

```
requeue Retry (Active_Retry)
```

In the above, the caller uses the value of Active_Retry as an index to select a specific entry in the family.

The resulting changes to the Manager type are as follows:

```
type Retry_Entry_Id is mod 2; --  hence 0 .. 1
type Retry_Barriers is array (Retry_Entry_Id) of Boolean;

protected type Manager is
   ... as before
private

   Signaled      : Event_States := (others => False);
   Retry_Enabled : Retry_Barriers := (others => False);
   Active_Retry  : Retry_Entry_Id := Retry_Entry_Id'First;

   entry Retry (Retry_Entry_Id)
     (Any_Of_These : Event_List;
      Enabler      : out Event);

end Manager;
```

Our entry family index type is Retry_Entry_Id. We happen to need two entries in this implementation, so a modular type with two values will suffice. Modular arithmetic will also express the logic nicely, as you'll see. The component Active_Retry is of this type, initialized to zero.

The entry Retry is now a family, as indicated by the entry declaration syntax specifying the index type Retry_Entry_Id within parentheses. Each entry has the same parameters as any others in the family, in this case the same parameters as in the previous implementation.

We thus have two Retry entries so that, at any given time, one of the entries can requeue onto the other one, instead of onto itself. An entry family makes that simple to express.

At runtime, one of the Retry entries will field requeue calls from Wait and will be the entry enabled by Signal. That entry is designated the *active* retry target, via the index held in the component Active_Retry.

Here's the updated body of Wait:

```
entry Wait
  (Any_Of_These : Event_List;
```

(continues on next page)

(continued from previous page)

```
   Enabler      : out Event)
when
   True
is
   Found_Signaled_Event : Boolean;
begin
   Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
   if not Found_Signaled_Event then
      requeue Retry (Active_Retry) with abort;
   end if;
end Wait;
```

The body is as before, except that the requeue target depends on the value of `Active_Retry`. (We'll discuss **with** abort later.)

When `Signal` executes, it now enables the *active retry* entry barrier:

```
procedure Signal (All_Of_These : Event_List) is
begin
   for C of All_Of_These loop
      Signaled (C) := True;
   end loop;
   Retry_Enabled (Active_Retry) := True;
end Signal;
```

The barrier component `Retry_Enabled` is now an array, using the same index type as the entry family.

The really interesting part of the implementation is the body of `Retry`, showing the expressive power of the language. The entry family member enabled by `Signal` goes through all its pending callers, attempting to satisfy them and requeuing those that it cannot. But instead of requeuing onto itself, it requeues them onto the other entry in the family. As a result, the ordering of the queues is immaterial. Again, the entry family makes this easy to express:

```
entry Retry (for K in Retry_Entry_Id)
  (Any_Of_These : Event_List;
   Enabler      : out Event)
when
   Retry_Enabled (K)
is
   Found_Signaled_Event : Boolean;
begin
   Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
   if Found_Signaled_Event then
      return;
   end if;
   --  otherwise...
   if Retry (K)'Count = 0 then  --  current caller is last one present
      --  switch to the other Retry family member for
      --  subsequent retries
      Retry_Enabled (K) := False;
      Active_Retry := Active_Retry + 1;
   end if;
   --  NB: K + 1 wraps around to the other family member
   requeue Retry (K + 1) with abort;
end Retry;
```

Note the first line:

```
entry Retry (for K in Retry_Entry_Id)
```

as well as the entry barrier (before the reserved word **is**):

```
when Retry_Enabled (K)
```

K is the entry family index, in this case iterating over all the values of `Retry_Entry_Id` (i.e., 0 .. 1).

We don't have to write a loop checking each family member's barrier; that happens automatically, via K. When a barrier at index K is found to be **True**, that corresponding entry can execute a prior caller.

Note the last statement, the one performing the requeue:

```
requeue Retry (K + 1) with abort;
```

Like the `Active_Retry` component, the index K is of the modular type with two possible values, so `K + 1` is always the *other* entry of the two. The addition wraps around, conveniently. As a result, the requeue is always onto the other entry, never itself, so the entry queue ordering makes no difference.

