Ada for the C++ or Java Developer

Release 1.0

Quentin Ochem, AdaCore

July 23, 2013

CONTENTS

CHAPTER

ONE

PREFACE

Nowadays it seems like talking about programming languages is a bit passé. The technical wars of the past decade have subsided and today we see a variety of high-level and well-established languages offering functionality that can meet the needs of any programmer.

Python, Java, C++, C#, and Visual Basic are recent examples. Indeed, these languages make it easier to write code very quickly, are very flexible, offer features with highly dynamic behavior, and some even allow compilers to deduce the developer's probable intent.

Why, then, talk about yet another language? Well, by addressing the general programming market, the aforementioned languages have become poorly suited for working within the domain of high-integrity systems. In highly reliable, secure and safe applications such as those found in and around airplanes, rockets, satellites, trains, and in any device whose failure could jeopardize human life or critical assets, the programming languages used must support the high standard of software engineering necessary to maintain the integrity of the system.

The concept of verification—the practice of showing that the system behaves and performs as intended—is key in such environments. Verification can be accomplished by some combination of review, testing, static analysis, and formal proof techniques. The increasing reliance on software and increasing complexity of today's systems has made this task more difficult. Technologies and practices that might have been perfectly acceptable ten or fifteen years ago are insufficient today. Thankfully, the state of the art in analysis and proof tools and techniques has also advanced.

The latest revisions of the Ada language, Ada 2005 and Ada 2012, make enhanced software integrity possible. From its inception in the 1980s, Ada was designed to meet the requirements of high-integrity systems, and continues to be well-suited for the implementation of critical embedded or native applications. And it has been receiving increased attention recently. Every language revision has enhanced expressiveness in many areas. Ada 2012, in particular, has introduced new features for contract-based programming that are valuable to any project where verification is part of the engineering lifecycle. Along with these language enhancements, Ada compiler and tool technology has also kept pace with general computing developments over the past few years. Ada development environments are available on a wide range of platforms and are being used for the most demanding applications.

It is no secret that we at AdaCore are very enthusiastic about Ada, but we will not claim that Ada is always the solution; Ada is no more a silver bullet than any other language. In some domains other languages make sense because of the availability of particular libraries or development frameworks. For example, C++ and Java are considered good choices for desktop programs or applications where a shortened time to market is a major objective. Other areas, such as website programming or system administration, tend to rely on different formalisms such as scripting and interpreted languages. The key is to select the proper technical approach, in terms of the language and tools, to meet the requirements. Ada's strength is in areas where reliability is paramount.

Learning a new language shouldn't be complicated. Programming paradigms have not evolved much since object oriented programming gained a foothold, and the same paradigms are present one way or another in many widely used languages. This document will thus give you an overview of the Ada language using analogies to C++ and Java—these are the languages you're already likely to know. No prior knowledge of Ada is assumed. If you are working on an Ada project now and need more background, if you are interested in learning to program in Ada, or if you need to perform an assessment of possible languages to be used for a new development, this guide is for you.

This document was prepared by Quentin Ochem, with contributions and review from Richard Kenner, Albert Lee, and Ben Brosgol.

CHAPTER

TWO

BASICS

Ada implements the vast majority of programming concepts that you're accustomed to in C++ and Java: classes, inheritance, templates (generics), etc. Its syntax might seem peculiar, though. It's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a simpler language—its syntax favors making it easier to conceptualize and read program code, rather than making it faster to write in a cleverly condensed manner. For example, full words like **begin** and **end** are used in place of curly braces. Conditions are written using **if**, **then**, **elsif**, **else**, and **end if**. Ada's assignment operator does not double as an expression, smoothly eliminating any frustration that could be caused by **=** being used where **==** should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens −− mark the start of a comment that continues to the end of the line. This is exactly the same as using **//** for comments in C++ and Java. There is no equivalent of **/*** ... ***/** block comments in Ada; use multiple −− lines instead.

Ada compilers are stricter with type and range checking than most C++ and Java programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding more creatively, but this helps detect problems and vulnerabilities at compile time—early on in the development cycle. In addition, dynamic checks (such as array bounds checks) provide verification that could not be done at compile time. Dynamic checks are performed at run time, similar to what is done in Java.

Ada identifiers and reserved words are case insensitive. The identifiers *VAR*, *var* and *VaR* are treated as the same; likewise **begin**, **BEGIN**, **Begin**, etc. Language-specific characters, such as accents, Greek or Russian letters, and Asian alphabets, are acceptable to use. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

abort	else	null	select
abs	elsif	of	separate
abstract	end	or	some
accept	entry	others	subtype
access	exception	out	synchronized
aliased	exit	overriding	tagged
all	for	package	task
and	function	pragma	terminate
array	generic	private	then
at	goto	procedure	type
begin	if	protected	until
body	in	raise	use
case	interface	range	when
constant	is	record	while
declare	limited	rem	with
delay	loop	renames	xor
delta	mod	requeue	
digits	new	return	
do	not	reverse	

Ada is designed to be portable. Ada compilers must follow a precisely defined international (ISO) standard language specification with clearly documented areas of vendor freedom where the behavior depends on the implementation. It's possible, then, to write an implementation-independent application in Ada and to make sure it will have the same effect across platforms and compilers.

Ada is truly a general purpose, multiple paradigm language that allows the programmer to employ or avoid features like run-time contract checking, tasking, object oriented programming, and generics. Efficiently programmed Ada is employed in device drivers, interrupt handlers, and other low-level functions. It may be found today in devices with tight limits on processing speed, memory, and power consumption. But the language is also used for programming larger interconnected systems running on workstations, servers, and supercomputers.

CHAPTER

THREE

COMPILATION UNIT STRUCTURE

C++ programming style usually promotes the use of two distinct files: header files used to define specifications (*.h*, *.hxx*, *.hpp*), and implementation files which contain the executable code (*.c*, *.cxx*, *.cpp*). However, the distinction between specification and implementation is not enforced by the compiler and may need to be worked around in order to implement, for example, inlining or templates.

Java compilers expect both the implementation and specification to be in the same *.java* file. (Yes, design patterns allow using interfaces to separate specification from implementation to a certain extent, but this is outside of the scope of this description.)

Ada is superficially similar to the C++ case: Ada compilation units are generally split into two parts, the specification and the body. However, what goes into those files is more predictable for both the compiler and for the programmer. With GNAT, compilation units are stored in files with a *.ads* extension for specifications and with a *.adb* extension for implementations.

Without further ado, we present the famous "Hello World" in three languages:

[Ada]

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

procedure Main is
begin
   Put_Line ("Hello World");
end Main;
```

[C++]

```
#include <iostream>
using namespace std;

int main(int argc, const char* argv[]) {
   cout << "Hello World" << endl;
}
```

[Java]

```
public class Main {
   public static void main(String [] argv) {
      System.out.println ("Hello World");
   }
}
```

The first line of Ada we see is the **with** clause, declaring that the unit (in this case, the Main subprogram) will require the services of the package *Ada.Text_IO*. This is different from how **#include** works in C++ in that it does not, in a

logical sense, copy/paste the code of *Ada.Text_IO* into *Main*. The **with** clause directs the compiler to make the public interface of the *Ada.Text_IO* package visible to code in the unit (here *Main*) containing the **with** clause. Note that this construct does not have a direct analog in Java, where the entire CLASSPATH is always accessible. Also, the name ''Main" for the main subprogram was chosen for consistency with C++ and Java style but in Ada the name can be whatever the programmer chooses.

The **use** clause is the equivalent of **using namespace** in C++, or **import** in Java (though it wasn't necessary to use **import** in the Java example above). It allows you to omit the full package name when referring to **with**ed units. Without the **use** clause, any reference to *Ada.Text_IO* items would have had to be fully qualified with the package name. The *Put_Line* line would then have read:

```
Ada.Text_IO.Put_Line ("Hello World");
```

The word "package" has different meanings in Ada and Java. In Java, a package is used as a namespace for classes. In Ada, it's often a compilation unit. As a result Ada tends to have many more packages than Java. Ada package specifications ("package specs" for short) have the following structure:

```
package Package_Name is

   -- public declarations

private

   -- private declarations

end Package_Name;
```

The implementation in a package body (written in a *.adb* file) has the structure:

```
package body Package_Name is

   -- implementation

end Package_Name;
```

The **private** reserved word is used to mark the start of the private portion of a package spec. By splitting the package spec into private and public parts, it is possible to make an entity available for use while hiding its implementation. For instance, a common use is declaring a **record** (Ada's **struct**) whose fields are only visible to its package and not to the caller. This allows the caller to refer to objects of that type, but not to change any of its contents directly.

The package body contains implementation code, and is only accessible to outside code through declarations in the package spec.

An entity declared in the private part of a package in Ada is roughly equivalent to a protected member of a C++ or Java class. An entity declared in the body of an Ada package is roughly equivalent to a private member of a C++ or Java class.

CHAPTER

FOUR

STATEMENTS, DECLARATIONS, AND CONTROL STRUCTURES

4.1 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[Ada]

```
--
--  Ada program to declare and modify Integers
--
procedure Main is
   --  Variable declarations
   A, B : Integer := 0;
   C    : Integer := 100;
   D    : Integer;
begin
   --  Ada uses a regular assignment statement for incrementation.
   A := A + 1;

