Ada 95 Fundamentals

"The Analytical Engine … is not merely adapted for tabulating the results of one particular function and of no other, but for developing and tabulating any function whatever."

- Ada Lovelace

widely acknowledged as the world's first computer programmer

Overview of Contents - 1

Overview of Contents - 2

Overview of Contents - 3

Overview of Contents - 4

What to Expect

- This is a very lab oriented course.
- There are 20 sections of materials and most sections end with at least one lab.
- You will learn more about how to program in Ada by actually programming in Ada than any other way.
- Doing the labs is essential to making progress in this course.
- Many times you will find that the lab for one section is the starting point for a lab in a subsequent section.
- So do the labs. Get the labs working.
- Try to do the labs without first referencing the solutions.
- If you are really stuck, peek at the solutions if you must, but don't simply cut and paste them into your work.
- At a minimum, read through the solution code, type the code in yourself, and try to understand what you are entering and why it works as it does.

Section 1
Introduction

A Brief History of Ada - 1

- 1974 – U.S. DoD realized that it was spending far too much on software development, particularly on embedded systems.
  - The DoD and its contractors were using hundreds of different programming languages
- 1975 – To control costs and develop more reliable systems, DoD decided to try to choose a single standard programming language
- 1977 – Concluded that no existing language met their requirements for reliability, maintainability, and efficiency.
  - Decided that Algol 68, Pascal, and PL/I were reasonable starting points
- DoD then held a competition to design a new programming language that would meet the requirements.

A Brief History of Ada - 2

- The selection process
  - 15 companies submitted proposals
  - 4 received contracts to go forward
  - 2 finalists selected
  - CII Honeywell Bull (France) won the contest
- 1979 - Language was named in honor of the Lady Ada Augusta, Countess of Lovelace (a.k.a. Ada Lovelace)
  - Ada Lovelace worked with Charles Babbage who designed the first “programmable mechanical computer”, the Analytical Engine. She wrote several programs for it and thus is credited as being the world's first computer programmer.
- The language eventually became known as Ada 83

A Brief History of Ada - 3

- 1987 – Ada 83 becomes an ISO Standard
  - Supports Exceptions; Has Built-in Multi-tasking,
  - Includes suite of compiler validation tests

This is when people started calling the original Ada, “Ada 83”

- 1995 – Revised Ada standard (Ada 95) published
  - General Focus: **Adding Object Oriented Features**
  - Added extended or tagged types
  - Added a hierarchical library facility
  - Added greater ability to manipulate pointers/references
- 2005 – “Amended” language standard (Ada 2005)
  - General Focus: **Enhanced Object Model**
  - Added Java-like “interfaces” and multiple interface inheritance
  - Added Constructors
  - Added calls to primitive operations using Object.Operation(...) syntax
  - Added to standard library - containers (Lists, Linked Lists, Sets, Vectors, etc.)

A Brief History of Ada - 4

- 2012 – Further revised standard (Ada 2012)
  - Programming by Contract (pre and post conditions along the lines of Eiffel)
  - Mapping multiple tasks onto multiple processors
  - More standard library functionality (Trees and Queues)
  - Iterators
  - Single Line Functions

What version of Ada is this course about?

- This course will use Ada 95 – the first Ada version with Object Oriented Programming capabilities
- Ada 83 was said to be Object Based instead of Object Oriented. There was no “class” concept and no dynamic binding (a.k.a. polymorphism, a.k.a. dynamic dispatching)

Some Ada Strengths

- Two kinds of applications stand out where Ada is particularly relevant: The very large and the very critical.
- Readability
  - Designed from the start with the notion that more time is spent reading, understanding, and maintaining programs than is spent in original development.
- Reduction in Debugging Time
  - Ada tries to catch as many errors as possible, as early as possible (generally at compile time)
- Is often possible to prove that an Ada program is correct.
- Efficiency/Speed
  - Ada is designed to be efficiently implementable since one of its key application domains is real-time, embedded systems

Where is Ada Used?

- Embedded Systems
- Avionics Software
- Air Traffic Control systems
- Global Positioning Systems (GPS)
- Medical Diagnostic Software
- Cell Phone Software
- High Speed Trains

What are Ada's Capabilities?