The **with** `abort` syntax can be read as "with abort enabled for the requeued caller task." Ordinarily, an aborted task that is suspended on an entry queue is removed from that queue. That removal is allowable in this version of protected type `Manager`, unlike the earlier FIFO version, because we are not using the count of prior callers to control the number of iterations in the protected action involving `Retry`. In the FIFO implementation we could not allow requeued callers to be removed from the `Retry` queue because the count of prior callers would no longer match the number of queued callers actually present. The protected action would await a caller that would never execute. In this more robust implementation that cannot happen, so it is safe to allow aborted tasks to be removed from the `Retry` queue.

Note that we do still check the count of pending queued callers, we just don't capture it and use it to control the number of iterations in the protected action. If we've processed the last caller for member K, we close member K's barrier immediately and then set the active member index to the other entry member. Consequently, a subsequent call to `Wait` will requeue to the other entry family member and `Signal` will, eventually, enable it.

Because we did not make the implementation visible to the package's clients, our internal changes will not require users to change any of their code.

Note that both the Ravenscar and Jorvik profiles allow entry families, but Ravenscar allows only one member per family because only one entry is allowed per protected object. Such an entry family doesn't provide any benefit over a single entry declaration. Jorvik allows multiple entry family members because it allows multiple entries per protected object. However, neither profile allows requeue statements, for the sake of simplifying the underlying run-time library implementation.

The full version using the entry-family approach is provided at the end of this text. Note that we have used a generic package so that we can factor out the specific kind of events involved, via the generic formal type. As long as the generic actual type is a discrete type the compiler will be happy. That correspondence is essential because we use the event type as an index for the array type `Event_States`.

```
generic
   type Event is (<>);
package Event_Management is

   type Manager is limited private;

   ...

private
```

(continues on next page)

(continued from previous page)

```
   type Event_States is array (Event) of Boolean;
   ...
end Event_Management;
```

Here is a small demonstration program. As before, we just have some simple event names to await and signal. We instantiate the generic package Event_Management with that Event type, and also the generic package Ada.Numerics.Discrete_Random so that we can randomly generate events to test the Event_Management instance.

```
--  Make the protected entry queues not be FIFO ordered, to
--  demonstrate that the type Manager handles this case too.
pragma Queuing_Policy (Priority_Queuing);

with Ada.Text_IO;   use Ada.Text_IO;
with Event_Management;

with Ada.Numerics.Discrete_Random;

procedure Demo_Events is

   type Event is (A, B, C, D);

   package Events is new Event_Management (Event);
   use Events;

   package Arbitrary_Event is
     new Ada.Numerics.Discrete_Random (Event);
   use Arbitrary_Event;

   G : Arbitrary_Event.Generator;

   Controller : Events.Manager;

   --  Tasks to await the events being signaled.
   --
   --  We give them priorities to exercise the priority-based
   --  implementation, but the values are arbitrary.

   task A_or_B_Processor with Priority => 5;
   task C_Processor      with Priority => 6;
   task D_Processor      with Priority => 7;

   ----------------------
   -- A_or_B_Processor --
   ----------------------

   task body A_or_B_Processor is
      Active : Event;
   begin
      loop
         Wait (Controller,
               Any_Of_These => Event_List'(A, B),
               Enabler => Active);
         Put_Line ("A_or_B_Processor responding to event " &
                   Active'Image);
      end loop;
   end A_or_B_Processor;

   -----------------
   -- C_Processor --
```

(continues on next page)