   --  Regular addition
   D := A + B + C;
end Main;
```

[C++]

```
/*
 *  C++ program to declare and modify ints
 */
int main(int argc, const char* argv[]) {
   //  Variable declarations
   int a = 0, b = 0, c = 100, d;

   //  C++ shorthand for incrementation
   a++;

   //  Regular addition
   d = a + b + c;
}
```

[Java]

```
/*
 *  Java program to declare and modify ints
 */
public class Main {
   public static void main(String [] argv) {
      //  Variable declarations
      int a = 0, b = 0, c = 100, d;

      //  Java shorthand for incrementation
      a++;

      //  Regular addition
      d = a + b + c;
   }
}
```

Statements are terminated by semicolons in all three languages. In Ada, blocks of code are surrounded by the reserved words **begin** and **end** rather than by curly braces. We can use both multi-line and single-line comment styles in the C++ and Java code, and only single-line comments in the Ada code.

Ada requires variable declarations to be made in a specific area called the *declarative part*, seen here before the **begin** keyword. Variable declarations start with the identifier in Ada, as opposed to starting with the type as in C++ and Java (also note Ada's use of the **:** separator). Specifying initializers is different as well: in Ada an initialization expression can apply to multiple variables (but will be evaluated separately for each), whereas in C++ and Java each variable is initialized individually. In all three languages, if you use a function as an initializer and that function returns different values on every invocation, each variable will get initialized to a different value.

Let's move on to the imperative statements. Ada does not provide **++** or −− shorthand expressions for increment/decrement operations; it is necessary to use a full assignment statement. The **:=** symbol is used in Ada to perform value assignment. Unlike C++'s and Java's **=** symbol, **:=** can not be used as part of an expression. So, a statement like *A* **:=** *B* **:=** *C;* doesn't make sense to an Ada compiler, and neither does a clause like "**if** *A* **:=** *B* **then** ...." Both are compile-time errors.

You can nest a block of code within an outer block if you want to create an inner scope:

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

procedure Main is
begin
   Put_Line ("Before the inner block");

   declare
      Alpha : Integer := 0;
   begin
      Alpha := Alpha + 1;
      Put_Line ("Now inside the inner block");
   end;

   Put_Line ("After the inner block");
end Main;
```

It is OK to have an empty declarative part or to omit the declarative part entirely—just start the inner block with **begin** if you have no declarations to make. However it is not OK to have an empty sequence of statements. You must at least provide a **null;** statement, which does nothing and indicates that the omission of statements is intentional.

4.2 Conditions

The use of the **if** statement:

[Ada]

```
if Variable > 0 then
   Put_Line (" > 0 ");
elsif Variable < 0 then
   Put_Line (" < 0 ");
else
   Put_Line (" = 0 ");
end if;
```

[C++]

```
if (Variable > 0)
   cout << " > 0 " << endl;
else if (Variable < 0)
   cout << " < 0 " << endl;
else
   cout << " = 0 " << endl;
```

[Java]

```
if (Variable > 0)
   System.out.println (" > 0 ");
else if (Variable < 0)
   System.out.println (" < 0 ");
else
   System.out.println (" = 0 ");
```

In Ada, everything that appears between the **if** and **then** keywords is the conditional expression—no parentheses required. Comparison operators are the same, except for equality (**=**) and inequality (**/=**). The english words **not**, **and**, and **or** replace the symbols **!**, **&**, and |, respectively, for performing boolean operations.

It's more customary to use **&&** and || in C++ and Java than **&** and | when writing boolean expressions. The difference is that **&&** and || are short-circuit operators, which evaluate terms only as necessary, and **&** and | will unconditionally evaluate all terms. In Ada, **and** and **or** will evaluate all terms; **and then** and **or else** direct the compiler to employ short circuit evaluation.

Here are what switch/case statements look like:

[Ada]

```
case Variable is
   when 0 =>
      Put_Line ("Zero");
   when 1 .. 9 =>
      Put_Line ("Positive Digit");
   when 10 | 12 | 14 | 16 | 18 =>
      Put_Line ("Even Number between 10 and 18");
   when others =>
      Put_Line ("Something else");
end case;
```

[C++]

```
switch (Variable) {
   case 0:
```

```
      cout << "Zero" << endl;
      break;
   case 1: case 2: case 3: case 4: case 5:
   case 6: case 7: case 8: case 9:
      cout << "Positive Digit" << endl;
      break;
   case 10: case 12: case 14: case 16: case 18:
      cout << "Even Number between 10 and 18" << endl;
      break;
   default:
      cout << "Something else";
}
```

[Java]

```
switch (Variable) {
   case 0:
      System.out.println ("Zero");
      break;
   case 1: case 2: case 3: case 4: case 5:
   case 6: case 7: case 8: case 9:
      System.out.println ("Positive Digit");
      break;
   case 10: case 12: case 14: case 16: case 18:
      System.out.println ("Even Number between 10 and 18");
      break;
   default:
      System.out.println ("Something else");
}
```

In Ada, the **case** and **end case** lines surround the whole case statement, and each case starts with **when**. So, when programming in Ada, replace **switch** with **case**, and replace **case** with **when**.

Case statements in Ada require the use of discrete types (integers or enumeration types), and require all possible cases to be covered by **when** statements. If not all the cases are handled, or if duplicate cases exist, the program will not compile. The default case, **default:** in C++ and Java, can be specified using **when others =>** in Ada.

In Ada, the **break** instruction is implicit and program execution will never fall through to subsequent cases. In order to combine cases, you can specify ranges using **..** and enumerate disjoint values using | which neatly replaces the multiple **case** statements seen in the C++ and Java versions.

4.3 Loops

In Ada, loops always start with the **loop** reserved word and end with **end loop**. To leave the loop, use **exit**—the C++ and Java equivalent being **break**. This statement can specify a terminating condition using the **exit when** syntax. The **loop** opening the block can be preceded by a **while** or a **for**.

The **while** loop is the simplest one, and is very similar across all three languages:

[Ada]

```
while Variable < 10_000 loop
   Variable := Variable * 2;
end loop;
```

[C++]

```
while (Variable < 10000) {
   Variable = Variable * 2;
}
```

[Java]

```
while (Variable < 10000) {
    Variable = Variable * 2;
}
```

Ada's **for** loop, however, is quite different from that in C++ and Java. It always increments or decrements a loop index within a discrete range. The loop index (or "loop parameter" in Ada parlance) is local to the scope of the loop and is implicitly incremented or decremented at each iteration of the loop statements; the program cannot directly modify its value. The type of the loop parameter is derived from the range. The range is always given in ascending order even if the loop iterates in descending order. If the starting bound is greater than the ending bound, the interval is considered to be empty and the loop contents will not be executed. To specify a loop iteration in decreasing order, use the **reverse** reserved word. Here are examples of loops going in both directions:

[Ada]

```
--  Outputs 0, 1, 2, ..., 9
for Variable in 0 .. 9 loop
   Put_Line (Integer'Image (Variable));
end loop;

--  Outputs 9, 8, 7, ..., 0
for Variable in reverse 0 .. 9 loop
   Put_Line (Integer'Image (Variable));
end loop;
```

[C++]

```
//  Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
   cout << Variable << endl;
}

//  Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >=0; Variable--) {
   cout << Variable << endl;
}
```

[Java]

```
//  Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
   System.out.println (Variable);
}

//  Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
   System.out.println (Variable);
}
```

Ada uses the *Integer* type's *'Image* attribute to convert a numerical value to a String. There is no implicit conversion between *Integer* and *String* as there is in C++ and Java. We'll have a more in-depth look at such attributes later on.

It's easy to express iteration over the contents of a container (for instance, an array, a list, or a map) in Ada and Java. For example, assuming that *Int_List* is defined as an array of Integer values, you can use:

[Ada]

```
for I of Int_List loop
   Put_Line (Integer'Image (I));
end loop;
```

[Java]

```
for (int i : Int_List) {
   System.out.println (i);
}
```

CHAPTER

FIVE

TYPE SYSTEM

5.1 Strong Typing

One of the main characteristics of Ada is its strong typing (i.e., relative absence of implicit type conversions). This may take some getting used to. For example, you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

```
procedure Strong_Typing is
   Alpha  : Integer := 1;
   Beta   : Integer := 10;
   Result : Float;
begin
   Result := Float (Alpha) / Float (Beta);
end Strong_Typing;
```

[C++]

```
void weakTyping (void) {
   int   alpha = 1;
   int   beta = 10;
   float result;

   result = alpha / beta;
}
```

[Java]

```
void weakTyping () {
   int   alpha = 1;
   int   beta = 10;
   float result;

   result = alpha / beta;
}
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from Integer to Float explicit. In fact it significantly changes the behavior of the computation. While the Ada code performs a floating point operation **1.0 / 10.0** and stores 0.1 in *Result*, the C++ and Java versions instead store 0.0 in *result*. This is because the C++ and Java versions perform an integer operation between two integer

variables: **1 / 10** is **0**. The result of the integer division is then converted to a *float* and stored. Errors of this sort can be very hard to locate in complex pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to *Float*:

```
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

In Ada, a floating point literal must be written with both an integral and decimal part. **10** is not a valid literal for a floating point value, while **10.0** is.