- Packages (modules) of related types, objects, and operations
- Packages and types can be made generic (parameterized through a template) to create reusable components
- Errors are indicated by exceptions
- Multi-Tasking is handled in the language itself
- Many predefined libraries for I/O, String manipulation, command line interface, etc.
- Object-oriented programming is supported (added with Ada 95)
  - Ada 95 was the first internationally standardized object-oriented programming language
- Interfaces to other languages (such as C, Fortran, COBOL) are included in the language itself. Ada can also be compiled to run in a Java Virtual Machine

An Ada “Compilation Unit” - 1

- The syntax of an Ada programming is formally and fully defined in the Ada Reference Manual (ARM).
- The ARM uses Extended Backus-Naur Form – EBNF to define Ada syntax
- We'll “learn” EBNF itself by example instead of by formal definitions
- The EBNF definition of a compilation unit is:

```
compilation_unit ::= context_clause library_item

library_item ::= package_declaration | package_body |
        subprogram_declaration | subprogram_body
```

- `::=` stands for “is defined as”
- `|` means “or”

An Ada “Compilation Unit” - 2

```
compilation_unit ::= context_clause library_item

library_item ::= package_declaration |
   package_body | subprogram_declaration |
   subprogram_body
```

- A *compilation_unit* is a *context_clause* followed by a *library_item*
- A library_item is one of the listed items
- This is all a bit formalistic without an example to go with it.

An Ada Program – a compilation unit

```
-- Print a simple message
with Ada.Text_IO;

procedure Hello is
   Greeting : constant String := “Hello Ms Lovelace”;
begin
   Ada.Text_IO.Put (Greeting);
   Ada.Text_IO.New_Line;
end Hello;
```

- There is, of course, a law written somewhere that all introductory programming classes must have a “Hello World” application. We have violated that law, but not in spirit.
- In the above the `context_clause` is the `with Ada.Text_IO;`
- The `library_item` is a `subprogram_body` which is a procedure named `Hello`.

Lab 1: Output a Name and Address - 1

- Goal
  - Write and run your first Ada program to familiarize yourself with the Ada environment used in this course.
- Tools
  - Become familiar with GNAT Programming Studio (GPS) and the Ada compiler tool chain

Lab 1: Output a Name and Address - 2

- Steps (Note: Steps vary by GNAT Studio Versions)
  1. Start GNAT Studio (GPS) using one of the following techniques
     - GPS Shortcut on Desktop
     - Start → GPS
     - Start → All Programs → GNAT → 2014 → GPS
  2. Select *Create new project with wizard* and *OK* or Select *Project → New...*
  3. Select *Single Project* and *Forward*
  4. Name the Project `Lab01_Address`
  5. Place Project in `C:\AdaFundamentals\MyWork\Lab01_Address` directory and select *Forward*
  6. Make sure Ada is the selected language and select *Forward*
  7. Select *Forward* again to bypass the VCS dialog
  8. In the Source Dirs dialog, select the +*Add* button

Lab 1: Output a Name and Address - 3

- Steps
  9. Navigate to `C:\AdaFundamentals\MyWork\Lab01_Address\src` creating directories as needed along the way; select that `src` directory and include it as the Source Directory.
  10. Remove `C:\AdaFundamentals\MyWork\Lab01_Address` as a source directory by checking it and selecting the -*Remove* button
  11. Select *Apply*

Lab 1: Output a Name and Address - 4

- Steps

12. Create a New File (File → New) and type the content of the `Hello` program from just before this lab into the file.
13. Save the file as `Address.adb` in the `src` directory
14. Select Project → Edit Project Properties
15. Select *Main files* tab; +Add; add the `Address.adb` file as a "main file" for this project; and select OK
16. Try building the project by selecting the hammer icon
17. See your first Error/Warning about file name mismatch
18. Fix the error by renaming the procedure to `Address`
19. Try the build again until successful

Lab 1: Output a Name and Address - 5

- Steps

  20. Try running the program using the right pointed triangle icon
  21. See output of “Hello Ms Lovelace”
  22. Now...alter the program to output a name and address on 3 separate lines like:

```
John Q. Public
123 Any Street
St. Louis, MO  63111
```

  23. Note that Strings used as parameters to the Put operation can simply be enclosed in double quotes
  24. Build and run your solution

Lab 1: Output a Name and Address - 6

- Steps

25. Open a DOS Cmd window and navigate to your project directory
    `C:\AdaFundamentals\MyWork\Lab01_Address`
26. Use the `dir` command to look at the contents of the directory
27. Use the *Clean All (Broom icon)* to clean up your project and look at the contents again (Use the *Execute* button to actually clean)

    Some compiler generated files named gnatinspec* remain. These help an IDE (like GPS) locate definitions of symbols.

