LEARN.
ADACORE.COM

Ada Idioms

Release 2024-09

Patrick Rogers

Sep 08, 2024

CONTENTS:

Warning

This version of the website contains UNPUBLISHED contents. Please do not share it externally!

This course will teach you the basics of the Ada programming language and is intended for those who already have a basic understanding of programming techniques. You will learn how to apply those techniques to programming in Ada.

This document was written by Patrick Rogers.

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[5]. 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;
2. Go to target directory;

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

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).

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

CHAPTER ONE

INTRODUCTION

CHAPTER

TWO

ESSENTIAL DESIGN IDIOMS FOR PACKAGES

2.1 Motivation

Packages, especially library packages, are modules, and as such they 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.

In particular, packages should exhibit high cohesion and loose coupling[1]. Cohesion is the degree to which the declarations within a module are related to one another, in terms of the problem being solved. In short, unrelated entities should not be declared in the same module so that the reader can focus on one primary concept. Coupling is the degree to which a module depends upon other modules. Loose coupling enhances comprehension and maintenance because it allows the developer/reader to examine and modify the module in relative isolation. Coupling and cohesion are interrelated, in that higher cohesion tends to result in less coupling.

2.2 Solution

Three idioms for packages were envisioned when the language was first designed. They were introduced and described in detail by the Rationale document for the initial language design[2]. They were then further developed in Grady Booch's book *Software Engineering with Ada*[3], 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 more maintainable source code.

These idioms pre-date later package facilities, such as private packages and hierarchical packages. Idioms for those packages will be described separately.

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

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

[1] Yourdon, E. and L. L. Constantine (1979). Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice-Hall.

[2] Ichbiah, J., J. Barnes, et al. (1986). Rationale for the Design of the Ada Programming Language.

[3] Booch, G. (1983). Software Engineering with Ada, Benjamin/Cummings Publishing Company.

2.3 Essential Idiom 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 the 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 applied with this idiom.

2.3.1 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.

2.3.2 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.

2.4 Essential Idiom 2: Groups of Related Program Units

In this idiom, the package can contain all of the declarations allowed by the first idiom, but will also contain declarations for operations, usually subprograms but other units are also allowed, e.g., protected types/objects. Hence:

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

The intent is that the types declared in the package are used by the operations, e.g., in the formal parameters and/or function return types. In particular, though, the types are not private types.

For example:

```ada
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 code, `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 6) package referenced in a with-clause. In any case this package declares types and subprograms that manipulate values of the types via parameters.

Variables might also be declared in the package, but not as the central purpose of the package. 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. For example:

```ada
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 implementations of the visible subprograms in the package body, as well as any internal entities declared in the body for the sake of implementing the visible subprograms.

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

```ada
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;
```

2.4.1 Pros

The types and the associated operations are grouped together, 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: they can create values via aggregates, for example, and can use array indexing to get to specific components.

2.4.2 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.

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. Ada — by design — does not allow developers to redefine extremely low-level operations such as array indexing. Consequently, those types must be compile-time visible to clients as array types. We could define functions as alternatives to indexing and aggregates, but would clients accept that relatively heavy approach?

2.5 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.

2.6 Bibliography

CHAPTER

THREE

ABSTRACT DATA TYPES

3.1 Motivation

In the previous idiom (the *Groups of Related Program Units* (page 7)), 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 the vast majority of cases, the resulting economic and engineering disadvantages far outweigh the advantages.

For the sake of illustration, let's make up 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 will 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 7) 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)[1],[2].

3.2 Solution

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 at the top files, then tracks, then sectors, then blocks, 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[2]:

- 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 visible to clients, but not the representation. 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

[1] Booch, G. (1983). Software Engineering with Ada, Benjamin/Cummings Publishing Company.

[2] Liskov, B. and S. Zilles (1974). Programming with Abstract Data Types. ACM SIGPLAN symposium on Very high level languages.

In general, at most one private type should be declared per ADT package, for the sake of simplicity. Note that the *limited-with* construct directly facilitates declaring mutually-dependent private types declared in their own dedicated packages. However, it is 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.

Any form of private type is allowed 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 is it not compile-time visible to clients.

The abstraction's operations will consist of subprograms that each have a formal parameter of the type. Clients will declare objects of the type and pass these objects to the formal parameters in order 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 cannot compile any operation that manipulates the internal representation directly.