5.2 Language-Defined Types

The principal scalar types predefined by Ada are *Integer*, *Float*, *Boolean*, and *Character*. These correspond to **int**, **float**, **bool/boolean**, and **char**, respectively. The names for these types are not reserved words; they are regular identifiers.

5.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators "+", "-", "*", and "/" for integer and floating-point types take operands of the same type and return a value of that type.

```
procedure Main is
   type Distance is new Float;
   type Area is new Float;

   D1 : Distance := 2.0;
   D2 : Distance := 3.0;
   A  : Area;
begin
   D1 := D1 + D2;         -- OK
   D1 := D1 + A;          -- NOT OK: incompatible types for "+" operator
   A  := D1 * D2;         -- NOT OK: incompatible types for ":=" assignment
   A  := Area (D1 * D2); -- OK
end Main;
```

Even though the **Distance** and **Area** types above are just **Float**s, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two **Distance**s yields a **Distance**) and prohibit some useful cases (for example multiplying two **Distance**s should deliver an **Area**). These situations can be handled through other mechanisms. A predefined operation can be identified as **abstract** to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C++ and Java's *enum*s.

[Ada]

```
type Day is
  (Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday);
```

[C++]

```
enum Day {
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday};
```

[Java]

```
enum Day {
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday}
```

But even though such enumerations may be implemented using a machine word, at the language level Ada will not confuse the fact that *Monday* is a *Day* and is not an *Integer*. You can compare a *Day* with another *Day*, though. To specify implementation details like the numeric values that correspond with enumeration values in C++ you include them in the original *enum* statement:

[C++]

```
enum Day {
   Monday    = 10,
   Tuesday   = 11,
   Wednesday = 12,
   Thursday  = 13,
   Friday    = 14,
   Saturday  = 15,
   Sunday    = 16};
```

But in Ada you must use both a type definition for *Day* as well as a separate *representation clause* for it like:

[Ada]

```
for Day use
  (Monday    => 10,
   Tuesday   => 11,
   Wednesday => 12,
   Thursday  => 13,
   Friday    => 14,
```

```
   Saturday  => 15,
   Sunday    => 16);
```

5.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a *range constraint* introduced with the **range** reserved word, for example:

```
procedure Main is
   type Grade is range 0 .. 100;

   G1, G2  : Grade;
   N       : Integer;
begin
   ...                -- Initialization of N
   G1 := 80;          -- OK
   G1 := N;           -- Illegal (type mismatch)
   G1 := Grade (N);   -- Legal, run-time range check
   G2 := G1 + 10;     -- Legal, run-time range check
   G1 := (G1 + G2)/2; -- Legal, run-time range check
end Main;
```

In the above example, *Grade* is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to *G1* is correct and will not raise a run-time exceprion. Assigning *N* to *G1* is illegal since *Grade* is a different type than *Integer*. Converting *N* to *Grade* makes the assignment legal, and a range check on the conversion confirms that the value is within 0 .. 100. Assigning *G1+10* to *G2* is legal since **+** for *Grade* returns a *Grade* (note that the literal *10* is interpreted as a *Grade* value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression *G1 + G2* may be outside the range of *Grade*, but the final result will be in range. Nevertheless, depending on the representation chosen for *Grade*, the addition may overflow. If the compiler represents *Grade* values as signed 8-bit integers (i.e., machine numbers in the range -128 .. 127) then the sum *G1+G2* may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

```
G1 := Grade (Integer (G1) + Integer (G2)) / 2);
```

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the *Day* example, one can write:

```
type Business_Day is new Day range Monday .. Friday;
type Weekend_Day is new Day range Saturday .. Sunday;
```

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind 'subtypes' in Ada. A subtype is a type with optional additional constraints. For example:

```
subtype Business_Day is Day range Monday .. Friday;
subtype Weekend_Day is Day range Saturday .. Sunday;
subtype Dice_Throw is Integer range 1 .. 6;
```

These declarations don't create new types, just new names for constrained ranges of their base types.

5.5 Generalized Type Contracts: Subtype Predicates

Range checks are a special form of type contracts; a more general method is provided by Ada subtype predicates, introduced in Ada 2012. A subtype predicate is a boolean expression defining conditions that are required for a given type or subtype. For example, the *Dice_Throw* subtype shown above can be defined in the following way:

```
subtype Dice_Throw is Integer
   with Dynamic_Predicate => Dice_Throw in 1 .. 6;
```

The clause beginning with **with** introduces an Ada 'aspect', which is additional information provided for declared entities such as types and subtypes. The *Dynamic_Predicate* aspect is the most general form. Within the predicate expression, the name of the (sub)type refers to the current value of the (sub)type. The predicate is checked on assignment, parameter passing, and in several other contexts. There is a "Static_Predicate" form which introduce some optimization and constrains on the form of these predicates, outside of the scope of this document.

Of course, predicates are useful beyond just expressing ranges. They can be used to represent types with arbitrary constraints, in particular types with discontinuities, for example:

```
type Not_Null is new Integer
   with Dynamic_Predicate => Not_Null /= 0;

type Even is new Integer
   with Dynamic_Predicate => Even mod 2 = 0;
```

5.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The *'Image* and *'Value* attributes allow you to transform a scalar value into a *String* and vice-versa. For example:

```
declare
   A : Integer := 99;
begin
   Put_Line (Integer'Image (A));
   A := Integer'Value ("99");
end;
```

Certain attributes are provided only for certain kinds of types. For example, the *'Val* and *'Pos* attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

```
declare
   C : Character := 'a';
begin
   C := Character'Val (Character'Pos (C) + 1);
end;
```

A more concise way to get the next value in Ada is to use the *'Succ* attribute:

```
declare
   C : Character := 'a';
begin
   C := Character'Succ (C);
end;
```

You can get the previous value using the *'Pred* attribute. Here is the equivalent in C++ and Java:

[C++]

```
char c = 'a';
c++;
```

[Java]

```
char c = 'a';
c++;
```

Other interesting examples are the *'First* and *'Last* attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, *Integer'First* returns -2^{31} and *Integer'Last* returns $2^{31} - 1$.

5.7 Arrays and Strings

C++ arrays are pointers with offsets, but the same is not the case for Ada and Java. Arrays in the latter two languages are not interchangable with operations on pointers, and array types are considered first-class citizens. Arrays in Ada have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds—the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from 'a' to 'z':

[Ada]

```
declare
   type Arr_Type is array (Integer range <>) of Character;
   Arr : Arr_Type (1 .. 26);
   C : Character := 'a';
begin
   for I in Arr'Range loop
      Arr (I) := C;
      C := Character'Succ (C);
   end loop;
end;
```

[C++]

```
char Arr [26];
char C = 'a';

for (int I = 0; I < 26; ++I) {
   Arr [I] = C;
   C = C + 1;
}
```

[Java]

```
char [] Arr = new char [26];
char C = 'a';

for (int I = 0; I < Arr.length; ++I) {
   Arr [I] = C;
   C = C + 1;
}
```

In C++ and Java, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of Integer as the index type, but does not provide any constraints (use <>, pronounced 'box', to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called *'Range*. In our example, *Arr'Range* can also be expressed as *Arr'First .. Arr'Last*; both expressions will resolve to *1 .. 26*. So the *'Range* attribute supplies the bounds for our **for** loop. There is no risk of stating either of the bounds incorrectly, as one might do in C++ where "I <= 26" may be specified as the end-of-loop condition.

As in C++, Ada *String*s are arrays of *Character*s. The C++ or Java *String* class is the equivalent of the Ada type *Ada.Strings.Unbounded_String* which offers additional capabilities in exchange for some overhead. Ada strings, importantly, are not delimited with the special character '\0' like they are in C++. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined *String* type is very straighforward to use:

```
My_String : String (1 .. 26);
```

Unlike C++ and Java, Ada does not offer escape sequences such as '\n'. Instead, explicit values from the *ASCII* package must be concatenated (via the concatenation operator, &). Here for example, is how to initialize a line of text ending with a new line:

> My_String : String := "This is a line with a end of line" & ASCII.LF;

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of *My_String*'s bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C++ or Java, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada, the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

[Ada]

```
declare
   type Arr_Type is array (Integer range <>) of Integer
   A1 : Arr_Type (1 .. 2);
   A2 : Arr_Type (1 .. 2);
begin
   A1 (1) := 0;
   A1 (2) := 1;

   A2 := A1;
end;
```

[C++]

```
int A1 [2];
int A2 [2];

A1 [0] = 0;
A1 [1] = 1;
```

```
for (int i = 0; i < 2; ++i) {
   A2 [i] = A1 [i];
}
```

[Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];

A1 [0] = 0;
A1 [1] = 1;

A2 = Arrays.copyOf(A1, A1.length);
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

```
declare
   type Arr_Type is array (Integer range <>) of Integer
   A1 : Arr_Type (1 .. 10);
   A2 : Arr_Type (1 .. 5);
begin
   A2 (1 .. 3) := A1 (4 .. 6);
end;
```

This assigns the 4th, 5th, and 6th elements of *A1* into the 1st, 2nd, and 3rd elements of *A2*. Note that only the length matters here: the values of the indexes don't have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