    But the other files (notably the `.exe` and `.o` files are gone)

    Leaving you with a project file (`.gpr`) and a source directory (`src`)

    Let's build and run this outside of the IDE

Lab 1: Output a Name and Address - 7

- Steps

  28. `gcc -c src\address.adb`

      Note creation of `address.o` (object file) and `address.ali` (Ada Library Information file = additional information used to check that the Ada program is consistent.)

  29. `gnatbind address.ali` (you can leave off the `.ali`)

  30. `gnatlink address.ali` (you can leave off the `.ali`)

      Note the created `address.exe`

  31. Run it by entering simply `address`

Lab 1: Solution

```
-- Output an address
with Ada.Text_IO;

procedure Address is
   Greeting : constant String :=
      "Hello Ms Lovelace";
begin
   Ada.Text_IO.Put ("John Q. Public");
   Ada.Text_IO.New_Line;
   Ada.Text_IO.Put_Line ("123 Any Street");
   Ada.Text_IO.Put ("St. Louis, MO  63111");
   Ada.Text_IO.New_Line;
end Address;
```

Lab 1: Some "Takeaways"

- Comments start with `--`
- Ada is picky about file names matching compilation unit names
  - A procedure named `Address` needs to be in a file named `address.adb`
- Ada is **not** picky about case
  - Ada is a ***case insensitive*** language: `Address`, `address`, and `AdDRESS` are all the same to Ada. This is true for variable names, procedure names, file names, etc.
- Statements are all terminated with semicolons
- Procedure `end` statements have an ***optional*** inclusion of the name of the procedure they are ending. Including this is considered good form in Ada and can help the compiler find errors in your code.

Lab 1: Review Questions

1. What is the name of the new procedure defined in the solution?
2. Where are `Put`, `Put_Line`, and `New_Line` defined?
3. What services do you think `Ada.Text_IO` provides for us?
4. Do you think `Ada.Text_IO.Put(1);` will work?

Configuring GPS for an Ada Version

- For this course, you will want a GPS project to only support and compile a particular Ada version (Ada 83, Ada 95, Ada 2005, or Ada 2012).
- For a given project, select
  Project → Edit Project Properties →
    Switches Tab (on left) → Ada Tab (on top)
- In Syntax section of dialog, select the Ada version you want (Ada 95 mode)

Section 2
Basic Ada Code Organization

Use Clauses

- Using `Ada.Text_IO.Put(...)` can get pretty wordy.
- *Use clauses* can allow you to abbreviate full procedure names

```
-- Print a message
with Ada.Text_IO;
use Ada.Text_IO;  -- use clause

procedure Hello2 is
begin
   Put_Line("Hello, world!");
end Hello2;
```

Variables, Integers, Parameters, and Exceptions

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Compute is

   procedure Double(Item : in out Integer) is
   begin -- procedure Double
      Item := Item * 2;
   end Double;

   X : Integer := 1;
begin
   loop
      Put(X);
      New_Line;
      Double(X);
   end loop;
end Compute;
```

Local Procedure Double

Local Variable X, Type Integer

Infinite Loop – Eventually the number will get too big to be represented in an Integer. Then what? Silently give wrong answer? (Rollover) Corrupt memory? Raise an Exception?

Program Units

- A compilation unit or program unit can be
  - Subprogram – Procedures and functions
  - Package – collection of related entities (other units)
    - The most important kind of program unit
  - Task Unit – defines a computation that can occur in parallel with other computations
  - Protected Unit – coordinates sharing between parallel computation (new in Ada 95)
  - Generic Unit – helps create reusable components – similar to C++ and Java Templates
- Most Ada programs are a set of a large number of packages with one procedure used as the “main” procedure to start the program. The main procedure need not be named “main”.

Specifications and Bodies

- Program units (e.g. subprograms and packages) normally consist of 2 parts
  - Specification: Information visible to other program units (and a private part that is not visible)
    - Somewhat analogous to a C/C++ header file (`.h`)
  - Body: Implementation details – not visible to other parts
    - Somewhat analogous to a C/C++ implementation file (`.c` or `.cpp`)
  - The distinction between Specification and Body allows components to be designed and written independently
- The 2 parts are *usually (but not always)* stored in separate files
  - Convention: `.ads` for Specification; `.adb` for Body
  - Separate specification is not required for subprograms (procedures and functions)
  - Separate body may be impossible (e.g. when a package is just a collection of declared constants)

Packages - 1

- The package is Ada's basic unit for defining a group of logically related entities
- We've already used a package called `Ada.Text_IO`
- Here's an excerpt from the `Ada.Text_IO` package *specification*