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, ending with the word **`private`** to indicate that the 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 consistent with the rest of the visible type declaration, except for anything representation-dependent.

The full type definition occurs in the package private part. Therefore, for any given object of the type, the representation details — the two record components in this example — cannot be referenced in client code. Clients must instead use the operations defined by the package, passing the client objects to the formal 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 the 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 as the only direct manipulation possible. 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.

Other ancillary type declarations may of course also be required in the package, either for the implementation or as additional parameters for the visible operations. The array type Content is an example of such an ancillary type. In this case it is hidden from clients because it is strictly an implementation artifact.

The ADT idiom extends the information hiding applied by the *Groups of Related Program Units* (page 7) 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:

```
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, it's up to clients, so the constructor is required.

Likewise, functions `Real_Part` and `Imaginary_Part` are selector/accessor functions that return the corresponding individual component values, given an argument of type `Complex_Number`. They are provided because complex numbers have those two parts by definition in mathematics. 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 stacks 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, beyond 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 so the function would be implemented in some other way. (We would have declared a subtype of **Natural**, using Capacity as the upper bound, for the function result type.)

True *getter* functions that do not meet an abstraction-defined requirement and exist purely to provide client access to the otherwise hidden representation components should not be 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 result from the strong interface presented, with guaranteed enforcement by the compiler rather than by reliance on client good behavior. The ADT designer can rely on client adherence to the intended abstraction because client code that violates the designer's abstraction — 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, 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. Not every type should be private, for example those that are explicitly numeric. But the ADT should be the default design idiom when decomposing a problem into a solution.

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 7)). 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 5) 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 9) idiom, which is the primary type specification facility in Ada. Additional package-oriented idioms will be described, 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 10, 2]. The Ada language requirements document, *Steelman*[3], 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.

[3] HOLWG (1978). Department of Defense Requirements for High Order Computer Programming Language "STEELMAN".

3.7 Bibliography

CHAPTER

FOUR

ABSTRACT DATA MACHINES

4.1 Motivation

4.2 Solution

The Abstract Data Machine (ADM) is similar to the *Abstract Data Type* (page 9) in that it presents an abstraction, something that doesn't already exist in the programming language. Furthermore, like the ADT, operations are provided to manipulate the abstraction data, state that is not otherwise compile-time visible to client code. These operations are thus enforced as the only manipulation possible, as per the designer's intent. (The Abstract Data Machine was introduced by Booch as the Abstract State Machine, but that name can be confused with another concept that, though somewhat related, is not the same thing.)

Unlike the ADT, however, the ADM does not define the abstraction as a type. To understand this point, recall that type declarations are descriptions for objects that will contain data. For example, our earlier `Stack` type was defined 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. No data actually exist, i.e., are allocated storage, until objects are declared. Clients can declare as many objects of type `Stack` as they require, and each object has a distinct, separate copy of those two components.

Clients can, of course, choose to declare only one object of a given type, in which case only one instance of the data described by the type will exist. But in that case, other than convenience there is no functional difference from declaring objects of the component types directly, rather than indirectly via some enclosing type. Instead of using the `Stack` type to declare a single composite object, for example, the developer could have instead declared two objects, 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 only one *stack* these two objects will suffice.

That's what the ADM does. The necessary state for a single abstraction instance is declared in a package, usually a library package. 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 any abstraction would require. Hence the package is the abstraction instance, as opposed to one or more objects of a type.

Therefore, the package declaration's visible section will contain 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 especially one or the other (or both) will contain object 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;

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;
```

Now there is no type presenting a `Stack` abstraction, and the operations do not take a stack parameter because the package and its data is the abstraction instance. There is only one stack of integers with this idiom. That is why the name of the package is changed from `Integer_Stacks`, i.e., from the plural form.

As with the ADT idiom, clients of an ADM can only manipulate the encapsulated state indirectly, via the visible operations. The difference is that the state to be manipulated is no longer a formal parameter. For example:

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

That call places the value 42 in the 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 array `Answers.Values`, i.e., within the `Answers` variable. Clients can declare as many `Stack` objects as they require, and each will contain a distinct copy of the state defined by the type. In the ADM version there is only one stack and therefore one instance of the state.

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;
   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 discuss the effect of child packages separately.)

The private section wasn't required when the data were declared in the package body. That's typical when using this idiom but is not a necessary characteristic.