[Ada]

```
declare
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type (1 .. 2);
   A2 : Arr_Type (1 .. 2);
begin
   if A1 = A2 then
```

[C++]

```
int A1 [2];
int A2 [2];

bool eq = true;

for (int i = 0; i < 2; ++i) {
   if (A1 [i] != A2 [i]) {
      eq = false;
   }
}

if (eq) {
```

[Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];
```

```
if (A1.equals (A2)) {
```

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the **others** expression to define a default value for all the unspecified array elements. Therefore, you can write:

```
declare
   type Arr_Type is array (Integer range <>) of Integer;
   A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
   A2 : Arr_Type (-2 .. 42) := (others => 0);
begin
   A1 := (1, 2, 3, others => 10);

   -- use a slice to assign A2 elements 11 .. 19 to 1
   A2 (11 .. 19) := (others => 1);
end;
```

5.8 Heterogeneous Data Structures

In Ada, there's no distinction between **struct** and **class** as there is in C++. All heterogeneous data structures are **record**s. Here are some simple records:

[Ada]

```
declare
   type R is record
      A, B : Integer;
      C    : Float;
   end record;

   V : R;
begin
   V.A := 0;
end;
```

[C++]

```
struct R {
   int A, B;
   float C;
};

R V;
V.A = 0;
```

[Java]

```
class R {
   public int A, B;
   public float C;
}

R V = new R ();
V.A = 0;
```

Ada allows specification of default values for fields just like C++ and Java. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by **others => <>** to specify that fields not listed will take their default values. For example:

```
type R is record
   A, B : Integer := 0;
   C    : Float := 0.0;
end record;

V1 : R => (1, 2, 1.0);
V2 : R => (A => 1, B => 2, C => 1.0);
V3 : R => (C => 1.0, A => 1, B => 2);
V3 : R => (C => 1.0, others => <>);
```

5.9 Pointers

Pointers, references, and access types differ in significant ways across the languages that we are examining. In C++, pointers are integral to a basic understanding of the language, from array manipulation to proper declaration and use of function parameters. Java goes a step further: everything is a reference, except for primitive types like scalars. Ada's design goes in the other direction: it makes more features available without requiring the explicit use of pointers.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

```
declare
   type R is record
      A, B : Integer;
   end record;

   V1, V2 : R;
begin
   V1.A := 0;
   V2 := V1;
   V2.A := 1;
end;
```

[C++]

```
struct R {
   int A, B;
};

R V1, V2;
V1.A = 0;
V2 = V1;
V2.A = 1;
```

[Java]

```
public class R {
   public int A, B;
}

R V1, V2;
V1 = new R ();
```

```
V1.A = 0;
V2 = V1;
V2.A = 1;
```

There's a fundamental difference between the Ada and C++ semantics above and the semantics for Java. In Ada and C++, objects are allocated on the stack and are directly accessed. *V1* and *V2* are two different objects and the assignment statement copies the value of *V1* into *V2*. In Java, *V1* and *V2* are two 'references' to objects of class *R*. Note that when *V1* and *V2* are declared, no actual object of class *R* yet exists in memory: it has to be allocated later with the **new** allocator operator. After the assignment *V2 = V1*, there's only one R object in memory: the assignment is a reference assignment, not a value assignment. At the end of the Java code, *V1* and *V2* are two references to the same objects and the *V2.A = 1* statement changes the field of that one object, while in the Ada and the C++ case *V1* and *V2* are two distinct objects.

To obtain similar behavior in Ada, you can use pointers. It can be done through Ada's 'access type':

[Ada]

```
declare
   type R is record
      A, B : Integer;
   end record;
   type R_Access is access R;

   V1 : R_Access;
   V2 : R_Access;
begin
   V1 := new R;
   V1.A := 0;
   V2 := V1;
   V2.A := 1;
end;
```

[C++]

```
struct R {
   int A, B;
};

R * V1, * V2;
V1 = new R ();
V1->A = 0;
V2 = V1;
V2->A = 0;
```

For those coming from the Java world: there's no garbage collector in Ada, so objects allocated by the **new** operator need to be expressly freed.

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append **.all**. The equivalent of *V1->A* in C++ can be written either as *V1.A* or *V1.all.A*.

Pointers to scalar objects in Ada and C++ look like:

[Ada]

```
procedure Main is
   type A_Int is access Integer;
   Var : A_Int := new Integer;
begin
```

```
   Var.all := 0;
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
  int * Var = new int;
  *Var = 0;
}
```

An initializer can be specified with the allocation by appending *'(value)*:

```
Var : A_Int := new Integer'(0);
```

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being **aliased**. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either **access all** (if the referenced object needs to be assigned to) or **access constant** (if the referenced object is a constant). The *'Access* attribute works like the C++ & operator to get a pointer to the object, but with a "scope accessibility" check to prevent references to objects that have gone out of scope. For example:

[Ada]

```
type A_Int is access all Integer;
Var : aliased Integer;
Ptr : A_Int := Var'Access;
```

[C++]

```
int Var;
int * Ptr = &Var;
```

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs—it's called *Ada.Unchecked_Deallocation*. To create your customized deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

```
with Ada.Unchecked_Deallocation;
procedure Main is
   type Integer_Access is access all Integer;
   procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);
   My_Pointer : Integer_Access := new Integer;
begin
   Free (My_Pointer);
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
  int * my_pointer = new int;
  delete my_pointer;
}
```

CHAPTER

SIX

FUNCTIONS AND PROCEDURES

6.1 General Form

Subroutines in C++ and Java are always expressed as functions (methods) which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term "subprogram" to refer to both functions and procedures.

Parameters can be passed in three distinct modes: **in**, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram. **out** is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller. **in out** is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant reference in C++). Ada also provides **access** parameters, in effect an explicit pass-by-reference indicator.

In Ada the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). (There are some exceptions to the "in general". For example, parameters of scalar types are always passed by copy, for all three modes.) C++ has the programmer specify how to pass the parameter, and Java forces primitive type parameters to be passed by copy and all other parameters to be passed by reference. For this reason, a 1:1 mapping between Ada and Java isn't obvious but here's an attempt to show these differences:

[Ada]

```
procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer);

function Func (Var : Integer) return Integer;

procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer)
is
begin
    Var2 := Func (Var1);
    Var3 := Var3 + 1;
end Proc;

function Func (Var : Integer) return Integer
is
begin
   return Var + 1;
end Func;
```

[C++]

```
void Proc
  (int Var1,
   int & Var2,
   int & Var3);

int Func (int Var);

void Proc
  (int Var1,
   int & Var2,
   int & Var3) {

   Var2 = Func (Var1);
   Var3 = Var3 + 1;
}

int Func (int Var) {
   return Var + 1;
}
```

[Java]

```
public class ProcData {
   public int Var2;
   public int Var3;

   public void Proc (int Var1) {
      Var2 = Func (Var1);
      Var3 = Var3 + 1;
   }
}

int Func (int Var) {
   return Var + 1;
}
```

The first two declarations for *Proc* and *Func* are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C++, a function that has not yet been seen cannot be used. Here, *Proc* can call *Func* because its specification has been declared. In Java, it's fine to have the declaration of the subprogram later .

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax, including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

6.2 Overloading

Different subprograms may share the same name; this is called "overloading." As long as the subprogram signatures (subprogram name, parameter types, and return types) are different, the compiler will be able to resolve the calls to the proper destinations. For example:

```
function Value (Str : String) return Integer;
function Value (Str : String) return Float;
```

```
V : Integer := Value ("8");
```

The Ada compiler knows that an assignment to *V* requires an *Integer*. So, it chooses the *Value* function that returns an *Integer* to satisfy this requirement.

Operators in Ada can be treated as functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To express an operator as a function, enclose it in quotes:

[Ada]

```
function "=" (Left : Day; Right : Integer) return Boolean;
```

[C++]

```
bool operator = (Day Left, int Right);
```

6.3 Subprogram Contracts

You can express the expected inputs and outputs of subprograms by specifying subprogram contracts. The compiler can then check for valid conditions to exist when a subprogram is called and can check that the return value makes sense. Ada allows defining contracts in the form of *Pre* and *Post* conditions; this facility was introduced in Ada 2012. They look like:

```
function Divide (Left, Right : Float) return Float
   with Pre  => Right /= 0.0,
        Post => Divide'Result * Right < Left + 0.0001
                and then Divide'Result * Right > Left - 0.0001;
```

The above example adds a *Pre* condition, stating that *Right* cannot be equal to 0.0. While the IEEE floating point standard permits divide-by-zero, you may have determined that use of the result could still lead to issues in a particular application. Writing a contract helps to detect this as early as possible. This declaration also provides a *Post* condition on the result.

Postconditions can also be expressed relative to the value of the input:

```
procedure Increment (V : in out Integer)
   with Pre  => V < Integer'Last,
        Post => V = V'Old + 1;
```

V'Old in the postcondition represents the value that *V* had before entering *Increment*.

CHAPTER

SEVEN

PACKAGES

7.1 Declaration Protection

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C++. An Ada package contains three parts that, for GNAT, are separated into two files: *.ads* files contain public and private Ada specifications, and *.adb* files contain the implementation, or Ada bodies.

Java doesn't provide any means to cleanly separate the specification of methods from their implementation: they all appear in the same file. You can use interfaces to emulate having separate specifications, but this requires the use of OOP techniques which is not always practical.

Ada and C++ do offer separation between specifications and implementations out of the box, independent of OOP.