```
package Text_IO is
  type File_Type is limited private;
  type File_Mode is (In_File, Out_File, Append_File);
  procedure Create (File : in out File_Type;
                    Mode : in File_Mode := Out_File;
                    Name : in String := "");
  procedure Close (File : in out File_Type);
  procedure Put_Line (File : in File_Type; Item : in String);
  procedure Put_Line (Item : in String);
end Text_IO;
```

Packages - 2

```
package Text_IO is
  type File_Type is limited private;
  type File_Mode is (In_File, Out_File, Append_File);
  procedure Create (File : in out File_Type;
                    Mode : in File_Mode := Out_File;
                    Name : in String := "");
  procedure Close (File : in out File_Type);
  procedure Put_Line (File : in File_Type; Item : in String);
  procedure Put_Line (Item : in String);
end Text_IO;
```

- The package defines a `File_Type` that represents an opened or created file. (Note: `limited private` will be discussed later.)
- Also defines a type called `File_Mode` which can only take one of the 3 specified (enumerated) values.
- Type definitions are followed by a set of subprograms that work with things of `File_Type`

Packages - 3

```
package Text_IO is
  type File_Type is limited private;
  type File_Mode is (In_File, Out_File, Append_File);
  procedure Create (File : in out File_Type;
                    Mode : in File_Mode := Out_File;
                    Name : in String := "");
  procedure Close (File : in out File_Type);
  procedure Put_Line (File : in File_Type; Item : in String);
  procedure Put_Line (Item : in String);
end Text_IO;
```

- Note that the procedures are “inside” that package. But their bodies are not in the specification.
  - This is Ada's mechanism for Information Hiding
  - The `File_Type` is an Abstract Data Type (ADT) a precursor to what Object-Oriented languages would call a “Class”
- Note that there are 2 procedures named `Put_Line` which differ based on the arguments they can accept. This is Ada's first form of “overloading” a name.
- There must also be a package body somewhere that defines these operations.

Compilation Units (Again)

- More details on what constitutes an Ada Compilation Unit

1. *compilation_unit ::= context_clause library_item*
2. *context_clause ::= {context_item}*
3. *context_item ::= with_clause | use_clause*
4. *with_clause ::= "with" library_unit_name {"," library_unit_name} ";"*
5. *use_clause ::= "use" library_unit_name {"," library_unit_name} ";"*
6. *library_item ::= package_declaration | package_body |*
   subprogram_declaration | subprogram_body

- { } means a set of 0 or more
- | means "or"
- ::= means "is defined as"
- Things in double quotes must be used "as is" - e.g keywords or required parts like the semicolons

Review of Basic Ada Structures (Not Data Structures)

- Ada programs are composed of *program units*
- There are different kinds of program units; we've seen *subprograms* and *packages*
- Subprograms define processing algorithms and can be *procedures* or *functions* - We haven't seen functions yet. They return a single value.
- *Packages* are the main Ada tool for grouping related things together and separating unrelated things
- In general, a program unit has two parts, a *specification* and a *body*
- Ada compilers compile *compilation units* – a program unit's specification or body
- Either a specification or a body can be preceded by a *context clause*
- A context clause is a set of *with clauses* and possibly *use clauses*
  - The compiler uses use clauses to search libraries to complete definitions that may be abbreviated in the actual program unit

Thought Question - 1

- Can an Ada compiler compile a package specification even if the implementation details (the package body) has not been written yet?

Thought Question - 2

- Can an Ada compiler compile a package specification even if the implementation details (the package body) has not been written yet?
- YES! A package specification is a compilation unit and thus can be compiled. The compiler could not produce an executable program until the package body has been implemented.
- A “dummy” body might be developed and then used by the team working on the code that uses the package while the team developing the package is still creating the real body.

Lab 2: Simple Sum

- Goals
  - Understand the differences between the `Ada.Text_IO` and `Ada.Integer_Text_IO` packages
  - Be able to write prompts to the user for information and read responses from the user
  - Compute a simple result and display it
  - Learn some of the limitations of the GPS IDE with respect to interactive I/O
- Notes
  - We saw a use of `Ada.Integer_Text_IO` several slides back
  - Use the GPS IDE to help you determine what procedures are available in the `Ada.Integer_Text_IO` package.
  - Type `Ada.Integer_Text_IO.` and either pause or press Ctrl-Space to see options.

Lab 2: Simple Sum

- Steps
  1. Write an Ada program that prompts the user for two integers, stores those integers in two separate Integer variables, and outputs the sum of the two integers.
- Notes
  - If you run the program inside the GPS IDE and your prompts use `Put` instead of `Put_Line`, your prompts **may** not show up until after you've entered numbers in response to them. This is a limitation of the IDE and not of the Ada programming language. Try entering the numbers as if the prompts were written out and see what happens.
  - After building your executable using the IDE, you can run your executable from a Windows/DOS Command Prompt to see the preferred behavior.

Lab 2: Solution

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure SimpleSum is
   Value1 : Integer;
   Value2 : Integer;

begin
   Ada.Text_IO.Put ("Enter first integer: ");
   Ada.Integer_Text_IO.Get(Value1);

   Ada.Text_IO.Put ("Enter second integer: ");
   Ada.Integer_Text_IO.Get(Value2);

   Ada.Text_IO.Put("The sum is: ");
   Ada.Integer_Text_IO.Put(Value1 + Value2, Width=>4);
   Ada.Text_IO.New_Line;
end SimpleSum;
```