(continued from previous page)

```
   -----------------

   task body C_Processor is
   begin
      loop
         Wait (Controller, C);
         Put_Line ("C_Processor responding to event C");
      end loop;
   end C_Processor;

   -----------------
   -- D_Processor --
   -----------------

   task body D_Processor is
   begin
      loop
         Wait (Controller, D);
         Put_Line ("D_Processor responding to event D");
      end loop;
   end D_Processor;

begin
   loop
      Signal (Controller, Random (G));
      --  The tasks have priority for the sake of realism
      --  since the queues are now ordered by priority.
      --  However, we don't want any one task to
      --  monopolize the output, so for the sake of the
      --  demonstration we give the other tasks time to
      --  suspend on their calls to Wait too. The delay
      --  also makes the output easier to read.
      delay 0.5;
   end loop;
end Demo_Events;
```

When executed, each task iteratively prints a message indicated that it is responding to one of the awaited events. One of the tasks waits for one of two specified events, and the other two tasks wait for a single event each. The main procedure signals events at random. The demo runs forever so you'll have to kill it manually.

Each task writes to `Standard_Output`. Strictly speaking, this tasking structure allows race conditions on that shared (logical) file, but this is just a simple demo of the event facility so it is not worth bothering to prevent them. For the same reason, we didn't declare a task type parameterized with a discriminant for those tasks that await a single event.

16.2.3 Concurrent Programming

Concurrent programming applications will likely use both idioms. Thread-safe buffers are extremely common, for example. But in addition, potentially blocking operations are sometimes necessary within regions of code that require mutually exclusive access.

16.2.4 Real-Time Programming

As we mentioned in the introduction, protected objects provide the same benefits for real-time programming as they provide for concurrent programming, albeit with additional semantics. Those additions include execution with a priority, in particular. Clients will assign a ceiling priority to each protected object, as described in the *System Implementation of PO Mutual Exclusion* (page 140) section above. The purpose is to limit the blocking experienced by tasks, along with other task interaction benefits on a uniprocessor.

The GNAT package hierarchy includes a thread-safe bounded buffer abstraction that can be used in real-time applications. The protected type is declared within a generic package, like so:

```
protected type Bounded_Buffer
  (Capacity : Positive;
   Ceiling  : System.Priority)
with
   Priority => Ceiling
is
   ...
private
    ...
end Bounded_Buffer;
```

The two discriminants allow the type to be parameterized when clients declare objects of the type. In this case, the `Capacity` discriminant will be given a value specifying the maximum number of `Element` values that the object should be able to contain. More to the point here, the `Ceiling` discriminant specifies the priority to be given to the protected object itself.

16.2.5 Embedded Systems Programming

In the canonical model for handling interrupts in Ada, the handlers are protected procedures. The enclosing PO is again given a priority, as for real-time programming, but now the priority level is that of the `Interrupt_Priority` subtype. The handlers are invoked by the hardware via the run-time library and execute at the priority specified. This is essentially use of the first idiom for protected objects, even though the encapsulated application data is hardware oriented.

That's the canonical model, and hence the portable approach, but other approaches are possible. For example, if the target and OS allow it, a developer can set up the system to directly vector to a non-protected procedure. However, doing so is not portable, loses the benefits of the integration with the priority scheme, and almost certainly includes limitations on the operations allowed within the procedure body.

For further discussion, see the *Interrupt Handling* (page 85) idiom entry.

16.3 Pros

Protected objects extend prior research in concurrent programming language constructs, specifically the monitor construct that replaces semaphores, mutexes, and condition variables. Condition synchronization is simply stated as a Boolean expression on an entry, with caller suspension and resumption handled automatically. The monitor's explicit mechanism for local caller suspension and resumption is no longer required. Furthermore, protected action semantics make the code simpler than the combination of condition variables with mutexes, including the need for the looping idiom in the PThreads model, because when the condition expressed by the barrier becomes **True** the awakened caller is guaranteed to hold the mutual exclusion lock.

Protected objects add asynchronous communication and synchronization to the existing synchronous mechanism of Ada 83, addressing a serious deficiency for both concurrent and real-time programming in Ada. They are also critical to embedded systems programming with Ada.

Most importantly, developers can use protected objects (types) to create just about any synchronization and communication protocol imaginable. Especially when combined with other language features, the result is a flexible, extremely expressive facility.