```
package Package_Name is
   -- public specifications
private
   -- private specifications
end Package_Name;

package body Package_Name is
   -- implementation
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. The **private** keyword splits the package spec into "public" and "private" parts. That is somewhat analogous to C++'s partitioning of the class construct into different sections with different visibility properties. In Java, the encapsulation has to be done field by field, but in Ada the entire definition of a type can be hidden. For example:

```
package Types is
   type Type_1 is private;
   type Type_2 is private;
   type Type_3 is private;
   procedure P (X : Type_1);
   ...
private
   procedure Q (Y : Type_1);
   type Type_1 is new Integer range 1 .. 1000;
   type Type_2 is array (Integer range 1 .. 1000) of Integer;
   type Type_3 is record
      A, B : Integer;
   end record;
end Types;
```

Subprograms declared above the **private** separator (such as *P*) will be visible to the package user, and the ones below (such as *Q*) will not. The body of the package, the implementation, has access to both parts.

7.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

```
-- root-child.ads

package Root.Child is
   --  package spec goes here
end Root.Child;

-- root-child.adb

package body Root.Child is
   --  package body goes here
end Root.Child;
```

Here, *Root.Child* is a child package of *Root*. The public part of *Root.Child* has access to the public part of *Root*. The private part of *Child* has access to the private part of *Root*, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

7.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a **with** clause that references the package, which is similar to the C++ **#include** directive. Visibility is implicit in Java: you can always access all classes located in your *CLASSPATH*. After a **with** clause, entities needs to be prefixed by the name of their package, like a C++ namespace or a Java package. This prefix can be omitted if a **use** clause is employed, similar to a C++ **using namespace** or a Java **import**.

[Ada]

```
-- pck.ads

package Pck is
   My_Glob : Integer;
end Pck;

-- main.adb

with Pck;

procedure Main is
begin
   Pck.My_Glob := 0;
end Main;
```

[C++]

```
// pck.h

namespace pck {
   extern int myGlob;
```

```
}

// pck.cpp

namespace pck {
   int myGlob;
}

// main.cpp

#include "pck.h"

int main (int argc, char ** argv) {
   pck::myGlob = 0;
}
```

[Java]

```
// Globals.java

package pck;

public class Globals {
   public static int myGlob;
}

// Main.java

public class Main {
   public static void main (String [] argv) {
      pck.Globals.myGlob = 0;
   }
}
```

CHAPTER

EIGHT

CLASSES AND OBJECT ORIENTED PROGRAMMING

8.1 Primitive Subprograms

Primitive subprograms in Ada are basically the subprograms that are eligible for inheritance / derivation. They are the equivalent of C++ member functions and Java instance methods. While in C++ and Java these subprograms are located within the nested scope of the type, in Ada they are simply declared in the same scope as the type. There's no syntactic indication that a subprogram is a primitive of a type.

The way to determine whether *P* is a primitive of a type *T* is if (1) it is declared in the same scope as *T*, and (2) it contains at least one parameter of type *T*, or returns a result of type *T*.

In C++ or Java, the self reference **this** is implicitly declared. It may need to be explicitly stated in certain situations, but usually it's omitted. In Ada the self-reference, called the 'controlling parameter', must be explicitly specified in the subprogram parameter list. While it can be any parameter in the profile with any name, we'll focus on the typical case where the first parameter is used as the 'self' parameter. Having the controlling parameter listed first also enables the use of OOP prefix notation which is convenient.

A **class** in C++ or Java corresponds to a **tagged type** in Ada. Here's an example of the declaration of an Ada tagged type with two parameters and some dispatching and non-dispatching primitives, with equivalent examples in C++ and Java:

[Ada]

```
type T is tagged record
   V, W : Integer;
end record;

type T_Access is access all T;

function F (V : T) return Integer;

procedure P1 (V : access T);

procedure P2 (V : T_Access);
```

[C++]

```
class T {
   public:
      int V, W;

      int F (void);
```

```
      void P1 (void);
};

void P2 (T * v);
```

[Java]

```
public class T {
      public int V, W;

      public int F (void) {};

      public void P1 (void) {};

      public static void P2 (T v) {};
}
```

Note that *P2* is not a primitive of *T*—it does not have any parameters of type *T*. Its parameter is of type *T_Access*, which is a different type.

Once declared, primitives can be called like any subprogram with every necessary parameter specified, or called using prefix notation. For example:

[Ada]

```
declare
   V : T;
begin
   V.P1;
end;
```

[C++]

```
{
  T v;
  v.P1 ();
}
```

[Java]

```
{
  T v = new T ();
  v.P1 ();
}
```

8.2 Derivation and Dynamic Dispatch

Despite the syntactic differences, derivation in Ada is similar to derivation (inheritance) in C++ or Java. For example, here is a type hierarchy where a child class overrides a method and adds a new method:

[Ada]

```
type Root is tagged record
   F1 : Integer;
end record;

procedure Method_1 (Self : Root);
```

```
type Child is new Root with record
   F2 : Integer;
end Child;

overriding
procedure Method_1 (Self : Child);

procedure Method_2 (Self : Child);
```

[C++]

```
class Root {
   public:
      int f1;
      virtual void method1 ();
};

class Child : public Root {
   public:
      int f2;
      virtual void method1 ();
      virtual void method2 ();
};
```

[Java]

```
public class Root {
   public int f1;
   public void method1 ();
}

public class Child extends Root {
   public int f2;
   @Override
   public void method1 ();
   public void method2 ();
}
```

Like Java, Ada primitives on tagged types are always subject to dispatching; there is no need to mark them **virtual**. Also like Java, there's an optional keyword **overriding** to ensure that a method is indeed overriding something from the parent type.

Unlike many other OOP languages, Ada differentiates between a reference to a specific tagged type, and a reference to an entire tagged type hierarchy. While *Root* is used to mean a specific type, *Root'Class*—a class-wide type—refers to either that type or any of its descendants. A method using a parameter of such a type cannot be overridden, and must be passed a parameter whose type is of any of *Root*'s descendants (including *Root* itself).

Next, we'll take a look at how each language finds the appropriate method to call within an OO class hierarchy; that is, their dispatching rules. In Java, calls to non-private instance methods are always dispatching. The only case where static selection of an instance method is possible is when calling from a method to the **super** version.

In C++, by default, calls to virtual methods are always dispatching. One common mistake is to use a by-copy parameter hoping that dispatching will reach the real object. For example:

```
void proc (Root p) {
   p.method1 ();
}

Root * v = new Child ();
```

```
proc (*v);
```

In the above code, *p.method1 ()* will not dispatch. The call to *proc* makes a copy of the *Root* part of *v*, so inside *proc*, *p.method1*() refers to the *method1*() of the root object. The intended behavior may be specified by using a reference instead of a copy:

```
void proc (Root & p) {
   p.method1 ();
}

Root * v = new Child ();

proc (*v);
```

In Ada, tagged types are always passed by reference but dispatching only occurs on class-wide types. The following Ada code is equivalent to the latter C++ example:

```
declare
   procedure Proc (P : Root'Class) is
   begin
      P.Method_1;
   end;

   type Root_Access is access all Root'Class;
   V : Root_Access := new Child;
begin
   Proc (V.all);
end;
```

Dispatching from within primitives can get tricky. Let's consider a call to *Method_1* in the implementation of *Method_2*. The first implementation that might come to mind is:

```
procedure Method_2 (P : Root) is
begin
   P.Method_1;
end;
```

However, *Method_2* is called with a parameter that is of the definite type *Root*. More precisely, it is a definite view of a child. So, this call is not dispatching; it will always call *Method_1* of *Root* even if the object passed is a child of *Root*. To fix this, a view conversion is necessary:

```
procedure Method_2 (P : Root) is
begin
   Root'Class (P).Method_1;
end;
```

This is called "redispatching." Be careful, because this is the most common mistake made in Ada when using OOP. In addition, it's possible to convert from a class wide view to a definite view, and to select a given primitive, like in C++:

[Ada]

```
procedure Proc (P : Root'Class) is
begin
   Root (P).Method_1;
end;
```

[C++]

```
void proc (Root & p) {
   p.Root::method1 ();
}
```

8.3 Constructors and Destructors

Ada does not have constructors and destructors in quite the same way as C++ and Java, but there is analagous functionality in Ada in the form of default initialization and finalization.

Default initialization may be specified for a record component and will occur if a variable of the record type is not assigned a value at initialization. For example:

```
type T is tagged record
   F : Integer := Compute_Default_F;
end record;

function Compute_Default_F return Integer is
begin
   Put_Line ("Compute");
   return 0;
end Compute_Default_F;

V1 : T;
V2 : T := (F => 0);
```

In the declaration of *V1*, *T.F* receives a value computed by the subprogram *Compute_Default_F*. This is part of the default initialization. *V2* is initialized manually and thus will not use the default initialization.