The ADM idiom applies information hiding to the internal state, similar to the ADT idiom, except that the state is not in objects. As well, like the *Groups of Related Program Units* (page 7), the implementations of the visible subprograms are hidden by the package body, along with any non-visible entities required for their implementation.

There will be no constructor functions returning a value of the abstraction type because there is no such type with the ADM. 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 because of the reasonable initial state, as is true with 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.

4.3 Pros

In terms of abstraction and information hiding, the ADM provides the same advantages as the ADT idiom: clients have no representation details visible 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 manipulation, it must be in the one package defining the abstraction itself. No other code would have the compile-time visibility necessary.

This idiom can be applied to any situation requiring abstraction, including hardware. For example, a particular microprocessor had an on-board rotary switch for arbitrary use by system designers. The switch value was 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 provided the current switch value.

However, the memory at that address was read-only, and rightly so because the only way to change the value was to physically rotate the switch. Writing to that address had no effect whatsoever. Although doing so was a logical error no indication was provided by the hardware. That silence was potentially confusing to developers. It certainly looked 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.

A good general rule is that if something shouldn't be done by clients, we should use the compiler to make it impossible. That's better than debugging, any day. Therefore, we will use the ADM idiom to represent the rotary switch. The compiler will enforce the read-only view and the 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
   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[6] 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.

4.4 Cons

An ADM defines only one abstraction instance. If more than one is required, the developer must copy-and-paste the entire package and then change the package unit name.

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 switch example, the ADM limitations are irrelevant. That said, certainty is usually not available — even the hardware changes.

[6] 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 9), an ancillary type might be required for the completion of the private type. That was the case with the *ADT version* (page 11) 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 proper.

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 package 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 Solution

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

Specifically, placement in the private part of the package will allow *programming by extension*[1] 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;
```

(continues on next page)

[1] Barnes, J. (1998). Programming In Ada 95, Addison-Wesley.

(continued from previous page)

```
   Capacity : constant := 100;
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;
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[7].

5.5 Relationship With Other Idioms

We assume the use of the *Abstract Data Type* (page 9) 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 a common 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 packages 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.

5.7 Bibliography

[7] 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.

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 9) (ADT) idiom. In particular, the parent type is a private type, and the derived type is declared as 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;
```

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

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. (That location provides the compile-time visibility to the type's representation that is required to implement the subprograms.) Only the primitive operations are inherited during type derivation.

Instead of a distinct constructor syntax, Ada uses regular functions to construct objects of types but the issues are the same. By definition, these so-called *constructor functions* return an object of the type in question. Other primitive functions are not constructors and can be inherited without difficulty.

Therefore, Ada language rules prevent constructor functions from being legally inherited, even though they are primitive operations for the type.

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 9) 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:

```
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:

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

In contrast, *concrete* types have an actual representation and can be used to declare objects. Likewise, concrete subprograms are fully implemented, callable units. In the following discussion, *abstract* has this OOP sense unless stated otherwise.

With that definition in place, 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. Assuming the extended child type is not abstract, the type extension will be illegal because only abstract types can have abstract subprograms. Thus, the compiler prevents this inappropriate constructor inheritance.

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

```
package Graphics is
   type Shape is tagged private;
   function Make (X, Y : Float) return Shape;
   ...
private
   type Shape is tagged record
      X : Float := 0.0;
      Y : Float := 0.0;
   end record;
end Graphics;
```

Note in particular the primitive function named `Make` that constructs a value of type `Shape`. The two formal parameters are assigned to the corresponding two components of the object returned by `Make`.

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 Solution

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

To prevent their being inherited, the solution 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 solution is to declare constructor functions in a separate package that is a *child* of the package declaring the tagged type. The actual term is a *hierarchical library package* but *child* conveys the concept and is less of a mouthful.