Alternatively use Put_Line

Mixing positional parameter Passing and named parameter Passing. Positionals must come first.

Section 3
Ada Lexical Elements
some of the “small stuff”

Lexical Elements

- Ada compilation units are composed of *lexical elements.*
- Lexical elements include
  - Identifiers: names of procedures, functions, variables, packages, etc.
  - Reserved words: `begin`, `end`, `procedure`, `package`, `if`, etc.
  - Punctuation marks: `; , '`
- Ada is a “free-form” language
  - Line breaks and spaces are mostly irrelevant to the compiler
  - Semicolons terminate statements, keywords (e.g. `begin` and `end`) define blocks
- Ada is case-insensitive (except for the content of Strings inside quotes)
- Consistency of capitalization and indentation are helpful but not required

Style Guide Highlights

- Keywords: lower case
- Identifiers have at least the initial letter capitalized
- Multi-word identifiers use a single underscore to separate words
  - Subsequent words are usually capitalized too
- One statement per line
- Use consistent indentation within a compilation unit
- There are online style guides

Identifiers

- Names for procedures, functions, packages, variables, etc.
- EBNF syntax definition

```
identifier ::= letter { [ “_” ] letter_or_digit }
      letter_or_digit ::= letter | digit
```

- Some compilers may limit the number of significant characters, but Ada compilers have to support at least 200 significant characters
- No two underscores in a row – Not Allowed
- Cannot start or end with an underscore
- Single letter identifiers are generally discouraged

Numeric Literals

- Any number included in a program's source code is a *numeric literal*
- Ada supports two kinds of numeric literals: real and integer
  - Real has a point (decimal point) in it – e.g. `7.0`
  - Integer does not – e.g. `7`
- Exponential notation is also supported – e.g. `1.25E9`
- Underscores can be inserted for easier reading, like commas are used in the U.S. - e.g. `1_000_002`
  - No two underscores in a row, can't start or end with an underscore
- Alternative bases supported
  - `2#1001_1000#` - base 2 = $2^7 + 2^4 + 2^3$ = 128 + 16 + 8 = 152
  - `16#EF#` - base base 16 = 14 x 16 + 15 = 239 = 2#1110_1111#
  - `16#E#E1` - base 16 exponential = 14 x 16^1 = 224

EBNF specification of Numeric Literals

numeric_literal ::= decimal_literal | based_literal

decimal_literal ::= numeral [. numeral] [exponent]

numeral ::= digit { digit | “_” }

exponent ::= “E” [“+” | “-”] numeral

based_literal ::= base “#” based_numeral “#” [exponent]

base ::= numeral

based_numeral ::= extended_digit { extended_digit | “_” }

extended_digit ::= digit | “A” | “B” | “C” | “D” | “E” | “F”

digit := “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

- Note: [ ] means optional; { } means 0 or more; | means or
- Legal? `5.5, 1_305_000.5, 2#1001_2000#, 8#157#, 0x50`

Character and String Literals

- Single character literals are enclosed in single quotes
  - e.g. `'a', '1', 'B', '''`
- String literals are enclosed in double quotes
- To put a double quote inside a string use two double quotes
  - e.g. `“I want to ride my bike.”`
  - `“He said, “”It's too cold to ride.”””`
  - `“”` - empty string
- Control characters (new line, tab, etc.) are not done via “escape characters” like `\n`, `\t`, etc.
- Use a predefined character set
  - `ASCII.NUL`, `ASCII.HT` (horizontal tab), `ASCII.CR` (carriage return)
  - More later