For additional expressive power, Ada provides a type called *Ada.Finalization.Controlled* from which you can derive your own type. Then, by overriding the *Initialize* procedure you can create a constructor for the type:

```
type T is new Ada.Finalization.Controlled with record
   F : Integer;
end record;

procedure Initialize (Self : in out T) is
begin
   Put_Line ("Compute");
   Self.F := 0;
end Initialize;

V1 : T;
V2 : T := (V => 0);
```

Again, this default initialization subprogram is only called for *V1*; *V2* is initialized manually. Furthermore, unlike a C++ or Java constructor, *Initialize* is a normal subprogram and does not perform any additional initialization such as calling the parent's initialization routines.

When deriving from *Controlled*, it's also possible to override the subprogram *Finalize*, which is like a destructor and is called for object finalization. Like *Initialize*, this is a regular subprogram. Do not expect any other finalizers to be automatically invoked for you.

Controlled types also provide functionality that essentially allows overriding the meaning of the assignment operation, and are useful for defining types that manage their own storage reclamation (for example, implementing a reference count reclamation strategy).

8.4 Encapsulation

While done at the class level for C++ and Java, Ada encapsulation occurs at the package level and targets all entities of the language, as opposed to only methods and attributes. For example:

[Ada]

```
package Pck is
   type T is tagged private;
   procedure Method1 (V : T);
private
   type T is tagged record
      F1, F2 : Integer;
   end record;
   procedure Method2 (V : T);
end Pck;
```

[C++]

```
class T {
   public:
      virtual void method1 ();
   protected:
      int f1, f2;
      virtual void method2 ();
};
```

[Java]

```
public class T {
   public void method1 ();
   protected int f1, f2;
   protected void method2 ();
}
```

The C++ and Java code's use of **protected** and the Ada code's use of **private** here demonstrates how to map these concepts between languages. Indeed, the private part of an Ada child package would have visibility of the private part of its parents, mimicking the notion of **protected**. Only entities declared in the package body are completely isolated from access.

8.5 Abstract Types and Interfaces

Ada, C++ and Java all offer similar functionality in terms of abstract classes, or pure virtual classes. It is necessary in Ada and Java to explicitly specify whether a tagged type or class is **abstract**, whereas in C++ the presence of a pure virtual function implicitly makes the class an abstract base class. For example:

[Ada]

```
package P is

    type T is abstract tagged private;

    procedure Method (Self : T) is abstract;
 private
    type T is abstract tagged record
       F1, F2 : Integer;
    end record;
```

```
 end P;
```

[C++]

```
class T {
   public:
      virtual void method () = 0;
   protected:
      int f1, f2;
};
```

[Java]

```
public abstract class T {
   public abstract void method1 ();
   protected int f1, f2;
};
```

All abstract methods must be implemented when implementing a concrete type based on an abstract type.

Ada doesn't offer multiple inheritance the way C++ does, but it does support a Java-like notion of interfaces. An interface is like a C++ pure virtual class with no attributes and only abstract members. While an Ada tagged type can inherit from at most one tagged type, it may implement multiple interfaces. For example:

[Ada]

```
type Root is tagged record
   F1 : Integer;
end record;
procedure M1 (Self : Root);

type I1 is interface;
procedure M2 (Self : I1) is abstract;

type I2 is interface;
procedure M3 (Self : I2) is abstract;

type Child is new Root and I1 and I2 with record
   F2 : Integer;
end record;

-- M1 implicitly inherited by Child
procedure M2 (Self : Child);
procedure M3 (Self : Child);
```

[C++]

```
class Root {
   public:
      virtual void M1();
      int f1;
};

class I1 {
   public:
      virtual void M2 () = 0;
};

class I2 {
   public:
```

```
      virtual void M3 () = 0;
};

class Child : public Root, I1, I2 {
   public:
      int f2;
      virtual void M2 ();
      virtual void M3 ();
};
```

[Java]

```
public class Root {
   public void M1();
   public int f1;
}

public interface I1 {
   public void M2 () = 0;
}

public class I2 {
   public void M3 () = 0;
}

public class Child extends Root implements I1, I2 {
      public int f2;
      public void M2 ();
      public void M3 ();
}
```

8.6 Invariants

Any private type in Ada may be associated with a *Type_Invariant* contract. An invariant is a property of a type that must always be true after the return from of any of its primitive subprograms. (The invariant might not be maintained during the execution of the primitive subprograms, but will be true after the return.) Let's take the following example:

```
package Int_List_Pkg is

   type Int_List (Max_Length : Natural) is private
     with Type_Invariant => Is_Sorted (Int_List);

   function Is_Sorted (List : Int_List) return Boolean;

   type Int_Array is array (Positive range <>) of Integer;

   function To_Int_List (Ints : Int_Array) return Int_List;

   function To_Int_Array (List : Int_List) return Int_Array;

   function "&" (Left, Right : Int_List) return Int_List;

   ... -- Other subprograms
private

   type Int_List (Max_Length : Natural) is record
```

```
      Length : Natural;
      Data   : Int_Array (1..Max_Length);
   end record;


   function Is_Sorted (List : Int_List) return Boolean is
      (for all I in List.Data'First .. List.Length-1 =>
            List.Data (I) <= List.Data (I+1));

end Int_List_Pkg;

package body Int_List_Pkg is

   procedure Sort (Ints : in out Int_Array) is
   begin
      ... Your favorite sorting algorithm
   end Sort;

   function To_Int_List (Ints : Int_Array) return Int_List is
      List : Int_List :=
       (Max_Length => Ints'Length,
        Length     => Ints'Length,
        Data       => Ints);
   begin
      Sort (List.Data);
      return List;
   end To_Int_List;

   function To_Int_Array (List : Int_List) return Int_Array is
   begin
      return List.Data;
   end To_Int_Array;

   function "&" (Left, Right : Int_List) return Int_List is
      Ints : Int_Array := Left.Data & Right.Data;
   begin
      Sort (Ints);
      return To_Int_List (Ints);
   end "&";

   ... -- Other subprograms
end Int_List_Pkg;
```

The *Is_Sorted* function checks that the type stays consistent. It will be called at the exit of every primitive above. It is permissible if the conditions of the invariant aren't met during execution of the primitive. In *To_Int_List* for example, if the source array is not in sorted order, the invariant will not be satisfied at the "begin", but it will be checked at the end.

CHAPTER

NINE

GENERICS

Ada, C++, and Java all have support for generics or templates, but on different sets of language entities. A C++ template can be applied to a class or a function. So can a Java generic. An Ada generic can be either a package or a subprogram.

9.1 Generic Subprograms

A feature that is similar across all three languages is the subprogram. To swap two objects:

[Ada]

```
generic
   type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
   Temp : A_Type := Left;
begin
   Left  := Right;
   Right := Temp;
end Swap;
```

[C++]

```
template <class AType>
AType swap (AType & left, AType & right) {
   AType temp = left;
   left  = right;
   right = temp;
}
```

[Java]

```
public <AType> void swap (AType left, AType right) {
   AType temp = left;
   left  = right;
   right = temp;
}
```

And examples of using these:

[Ada]

```
declare
   type R is record
      F1, F2 : Integer;
```

```
   end record;

   procedure Swap_R is new Swap (R);
   A, B : R;
begin
   ...
   Swap_R (A, B);
end;
```

[C++]

```
class R {
   public:
      int f1, f2;
};

R a, b;
...
swap (a, b);
```

[Java]

```
public class R {
   public int f1, f2;
}

R a = new R(), b = new R();
...
swap (a, b);
```

The C++ template and Java generic both become usable once defined. The Ada generic needs to be explicitly instantiated using a local name and the generic's parameters.

9.2 Generic Packages

Next, we're going to create a generic unit containing data and subprograms. In Java or C++, this is done through a class, while in Ada, it's a 'generic package'. The Ada and C++ model is fundamentally different from the Java model. Indeed, upon instantiation, Ada and C++ generic data are duplicated; that is, if they contain global variables (Ada) or static attributes (C++), each instance will have its own copy of the variable, properly typed and independent from the others. In Java, generics are only a mechanism to have the compiler do consistency checks, but all instances are actually sharing the same data where the generic parameters are replaced by *java.lang.Object*. Let's look at the following example:

[Ada]

```
generic
   type T is private;
package Gen is
   type C is tagged record
      V : T;
   end record;

   G : Integer;
end Gen;
```

[C++]

```
template <class T>
class C{
   public:
     T v;
     static int G;
};
```

[Java]

```
public class C <T> {
     public T v;
     public static int G;
}
```

In all three cases, there's an instance variable (*v*) and a static variable (*G*). Let's now look at the behavior (and syntax) of these three instantiations:

[Ada]

```
declare
   package I1 is new Gen (Integer);
   package I2 is new Gen (Integer);
   subtype Str10 is String (1..10);
   package I3 is new Gen (Str10);
begin
   I1.G := 0;
   I2.G := 1;
   I3.G := 2;
end;
```

[C++]

```
C <int>::G = 0;
C <int>::G = 1;
C <char *>::G = 2;
```

[Java]

```
C.G = 0;
C.G = 1;
C.G = 2;
```

In the Java case, we access the generic entity directly without using a parametric type. This is because there's really only one instance of *C*, with each instance sharing the same global variable *G*. In C++, the instances are implicit, so it's not possible to create two different instances with the same parameters. The first two assignments are manipulating the same global while the third one is manipulating a different instance. In the Ada case, the three instances are explicitly created, named, and referenced individually.