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 the latter will 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 solution given, 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 solution, 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.

```ada
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:

```ada
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:

```ada
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 solution.

CHAPTER

SEVEN

REDUCING OBJECT CODE FROM GENERIC PACKAGE INSTANTIATIONS

7.1 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.

7.2 Solution

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:

```ada
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:

```ada
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:

```ada
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.

```ada
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.

7.3 Pros

The total object code size is reduced, compared to the alternative of many local instantiations.

7.4 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.

7.5 Relationship With Other Idioms

None.

7.6 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

EIGHT

USING BUILDING BLOCKS TO EXPRESS INHERITANCE IDIOMS

8.1 Motivation

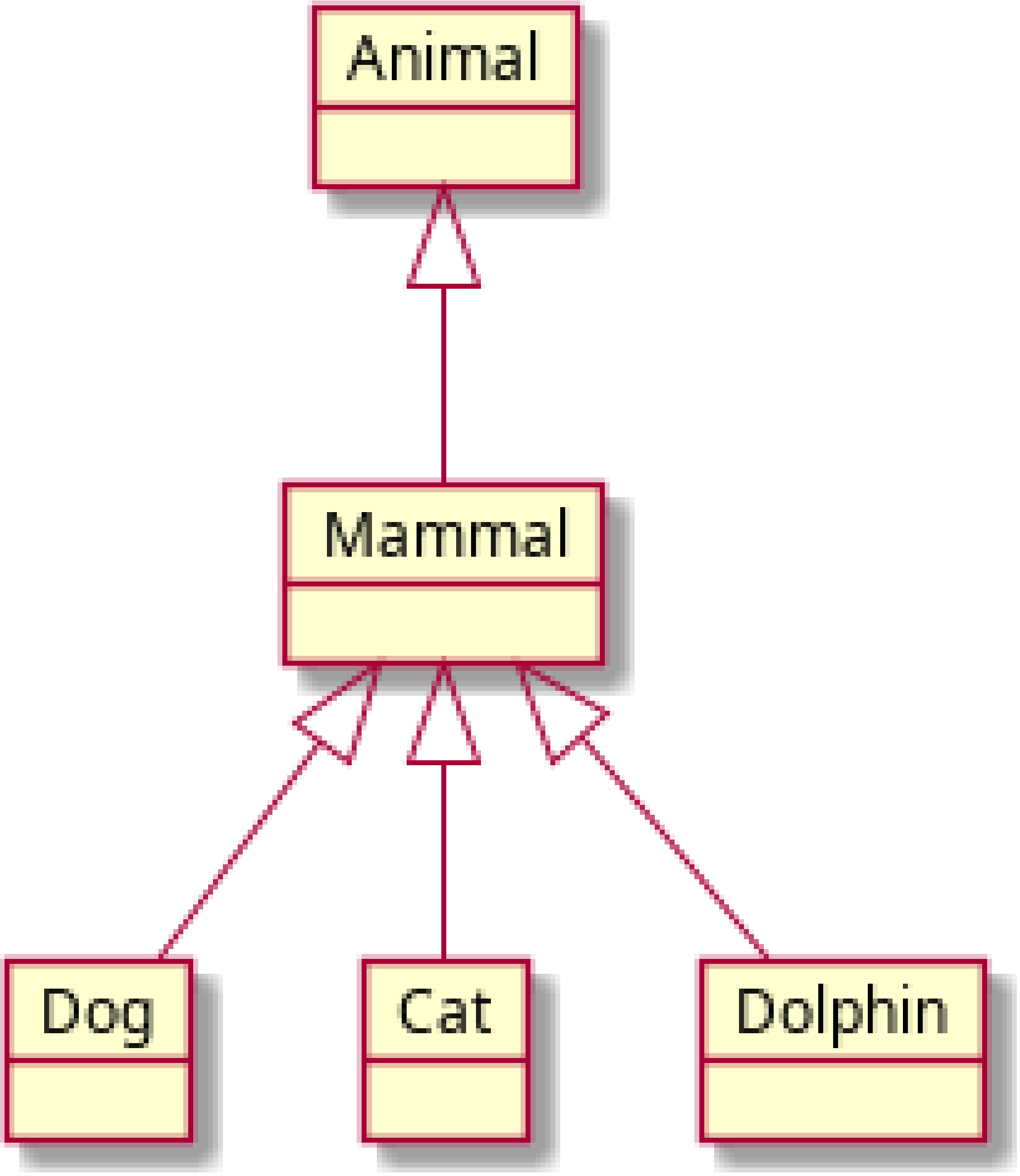
Betrand Meyer's magisterial book on OOP[1] includes a taxonomy of inheritance idioms. Two especially well-known entries in that taxonomy are Subtype Inheritance[8] and Implementation Inheritance[9]. The name of the first idiom is perhaps confusing from an Ada point of view because Ada subtypes have a 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 is 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:

[1] Meyer, B. (1997). Object-Oriented Software Construction, Prentice-Hall.

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

[9] 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 the same.

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 35, 1], 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 39) and *Implementation Inheritance* (page 42) 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 not visible to clients in both. In Subtype Inheritance clients do have compile-time visibility to the relationship, whereas in Implementation Inheritance clients do not 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.

8.1.1 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 9) 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 the beginning Ada has also supported a static form of inheritance, using types that are not tagged. The solution 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:

```ada
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:

```ada
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.

8.2 Solution

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

- We will use tagged types for the sake of the 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, however those cases are very narrow in applicability.
- We will 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 9) for justification and details.
- For the most general capabilities, we will assume that the parent type is visibly tagged.
- We are going to declare the child type in a distinct, dedicated package, following the *ADT idiom* (page 9). This package may or may not be a hierarchical child of the parent package. The solution 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 42) solution.

To avoid unnecessary code duplication, examples for the two idiom solutions use the same parent type, declared as a simple tagged private type. The parent type could itself be derived from some other tagged type, but that changes nothing conceptually significant. The parent type is declared 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;
```

8.2.1 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, automatically 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 be any overriding declarations for those inherited operations that are to be changed.

For example, here is a basic bank account *ADT* (page 9) that we will use as the parent type in a derivation:

Listing 1: 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
```

(continues on next page)

(continued from previous page)

```
     with Pre'Class => Open (This);

   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            : in String;
      Initial_Deposit : in 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 2: 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            : in String;
      Initial_Deposit : in 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)

```

   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 hierarchical 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:

```
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:

```
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:

```
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.

8.2.2 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:

```
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, automatically 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. Clients generally don't have any business directly calling the operations defined by the two language-defined controlled types so we usually use implementation inheritance. But if clients did have the need, we would use Subtype Inheritance instead.

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 3: unbounded_sequential_queues.ads

```
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 : in 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 writing 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 did explicitly write a body for Finalize that simply called Reset, but it avoids the code and the extra layer of subprogram call. Clients can call Reset whenever they want, but the procedure will also be called automatically, via Finalize, when any Queue object ceases to exist.

8.3 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.

8.4 Cons

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

8.5 Relationship With Other Idioms

We assume the *Abstract Data Type idiom* (page 9), 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.

8.6 Notes

8.7 Bibliography

CHAPTER

NINE

PROVIDING COMPONENT ACCESS TO ENCLOSING RECORD OBJECTS

9.1 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 the sake of concision 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 9). 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.

9.2 Solution

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

The solution 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 Enclosing is tagged limited private;
private

   protected type Controller (Instance : not null access Enclosing) is
      -- Part 1

      procedure Increment_X;
   end Controller;

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

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

end P;
```

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

```ada
protected type Controller (Instance : not null access Enclosing) is
```

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

But not just any object of type `Enclosing` will suffice. For Part #2, we must give the `Instance` 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 `Instance` is elided. The following is that code again, now showing just the declaration for Enclosing, but also including the construct that is actually passed. This is where the subtlety comes into play:

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

The subtlety is the expression Enclosing'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 Enclosing is elaborated, the name Enclosing 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 Enclosing, the expression Enclosing'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
         Instance.X := Instance.X + 1;
      end Increment_X;

   end Controller;

end P;
```

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

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

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

```
type Enclosing is tagged limited private;
```

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

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

The type need not be tagged for this idiom solution, but if you do make it tagged, the partial and full views must always match. That is, a tagged type must be limited in both views if it is limited in either view.

For the idiom solution to be legal, the type's completion in the private part must always be *immutably limited*, meaning that it is always truly limited. There are various ways to make that happen (see AARM22 7.5 (8.1/3)[10]) 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*.

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 the private full view:

```
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 explicitly limited really are limited, and always have a view as a limited type. That's important because the type given in `type_name'Access` must be aliased for `'Access` to be meaningful and possible on the corresponding objects. But if the type's view could change between limited and not limited, it would be aliased in some contexts and not aliased in others. To prevent that complexity, the language requires the type's view to be permanently limited so that the type will be permanently aliased. An immutably limited type is permanently aliased. In practice, we're working with record types and type extensions, so just make the type definition explicitly limited and all will be well:

```
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.

