Ada95

Ada is a general-purpose, internationally standardized computer programming language developed by the U.S. Department of Defense (DoD) to help software designers and programmers develop large, reliable applications. The Ada 95 (1995) version [AdaLRM 95] supersedes the 1983 standard Ada 83. It corrects some shortcomings uncovered from nearly a decade of using Ada 83, and exploits developments in software technology that were not sufficiently mature at the time of Ada's original design. Specifically, Ada95 provides extensive support for object-oriented programming (OOP), efficient real-time concurrent programming, improved facilities for programming in the large, and increased ability to interface with code written in other languages (1).

The Ada language was developed explicitly to support software engineering - it supports principles of good software engineering and discourages poor practices by prohibiting them where possible. Features supporting code clarity and encapsulation (use of packages, use of generic packages and subprograms with generic parameters, and private and limited private types) provide support for maintenance and reusability. Ada95 also provides full support for object-oriented programming, which allows for a high level of reusability:

- Encapsulation of objects and their operations
- OOP inheritance- allowing new abstractions to be built from existing ones by inheriting their properties at either compile time or runtime
- An explicit pointer approach to polymorphism- the programmer must decide to use pointers to represent objects [Brosgol 93]
- Dynamic binding

Ada95 also provides special features (hierarchical libraries and partitions) to assist in the development of very large and distributed software components and systems (1).

Ada95 improves the specification of previous Ada features that explicitly support concurrency and real-time processing, such as tasking, type declarations, and low-level language features. A Real-Time Programming Annex has been added to better specify the language definition and model for concurrency.

Ada95 has its own terminology. All Ada95 programs share a basic structure:

```
with Package_Name; use Package_Name;

procedure Program_Name is

  Variable : Some_Type;

begin

  Statement_1;
  Statement_2;

end Program_Name;
```

The **procedure** is basically the program's name.

package, a source file that stores certain commands that do such things as print text, perform mathematical functions, etc. Compare these to the header files in C.

variable, an area of memory in which a value such as a number, a character, or a word is stored.

statement, a command that performs a specific function.

Inheritance

A particular issue in Ada95 is multiple inheritance. Ada95 provides several mechanisms to support multiple inheritance, where multiple inheritance is a means for incrementally building new abstractions from existing ones. Specifically, Ada supports multiple inheritance module inclusion (via multiple with/use clauses), multiple inheritance "is-implemented-using" via private extensions and record composition, and multiple inheritance mixins via the use of generics, formal packages, and access discriminants. The Ada inheritance features support type extension so that data definitions and interfaces may be customized for an application (2).

Reusable

One of the design goals of Ada was to facilitate the creation and use of reusable parts to improve productivity. Ada95 provides features to develop reusable parts and to adapt them once they are available. Packages, visibility control, and separate compilation support modularity and information hiding. Reusable code is developed in many ways. Code may be scavenged from a previous project. A reusable library of code may be developed from scratch for a particularly well-understood domain, such as a math library. Reusable code may be developed as an intentional byproduct of a specific application. Reusable code may be developed a certain way because a design method requires it (2).

Example:

General-purpose stack abstraction:

```
------------------------------------------------------------------
generic
   type Item is private;
package Bounded_Stack is
   procedure Push (New_Item    : in     Item);
   procedure Pop  (Newest_Item :    out Item);
   ...
end Bounded_Stack;
```

--

Renamed appropriately for use in current application:

```
with Bounded_Stack;

...

  type Tray is ...
  package Tray_Stack is
    new Bounded_Stack (Item => Tray);
```

Packages

Abstraction and encapsulation are supported by the package concept and by private types**.** Packages are the principal structuring facility in Ada. They are intended to be used as direct support for abstraction, information hiding, and modularization. For example, they are useful for encapsulating machine dependencies as an aid to portability. A single specification can have multiple bodies isolating implementation-specific information so other parts of the code do not need to change. Encapsulating areas of potential change helps to minimize the effort required to implement that change by preventing unnecessary dependencies among unrelated parts of the system (2).