9.3 Generic Parameters

Ada offers a wide variety of generic parameters which is difficult to translate into other languages. The parameters used during instantiation—and as a consequence those on which the generic unit may rely on—may be variables, types, or subprograms with certain properties. For example, the following provides a sort algorithm for any kind of array:

```
generic
   type Component is private;
```

```
   type Index is (<>);
   with function "<" (Left, Right : Component) return Boolean;
   type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

The above declaration states that we need a type (*Component*), a discrete type (*Index*), a comparison subprogram ("<"), and an array definition (*Array_Type*). Given these, it's possible to write an algorithm that can sort any *Array_Type*. Note the usage of the **with** reserved word in front of the function name, to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```
type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type, such as String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<>); -- T is a discrete type
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type, T2 is its designated type
```

CHAPTER

TEN

EXCEPTIONS

Exceptions are a mechanism for dealing with run-time occurrences that are rare, that usually correspond to errors (such as improperly formed input data), and whose occurrence causes an unconditional transfer of control.

10.1 Standard Exceptions

Compared with Java and C++, the notion of an Ada exception is very simple. An exception in Ada is an object whose "type" is **exception**, as opposed to classes in Java or any type in C++. The only piece of user data that can be associated with an Ada exception is a String. Basically, an exception in Ada can be raised, and it can be handled; information associated with an occurrence of an exception can be interrogated by a handler.

Ada makes heavy use of exceptions especially for data consistency check failures at run time. These include, but are not limited to, checking against type ranges and array boundaries, null pointers, various kind of concurrency properties, and functions not returning a value. For example, the following piece of code will raise the exception *Constraint_Error*:

```
procedure P is
   V : Positive;
begin
   V := -1;
end P;
```

In the above code, we're trying to assign a negative value to a variable that's declared to be positive. The range check takes place during the assignment operation, and the failure raises the *Constraint_Error* exception at that point. (Note that the compiler may give a warning that the value is out of range, but the error is manifest as a run-time exception.) Since there is no local handler, the exception is propagated to the caller; if *P* is the main procedure, then the program will be terminated.

Java and C++ **throw** and **catch** exceptions when **try**ing code. All Ada code is already implicitly within **try** blocks, and exceptions are **raise**d and handled.

[Ada]

```
begin
   Some_Call;
exception
   when Exception_1 =>
      Put_Line ("Error 1");
   when Exception_2 =>
      Put_Line ("Error 2");
   when others =>
      Put_Line ("Unknown error");
end;
```

[C++]

```
try {
   someCall ();
} catch (Exception1) {
   cout << "Error 1" << endl;
} catch (Exception2) {
   cout << "Error 2" << endl;
} catch (...) {
   cout << "Unknown error" << endl;
}
```

[Java]

```
try {
   someCall ();
} catch (Exception1 e1) {
   System.out.println ("Error 1");
} catch (Exception2 e2) {
   System.out.println ("Error 2");
} catch (Throwable e3) {
   System.out.println ("Unknown error");
}
```

Raising and throwing exceptions while within an exception handler is permissible in all three languages.

10.2 Custom Exceptions

Custom exception declarations resemble object declarations, and they can be created in Ada using the **exception** keyword:

```
My_Exception : exception;
```

Your exceptions can then be raised using a **raise** statement, optionally accompanied by a message following the **with** reserved word:

[Ada]

```
raise My_Exception with "Some message";
```

[C++]

```
throw My_Exception ("Some message");
```

[Java]

```
throw new My_Exception ("Some message");
```

Language defined exceptions can also be raised in the same manner:

```
raise Constraint_Error;
```

CHAPTER

ELEVEN

CONCURRENCY

11.1 Tasks

Java and Ada both provide support for concurrency in the language. The C++ language has added a concurrency facility in its most recent revision, C++11, but we are assuming that most C++ programmers are not (yet) familiar with these new features. We thus provide the following mock API for C++ which is similar to the Java *Thread* class:

```
class Thread {
   public:
      virtual void run (); // code to execute
      void start (); // starts a thread and then call run ()
      void join (); // waits until the thread is finished
};
```

Each of the following examples will display the 26 letters of the alphabet twice, using two concurrent threads/tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

```
procedure Main is -- implicitly called by the environment task
   task My_Task;

   task body My_Task is
   begin
      for I in 'A' .. 'Z' loop
         Put_Line (I);
      end loop;
   end My_Task;
begin
   for I in 'A' .. 'Z' loop
      Put_Line (I);
   end loop;
end Main;
```

[C++]

```
class MyThread : public Thread {
   public:

   void run () {
      for (char i = 'A'; i <= 'Z'; ++i) {
         cout << i << endl;
      }
```

```
   }
};

int main (int argc, char ** argv) {
   MyThread myTask;
   myTask.start ();

   for (char i = 'A'; i <= 'Z'; ++i) {
      cout << i << endl;
   }

   myTask.join ();

   return 0;
}
```

[Java]

```
public class Main {
   static class MyThread extends Thread {
      public void run () {
         for (char i = 'A'; i <= 'Z'; ++i) {
            System.out.println (i);
         }
      }
   }

   public static void main (String args) {
      MyThread myTask = new MyThread ();
      myTask.start ();

      for (char i = 'A'; i <= 'Z'; ++i) {
         System.out.println (i);
      }
      myTask.join ();
   }
}
```

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the **begin**. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

An Ada task type is somewhat analogous to a Java *Thread* subclass, but in Java the instances of such a subclass are always dynamically allocated. In Ada an instance of a task type may either be declared or dynamically allocated.

Task types can be parametrized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the implementation's task scheduling algorithm.

[Ada]

```
task type My_Task (First : Character);

task body My_Task (First : Character) is
begin
```

```
   for I in First .. 'Z' loop
      Put_Line (I);
   end loop;
end My_Task;

procedure Main is
   Tab : array (0 .. 9) of My_Task ('G');
begin
   null;
end Main;
```

[C++]

```
class MyThread : public Thread {
   public:

   char first;

   void run () {
      for (char i = first; i <= 'Z'; ++i) {
         cout << i << endl;
      }
   }
};

int main (int argc, char ** argv) {
   MyThread tab [10];

   for (int i = 0; i < 9; ++i) {
      tab [i].first = 'G';
      tab [i].start ();
   }

   for (int i = 0; i < 9; ++i) {
      tab [i].join ();
   }

   return 0;
}
```

[Java]

```
public class MyThread extends Thread {
   public char first;

   public MyThread (char first){
      this.first = first;
   }

   public void run () {
      for (char i = first; i <= 'Z'; ++i) {
         cout << i << endl;
      }
   }
}

public class Main {
   public static void main (String args) {
      MyThread [] tab = new MyThread [10];
```

```
      for (int i = 0; i < 9; ++i) {
         tab [i] = new MyThread ('G');
         tab [i].start ();
      }

      for (int i = 0; i < 9; ++i) {
         tab [i].join ();
      }
   }
}
```

In Ada a task may be allocated on the heap as opposed to the stack. The task will then start as soon as it has been allocated, and terminates when its work is completed. This model is probably the one that's the most similar to Java:

[Ada]

```
type Ptr_Task is access My_Task;

procedure Main is
   T : Ptr_Task;
begin
   T := new My_Task ('G');
end Main;
```

[C++]

```
int main (int argc, char ** argv) {
   MyThread * t = new MyThread ();
   t->first = 'G';
   t->start ();
   return 0;
}
```

[Java]

```
public class Main {
   public static void main (String args) {
      MyThread t = new MyThread ('G');

      t.start ();
   }
}
```

11.2 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Ada's rendezvous facility cannot be modeled with C++ or Java without complex machinery. Therefore, this section will just show examples written in Ada.

Let's consider the following example:

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

procedure Main is

   task After is
      entry Go;
```

```
   end After ;

   task body After is
   begin
      accept Go;
      Put_Line ("After");
   end After;

begin
   Put_Line ("Before");
   After.Go;
end;
```

The *Go* **entry** declared in *After* is the external interface to the task. In the task body, the **accept** statement causes the task to wait for a call on the entry. This particular **entry** and **accept** pair doesn't do much more than cause the task to wait until *Main* calls *After.Go*. So, even though the two tasks start simultaneously and execute independently, they can coordinate via *Go*. Then, they both continue execution independently after the rendezvous.

The **entry/accept** pair can take/pass parameters, and the **accept** statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

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

procedure Main is

   task After is
      entry Go (Text : String);
   end After ;

   task body After is
   begin
      accept Go (Text : String) do
         Put_Line ("After: " & Text);
      end Go;
   end After;

begin
   Put_Line ("Before");
   After.Go ("Main");;
end;
```

In the above example, the *Put_Line* is placed in the **accept** statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the **begin** of *Main*, task *After* is started and the main procedure is suspended.
2. *After* reaches the **accept** statement and is suspended, since there is no pending call on the *Go* entry.
3. The main procedure is awakened and executes the *Put_Line* invocation, displaying the string "Before".
4. The main procedure calls the *Go* entry. Since *After* is suspended on its **accept** statement for this entry, the call succeeds.
5. Tha main procedure is suspended, and the task *After* is awakened to execute the body of the **accept** statement. The actual parameter "Main" is passed to the **accept** statement, and the *Put_Line* invocation is executed. As a result, the string "After: Main" is displayed.
6. When the **accept** statement is completed, both the *After* task and the main procedure are ready to run. Suppose that the *Main* procedure is given the processor. It reaches its **end**, but the local task *After* has not yet terminated.