9.3 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[11]. 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 Ada Drivers Library (ADL)[12] 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.

[10] http://www.ada-auth.org/standards/22aarm/html/AA-7-5.html

[11] https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

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

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 53)):

```
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 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 35) 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);
   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);

      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:
```

(continues on next page)

(continued from previous page)

```
      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.

9.4 Pros

The solution 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 idiom solution itself requires little text. Interrupt handling is relatively complex in any language.

9.5 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.

9.6 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 idiom solution because there would be no enclosing record type. But as described in the ADM discussion, the *ADT approach* (page 9) is usually superior.

9.7 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.

9.8 Bibliography

None.

9.9 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[13].

First, the basic Serial_IO abstraction:

```ada
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;
```

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

And the package body:

```
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
```

(continues on next page)

(continued from previous page)

```
     (Transceiver  : not null access USART;
      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
      Controller : IO_Manager (IRQ, IRQ_Priority, Serial_Port'Access);
```

(continues on next page)

(continued from previous page)

```
      --  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;
         end if;
```

(continues on next page)

(continued from previous page)

```
         --  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

TEN

CONTROLLING OBJECT INITIALIZATION AND CREATION

10.1 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 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;
   Null_Tree : constant Tree;
   ...
private
   type Leaf_and_Branch is record ...
   type Tree is access Leaf_and_Branch;
   Null_Tree : constant Tree := null;
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. 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 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.

```
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, and 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 solutions.

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 solutions 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:

```
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.

```
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 25) 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[14] 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 solution using a real-world hardware device.

10.1.1 Requiring Initialization by Clients

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 solutions are type-specific, so when we say *objects* we mean objects of a type that has been designed with one of these two idiom solutions. Neither solution 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 9) 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 9), 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 solutions. The types are not necessarily limited as well, but in one situation they will be limited too. But in both solutions the primary types will be private types.

10.2 Solution 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,

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

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:

```
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 76) 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
```

(continues on next page)

(continued from previous page)

```
   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 solution 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[15]. 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 solution does exactly that.

The following is a cut-down version of the package declaration using this idiom solution, 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)[16]

[15] https://en.wikipedia.org/wiki/Proportional–integral–derivative_controller

[16] https://github.com/AdaCore/Robotics_with_Ada/blob/master/src/control_systems/

```
package Process_Control is

   type PID_Controller (<>) is tagged limited private;

   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 25). 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[17].

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.

```ada
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).

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

10.3 Solution 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.

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
```

(continues on next page)

(continued from previous page)

```
      This.Top := This.Top + 1;
      This.Content (This.Top) := Value;
   end Push;

end Bounded_Stacks;

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

(continues on next page)

(continued from previous page)

```
   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:

```
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 solution is not enforced at compile-time, it is nevertheless sufficient to ensure no uninitialized object of the type can be used.

10.3.1 Preventing Object Creation by Clients

The other idiom requirement is the ability to control object creation itself. The solution 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 solution.

For a concrete example, we can apply the Singleton design pattern to represent the *time stamp counter* (TSC[18]) 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[19], some care is required when using the register for that purpose on modern hardware, but it will suffice to illustrate the idiom solution. 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 solution.

```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
```

(continues on next page)

[18] https://en.wikipedia.org/wiki/Time_Stamp_Counter
[19] https://en.wikipedia.org/wiki/Time_Stamp_Counter

(continued from previous page)

```
   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.

   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

   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 21) 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_
```

(continues on next page)

(continued from previous page)

```
↪Count;
   function Second_Sample (This : not null access Timestamp_Sampler) return Cycle_
↪Count;
   function Elapsed       (This : not null access Timestamp_Sampler) return Cycle_
↪Count;

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

(continues on next page)

(continued from previous page)

```
   function Second_Sample (This : not null access Timestamp_Sampler) return Cycle_
↪Count is
     (This.Second);

   -------------
   -- 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[20] this way reliably.

10.4 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.

10.5 Cons

None.

10.6 Relationship With Other Idioms

The *Abstract Data Type* (page 9) is assumed, in the form of a private type.

10.7 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 solution, 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

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

that solution. Prior to that version an object of a limited type could not be initialized in the object's declaration.

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[21]. 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[22], is supported (which would require Annex C[23] 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[24], 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 solutions. In both cases notification amounts to releasing the previously suspended task for further execution.

In the Solution section below, 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.