16.4 Cons

The private part of a protected definition can contain only declarations for protected operations and component declarations. This limitation leads to declarations, such as the array type `Event_States`, that are purely implementation artifacts but cannot be hidden inside the private part. These artifacts will usually be declared immediately before the protected object or type, thus making them compile-time visible to clients whenever the protected type or object is visible to clients. Note that anonymously-typed array objects are not allowed in the private part. You will understand why these limitations exist when you consider that protected objects, when first conceived, were known as *protected records*. They have only slightly more declarative options than those of record types.

Protected objects have more capabilities than semaphores, mutexes, and condition variables. As a consequence, they may have more run-time overhead, but not much. For the automatic mutual exclusion implementation, the expense can be literally zero when priorities are used instead of actual locks for the implementation.

16.5 Relationship With Other Idioms

None

16.6 Notes

1. Thanks to Andrei Gritsenko (Андрей Гриценко@disqus_VErl9jPNvR) for suggesting a nice simplification of the FIFO version of the event waiting facility.
2. For more on tasking and topics like this, including examples of the second idiom, see the book by Burns and Wellings, Concurrent and Real-Time Programming In Ada, Cambridge University Press, 2007. Yes, 2007, but it is excellent and remains directly applicable today. The implementation of the event manager is based on their `Resource_Controller` example, for example.

16.7 Full Code for `Event_Management` Generic Package

The full implementation of the approach that works regardless of whether the queues are FIFO ordered is provided below. Note that it includes some defensive code that we did not mention above, for the sake of simplicity. See in particular procedures `Signal` and `Wait` that take an `Event_List` as inputs.

When compiling this generic package, you may get warnings indicating that the use of parentheses for aggregates is an obsolete feature and that square brackets should be used instead. Feel free to ignore them. Parentheses are not obsolete, neither in a practical sense nor in the language standard's sense of being obsolescent. There are indeed cases where square brackets are better, or even required, but those situations don't appear here.

Listing 15: event_management.ads

```ada
--  This package provides a means for blocking a calling task
--  until/unless any one of an arbitrary set of "events" is
--  "signaled."

--  NOTE: this implementation allows either priority-ordered or
--  FIFO-ordered queuing.

generic
   type Event is (<>);
```

(continues on next page)

(continued from previous page)

```ada
package Event_Management is

   type Manager is limited private;

   type Event_List is array (Positive range <>) of Event;

   procedure Wait
     (This          : in out Manager;
      Any_Of_These  :        Event_List;
      Enabler       :    out Event)
   with
     Pre => Any_Of_These'Length > 0;
   --  Block until/unless any one of the events in Any_Of_These has
   --  been signaled. The one enabling event will be returned in the
   --  Enabler parameter, and is cleared internally as Wait exits.
   --  Any other signaled events remain signaled. Note that,
   --  when Signal is called, the events within the aggregate
   --  Any_of_These are checked (for whether they are signaled)
   --  in the order they appear in the aggregate. We use a precondition
   --  on Wait because the formal parameter Enabler is mode out, and type
   --  Event is a discrete type. As such, if there was nothing in the list
   --  to await, the call would return immediately, leaving Enabler's value
   --  undefined.

   procedure Wait
     (This     : in out Manager;
      This_One : Event);
   --  Block until/unless the specified event has been signaled.
   --  This procedure is a convenience routine that can be used
   --  instead of an aggregate with only one event component.

   procedure Signal
     (This          : in out Manager;
      All_Of_These  : Event_List);
   --  Indicate that all of the events in All_Of_These are now
   --  signaled. The events remain signaled until cleared by Wait.
   --  We don't use a similar precondition like that of procedure
   --  Wait because, for Signal, doing nothing is what the empty
   --  list requests.

   procedure Signal
     (This     : in out Manager;
      This_One : Event);
   --  Indicate that event This_One is now signaled. The event
   --  remains signaled until cleared by Wait. This procedure is a
   --  convenience routine that can be used instead of an aggregate
   --  with only one event component.

private

   type Event_States is array (Event) of Boolean;

   type Retry_Entry_Id is mod 2;

   type Retry_Barriers is array (Retry_Entry_Id) of Boolean;

   protected type Manager is
      entry Wait
        (Any_Of_These : Event_List;
         Enabler      : out Event);
      procedure Signal (All_Of_These : Event_List);
```