The main procedure is suspended.

7. The *After* task continues, and terminates since it is at its **end**. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.

11.3 Selective Rendezvous

The accept statement by itself can only wait for a single event (call) at a time. The **select** statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call *Increment*, *Decrement*, and *Get*:

```
task Counter is
   entry Get (Result : out Integer);
   entry Increment;
   entry Decrement;
end Counter;

task body Counter is
   Value : Integer := 0;
begin
   loop
      select
         accept Increment do
            Value := Value + 1;
         end Increment;
      or
         accept Decrement do
            Value := Value - 1;
         end Decrement;
      or
         accept Get (Result : out Integer) do
            Result := Value;
         end Get;
      or
         delay 1.0 * Minute;
         exit;
      end select;
   end loop;
end Counter;
```

When the task's statement flow reaches the **select**, it will wait for all four events—three entries and a delay—in parallel. If the delay of one minute is exceeded, the task will execute the statements following the **delay** statement (and in this case will exit the loop, in effect terminating the task). The accept bodies for the *Increment*, *Decrement*, or *Get* entries will be otherwise executed as they're called. These four sections of the **select** statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a "race condition" could occur where multiple tasks simultaneously calling, say, *Increment*, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call *Increment* then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and *Counter* is not ready to accept the call (i.e., if *Counter* is not suspended at the **select** statement) then the calling task is suspended, and placed in the queue of the entry that it is calling. From the perspective of the *Counter* task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case *Counter* is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the one minute delay (whose effect was noted above).
- There is a call pending on exactly one of the entries. In this case control passes to the **select** branch with an **accept** statement for that entry. The choice of which caller to accept, if more than one, depends on the queuing policy, which can be specified via a pragma defined in the Real-Time Systems Annex of the Ada standard; the default is First-In First-Out.
- There are calls pending on more than one entry. In this case one of the entries with pending callers is chosen, and then one of the callers is chosen to be de-queued (the choices depend on the queueing policy).

11.4 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared data object, an alternative (and generally preferable) style is through a *protected object*, an efficiently implementable mechanism that makes the effect more explicit. A protected object has a public interface (its *protected operations*) for accessing and manipulating the object's components (its private part). Mutual exclusion is enforced through a conceptual lock on the object, and encapsulation ensures that the only external access to the components are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by procedures or entries, and read-only operations by functions. The lock mechanism is implemented so that it's possible to perform concurrent read operations but not concurrent write or read/write operations.

Let's reimplement our earlier tasking example with a protected object called *Counter*:

```
protected Counter is
   function Get return Integer;
   procedure Increment;
   procedure Decrement;
private
   Value : Integer := 0;
end Counter;

protected body Counter is
   function Get return Integer is
   begin
      return Value;
   end Get;

   procedure Increment is
   begin
     Value := Value + 1;
   end Increment;

   procedure Decrement is
   begin
      Value := Value - 1;
   end Decrement;
end Counter;
```

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

```
Counter.Increment;
Counter.Decrement;
Put_Line (Integer'Image (Counter.Get));
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```
declare
   protected type Counter is
      -- as above
   end Counter;

   protected body Counter is
      -- as above
   end Counter;

   C1 : Counter;
   C2 : Counter;
begin
   C1.Increment;
   C2.Decrement;
   ...
end;
```

Protected objects and types can declare a procedure-like operation known as an "entry". An entry is somewhat similar to a procedure but includes a so-called *barrier condition* that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

```
protected type Binary_Semaphore is
  entry Wait;
  procedure Signal;
private
  Signaled : Boolean := False;
end Binary_Semaphore;

protected body Binary_Semaphore is
  entry Wait when Signaled is
  begin
    Signaled := False;
  end Wait;

  procedure Signal is
  begin
    Signaled := True;
  end Signal;
end Binary_Semaphore;
```

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the *References* section.

CHAPTER

TWELVE

LOW LEVEL PROGRAMMING

12.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```
type R is record
   V  : Integer range 0 .. 255;
   B1 : Boolean;
   B2 : Boolean;
end record
with Pack;
```

[C++]

```
struct R {
   unsigned int v:8;
   bool b1;
   bool b2;
};
```

[Java]

```
public class R {
   public byte v;
   public boolean b1;
   public boolean b2;
}
```

The Ada and the C++ code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C++ and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C++, the number of bits required by each field needs to be specified. Here, we're stating that v is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The **Pack** aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

```
type R is record
   V  : Integer range 0 .. 255;
   B1 : Boolean;
   B2 : Boolean;
end record;

for R use record
   --  Occupy the first bit of the first byte.
   B1 at 0 range 0 .. 0;

   --  Occupy the last 7 bits of the first byte,
   --  as well as the first bit of the second byte.
   V  at 0 range 1 .. 8;

   --  Occupy the second bit of the second byte.
   B2 at 1 range 1 .. 1;
end record;
```

We omit the **with** *Pack* directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type *R* across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

12.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the *rdtsc* instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64 bit value split between the *edx* and *eax* registers.

GNAT provides a subprogram called *System.Machine_Code.Asm* that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

```
with System.Machine_Code; use System.Machine_Code;
with Interfaces;          use Interfaces;

function Get_Processor_Cycles return Unsigned_64 is
   Low, High : Unsigned_32;
   Counter   : Unsigned_64;
begin
   Asm ("rdtsc",
        Outputs =>
          (Unsigned_32'Asm_Output ("=a", High),
           Unsigned_32'Asm_Output ("=d", Low)),
        Volatile => True);

   Counter :=
     Unsigned_64 (High) * 2 ** 32 +
```

```
      Unsigned_64 (Low);

   return Counter;
end Get_Processor_Cycles;
```

The *Unsigned_32'Asm_Output* clauses above provide associations between machine registers and source-level variables to be updated. "=a" and "=d" refer to the *eax* and *edx* machine registers, respectively. The use of the *Unsigned_32* and *Unsigned_64* types from package *Interfaces* ensures correct representation of the data. We assemble the two 32-bit values to form a single 64 bit value.

We set the *Volatile* parameter to *True* to tell the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the *eax* and *edx* registers to implement the *High* and *Low* variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

12.3 Interfacing with C

Much effort was spent making Ada easy to interface with other languages. The *Interfaces* package hierarchy and the pragmas *Convention*, *Import*, and *Export* allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

```
struct my_struct {
   int A, B;
};

void call (my_struct * p) {
   printf ("%d", p->A);
}
```

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C **struct** *my_struct* is represented, we can use the following record along with a **pragma** *Convention*. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

```
type my_struct is record
   A : Interfaces.C.int;
   B : Interfaces.C.int;
end record;
pragma Convention (C, my_struct);
```

Describing a foreign subprogram call to Ada code is called "binding" and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a **void** function maps to an Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a **pragma** *Import*:

```
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The *Import* pragma specifies that whenever *Call* is invokeed by Ada code, it should invoke the *call* function with the C calling convention.

And that's all that's necessary. Here's an example of a call to *Call*:

```
declare
   V : my_struct := (A => 1, B => 2);
begin
   Call (V);
end;
```

You can also make Ada subprograms available to C code, and examples of this can be found in the GNAT User's Guide. Interfacing with C++ and Java use implementation-dependent features that are also available with GNAT.

CHAPTER

THIRTEEN

CONCLUSION

All the usual paradigms of imperative programming can be found in all three languages that we surveyed in this document. However, Ada is different from the rest in that it's more explicit when expressing properties and expectations. This is a good thing: being more formal affords better communication among programmers on a team and between programmers and machines. You also get more assurance of the coherence of a program at many levels. Ada can help reduce the cost of software maintenance by shifting the effort to creating a sound system the first time, rather than working harder, more often, and at greater expense, to fix bugs found later in systems already in production. Applications that have reliability needs, long term maintenance requirements, or safety/security concerns are those for which Ada has a proven track record.

It's becoming increasingly common to find systems implemented in multiple languages, and Ada has standard interfacing facilities to allow Ada code to invoke subprograms and/or reference data structures from other language environments, or vice versa. Use of Ada thus allows easy interfacing between different technologies, using each for what it's best at.

We hope this guide has provided some insight into the Ada software engineer's world and has made Ada more accessible to programmers already familiar with programming in other languages.

CHAPTER

FOURTEEN

REFERENCES

The Ada Information Clearinghouse website http://www.adaic.org/learn/materials/, maintained by the Ada Resource Association, contains links to a variety of training materials (books, articles, etc.) that can help in learning Ada. The Development Center page http://www.adacore.com/knowledge on AdaCore's website also contains links to useful information including vides and tutorials on Ada.

The most comprehensive textbook is John Barnes' *Programming in Ada 2005*, which is oriented towards professional software developers.