[21] http://www.ada-auth.org/standards/12rm/html/RM-C-3.html

[22] http://www.ada-auth.org/standards/12rm/html/RM-D.html

[23] http://www.ada-auth.org/standards/12rm/html/RM-C.html

[24] https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

11.2 Solution

11.2.1 First Level Handler Alone

In this solution 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.

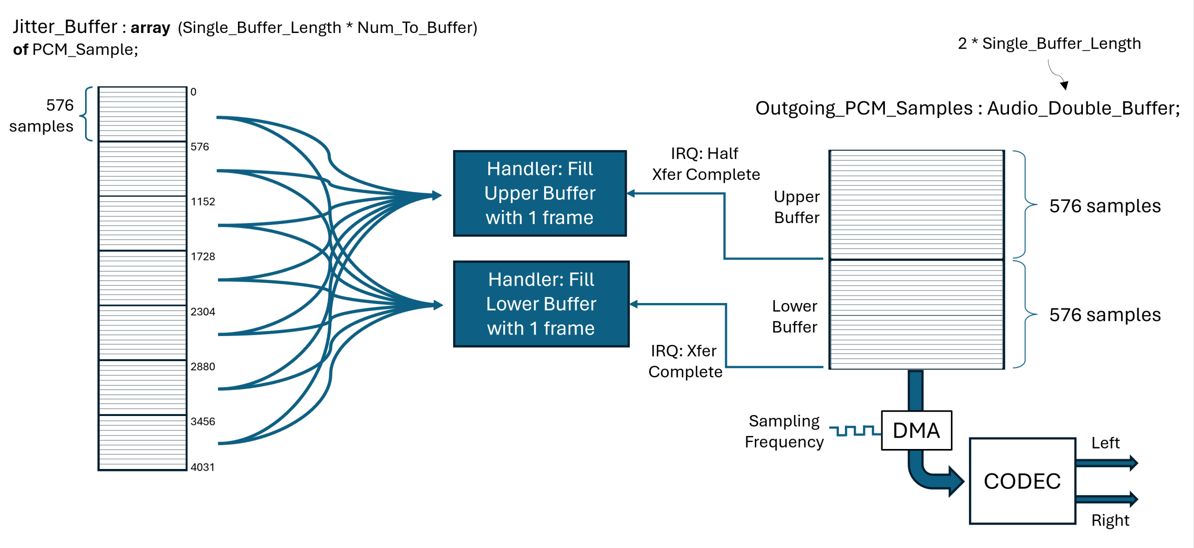
For a concrete example, consider an audio streaming device that takes PCM samples from Ethernet via incoming UDP packets and transfers them to an audio CODEC device[25] on the target board. The CODEC output is physically connected to a high-quality amplifier and speakers. No upper-level application thread requires notification of the copying: if the transfer is working the audio output occurs, otherwise it does not.

In our solution 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.

[25] 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.

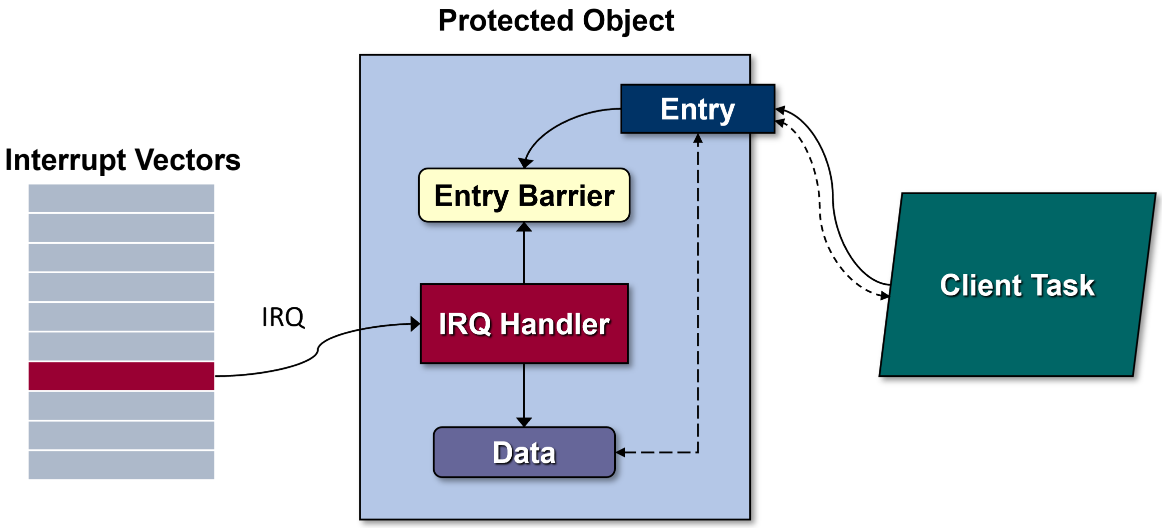
A related issue, perhaps requiring a run-time change, is the *secondary stack* allocation for interrupt handlers. The secondary stack is a common approach to implementing calls to functions that return values of unconstrained subtypes (usually, unconstrained array types, such as **String**). Because 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 solution did not require notifying a task, but these next idiom solutions 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 solution 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 45). In that solution, 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
```

(continues on next page)

(continued from previous page)