(continues on next page)

(continued from previous page)

```ada
   private
      Signaled      : Event_States := (others => False);
      Retry_Enabled : Retry_Barriers := (others => False);
      Active_Retry  : Retry_Entry_Id := Retry_Entry_Id'First;
      entry Retry (Retry_Entry_Id)
        (Any_Of_These : Event_List;
         Enabler      : out Event);
   end Manager;

end Event_Management;
```

And the generic package body:

Listing 16: event_management.adb

```ada
package body Event_Management is

   ----------
   -- Wait --
   ----------

   procedure Wait
     (This         : in out Manager;
      Any_Of_These :        Event_List;
      Enabler      :    out Event)
   is
   begin
      This.Wait (Any_Of_These, Enabler);
   end Wait;

   ----------
   -- Wait --
   ----------

   procedure Wait
     (This     : in out Manager;
      This_One : Event)
   is
      Unused : Event;
   begin
      This.Wait (Event_List'(1 => This_One), Unused);
   end Wait;

   ------------
   -- Signal --
   ------------

   procedure Signal
     (This         : in out Manager;
      All_Of_These : Event_List)
   is
   begin
       --  Calling Manager.Signal has an effect even when the list
       --  is empty, albeit minor, so we don't call it in that case
      if All_Of_These'Length > 0 then
         This.Signal (All_Of_These);
      end if;
   end Signal;

   ------------
   -- Signal --
```

(continues on next page)

(continued from previous page)

```
------------

procedure Signal
  (This     : in out Manager;
   This_One : Event)
is
begin
   This.Signal (Event_List'(1 => This_One));
end Signal;

-------------
-- Manager --
-------------

protected body Manager is

   procedure Check_Signaled
     (These   : Event_List;
      Enabler : out Event;
      Found   : out Boolean);

   ----------
   -- Wait --
   ----------

   entry Wait
     (Any_Of_These : Event_List;
      Enabler      : out Event)
   when
      True
   is
      Found_Signaled_Event : Boolean;
   begin
      Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
      if not Found_Signaled_Event then
         requeue Retry (Active_Retry) with abort;
      end if;
   end Wait;

   ------------
   -- Signal --
   ------------

   procedure Signal (All_Of_These : Event_List) is
   begin
      for C of All_Of_These loop
         Signaled (C) := True;
      end loop;
      Retry_Enabled (Active_Retry) := True;
   end Signal;

   -----------
   -- Retry --
   -----------

   entry Retry (for K in Retry_Entry_Id)
     (Any_Of_These : Event_List;
      Enabler      : out Event)
   when
      Retry_Enabled (K)
   is
```

(continues on next page)

(continued from previous page)

```
         Found_Signaled_Event : Boolean;
      begin
         Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
         if Found_Signaled_Event then
            return;
         end if;
         --  otherwise...
         if Retry (K)'Count = 0 then -- current caller is last one
            --  switch to the other Retry family member for
            --  subsequent retries
            Retry_Enabled (K) := False;
            Active_Retry := Active_Retry + 1;
         end if;
         --  NB: K + 1 wraps around to the other family member
         requeue Retry (K + 1) with abort;
      end Retry;

      --------------------
      -- Check_Signaled --
      --------------------

      procedure Check_Signaled
        (These   : Event_List;
         Enabler : out Event;
         Found   : out Boolean)
      is
      begin
         for C of These loop
            if Signaled (C) then
               Signaled (C) := False;
               Enabler := C;
               Found := True;
               return;
            end if;
         end loop;
         Enabler := Event'First; -- arbitrary, avoids undefined value
         Found := False;
      end Check_Signaled;

   end Manager;

end Event_Management;
```