A package called Directory could contain type and subprogram declarations to support a generalized view of an external directory that contains external files.

```
package Directory is

  type Directory_Listing is limited private;

  procedure Read_Current_Directory (D : in out Directory_Listing);
  generic
    with procedure Process (Filename : in String);
  procedure Iterate (Over : in Directory_Listing);

  ...

private

  type Directory_Listing is ...

end Directory;
```

Polymorphism

Polymorphism is a means of factoring out the differences among a collection of abstractions so that programs may be written in terms of the common properties. Polymorphism allows the different objects in a heterogeneous data structure to be treated the same way, based on dispatching operations defined on the root tagged type. This eliminates the need for case statements to select the processing required for each specific type.

Example

An array of type Employee_List can contain pointers to part-time and full-time employees (and possibly other kinds of employees in the future):

```
--------------------------------------------------------------------------------
package Personnel is
   type Employee  is tagged limited private;
   type Reference is access all Employee'Class;
   ...
private
   ...
end Personnel;
```

```
--------------------------------------------------------------------------------
with Personnel;
package Part_Time_Staff is
  type Part_Time_Employee is new Personnel.Employee with
    record
      ...
    end record;
  ...
end Part_Time_Staff;
--------------------------------------------------------------------------------
with Personnel;
package Full_Time_Staff is
  type Full_Time_Employee is new Personnel.Employee with
    record
      ...
    end record;
  ...
end Full_Time_Staff;
--------------------------------------------------------------------------------

...

type Employee_List is array (Positive range <>) of
Personnel.Reference;

Current_Employees : Employee_List (1..10);

...

Current_Employees(1) := new Full_Time_Staff.Full_Time_Employee;
Current_Employees(2) := new Part_Time_Staff.Part_Time_Employee;
```

...

rationale

Exception handling

In Ada you can write exception handlers that deal with predefined (constraint_error, tasking_error, program_error, storage_error) or user-defined exceptions. When exceptions are raised control does not pass to the operating system but appropriate action can be initiated inside the program in what is called the exception handler.

Examples of user-defined exception declarations:

```
Singular : exception;
 Error    : exception;
Overflow, Underflow : exception;
```

Example of an exception handler:

```
begin
      Open(File, In_File, "input.txt");
    exception
      when E : Name_Error =>
        Put("Cannot open input file : ");
        Put_Line(Exception_Message(E));
        raise;
    end;
```

Common criticisms of Ada

The Ada language is very large - yes, that is true.

Ada is difficult to learn - certainly, if you are moving from C to C++, it seems that the steps are small. However, coming at C++ or Ada from scratch, there may not be as much of a difference, though it is true that you need to know more Ada to write your first Ada program than C++ to write your first C++ program.

Ada compilers are expensive - GNAT is made freely available (though you will get better support, and more up to date releases of the tool set, if you pay).

Lack of library support - there are more C++ libraries in existence than Ada bindings.

Ada is only for military applications - well, the American DoD was the driving force behind Ada. Ada was designed to have general-purpose applicability.

A minimal comparison of Java with C++ and Ada 95

	Java	**C++**	**Ada 95**
Inheritance	Single (but with multiple subtyping)	Multiple	Single (but supports MI)
Preprocessor	No	Yes	No
Separate Interface/Implementation	No (interface generated from code)	Yes (header files)	Yes (specifications)
Garbage Collection	Yes	No	Yes

Operator Overloading	No	Yes	Yes
Pointer Arithmetic	No	Yes	No
Generics	No (but extensive polymorphism)	Yes ("templates")	Yes
Exceptions	Yes	Yes	Yes

Reference:

1. http://www.sei.cmu.edu/str/descriptions/ada95.html

2. **Ada 95 Quality and Style Guide**

   http://www.informatik.uni-stuttgart.de/ifi/ps/ada-doc/style_guide