```
   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);

      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;

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

(continues on next page)

(continued from previous page)

```
   -- 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
            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;

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

(continues on next page)

(continued from previous page)

```
      -- 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;
```

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.

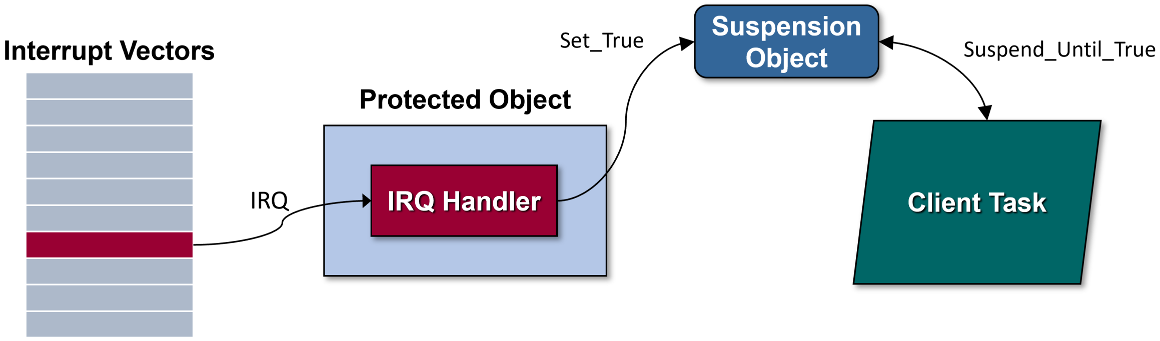
Using protected entries is ideal for this case because, after all, that is exactly what they are designed to do. Note that declaring multiple protected entries in a single protected type/object requires the Jorvik usage profile to be applied.

11.2.4 Task Notification Without Communication

In this solution, 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 idiom solution 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 solution we just explored, and thus to the *Providing Component Access to Enclosing Record Objects* (page 45) idiom. In that solution, 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 solution 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:

```ada
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
```

(continues on next page)

(continued from previous page)

```
   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
```

(continues on next page)

(continued from previous page)

```
      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 solution 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 solution.

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.

11.8 Bibliography

CHAPTER

TWELVE

TYPE PUNNING

12.1 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[26] 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."[1] 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, which must be explicit, involve only those types that the language defines as meaningfully convertible.

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

[1] S. Tucker Taft, 1993, post in Comp.Lang.Ada, https://groups.google.com/g/comp.lang.ada/c/9WXgvv8Xjuw/m/JMyo9_P7nxAJ

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.

12.2 Solution

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.

12.2.1 First Solution: 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 := 0;

   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 know that detail.

This expression of type-punning does not use an escape hatch but does achieve the effect.

12.2.2 Second Solution: 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:

```ada
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 9), 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.

```ada
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
```

(continues on next page)

(continued from previous page)

```
   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);
   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 9). 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:

```ada
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.

```ada
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[27] headers, TCP[28] headers, and so on.

12.3 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.

12.4 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.

12.5 Relationship With Other Idioms

None.

12.6 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[29].)

[27] https://en.wikipedia.org/wiki/Address_Resolution_Protocol

[28] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[29] https://learn.adacore.com/courses/advanced-ada/parts/resource_management/access_types.html#adv-ada-access-address

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.

12.7 Bibliography

CHAPTER

THIRTEEN

APPENDICES