Type Integer - 1

- Using and Defining “Types” is central to programming in Ada
- The Ada type Integer is used when you don't particularly care what the minimum and maximum integer values that can be stored are
- If you care, *and often you do*, don't use Integer (we'll see later how to specify that you care.)
- The compiler will choose a “natural” integer representation for Integer
- It is guaranteed to have a range at least of `–32_767` to `32_767` (So an Integer will be at least 16 bits, but often is longer)
  - Lowest value: `Integer'First`
  - Highest value: `Integer'Last`
  - `'First` and `'Last` are *Attributes* of the type Integer … we'll see more of these later

Type Integer - 2

- Integer Operations:
  - Add: `+`
  - Subtract: `-`
  - Multiply: `*`
  - Integer Divide: `/  7/3 = 2`, `-7/3 = -2`
  - Exponential: `**`
- Assignment: `:=`, not `=`, not `==`
- Comparisons: `=`, `>`, `>=`, `<`, `<=`, `/=` (not equal)
- Integers are not Booleans!!
  - Neither `0`, nor `1`, nor any other number means true or false
  - This helps the compiler catch programming errors early.

Subprogram Declarations and Parameters - 1

- Declaration
- Procedures have no return value (but can alter and return parameter values)
- Functions have a return value
- Example Procedure Declarations: equivalent

```
procedure Average(A, B : in Integer; Result : out Integer);

procedure Average(A : in Integer; B : in Integer;
                  Result : out Integer);
```

- Example Function Declaration:

```
function Average_Two(A, B : in Integer) return Integer;
```

Subprogram Declarations and Parameters - 2

- Parameter modes: `in`, `out`, and `in out`
  - `in` – the parameter's value may be used but not changed
    - `A : in Integer → A :=` *`anything`*`;` Not allowed
  - `out` – the parameter's value may be changed but not used
    - `B : out Integer → anything := … B …;` Not allowed
    - `if B > 3` Not allowed
  - `in out` – the parameter's value may be used and/or changed
  - `in` is the default mode. Don't leave things to default mode.
- Some weird things could be legal

```
procedure Do_This( Integer : in A );
```

  - Would be legal if there is a type `A`. Would create a parameter variable named `Integer`. A just plain bad idea.

Subprogram Bodies and Local Variables - 1

- A subprogram *body* defines the actual algorithm used
- A subprogram body starts with a subprogram specification (a declaration without the final semicolon) followed by `is`.
- Then comes the declaration of any local variables, the keyword `begin`, the statements to be executed, and then the keyword `end` (usually followed by the subprogram name)

```
procedure Average(A, B : in Integer; Result : out Integer) is
begin
  Result := (A + B) / 2;
end Average;
```

Subprogram Bodies and Local Variables - 2

- Local variables only exist as long as the subprogram exists.
- Local variables can be assigned initial values using `:=`
- Functions return values using a `return` statement.

```
function Sum(A, B : in Integer) return Integer is
  Total : Integer := A;
begin
  Total := Total + B;
  return Total;
end Sum;
```

Local Subprograms

- The following defines and uses a local function

```
function Sum_Squares(A, B : in Integer) return Integer is

  function Square(X : in Integer) return Integer is
  begin  -- beginning of function Square
    return X*X;
  end Square;

begin  -- beginning of function Sum_Squares

  return Square(A) + Square(B);

end Sum_Squares;
```

Subprograms: EBNF Specification

```
subprogram_body ::= subprogram_specification “is”
                      declarative_part
                    “begin”
                      sequence_of_statements
                    “end” [designator] “;”

declarative_part ::= { declarative_item }

declarative_item :== object_declaration | subprogram_body

object_declaration ::= identifier_list : [constant] type
                       [“:=” expression] “;”
```

- Note (again) semicolons are statement terminators, like C/C++ and Java. Not statement separators like Pascal.

Lab 3: Simple Sum Again

- Goals
  - Organize code by writing a local subprogram (procedure or function)
  - Use `in` and `out` to indicate parameter mode
- Steps
  1. Using your Lab 2 solution as a starting point, create a procedure within your main procedure which, when given two input integer parameters, computes their sum and returns the sum through another parameter.
  2. Use that procedure within your “main” procedure to compute the sum before printing it out

Lab 3: Solution

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure SumProcedure is
   Value1, Value2, Sum : Integer;
   procedure ComputeSum(Val1 : in Integer;
                        Val2 : in Integer;
                        Sum  : out Integer) is
   begin
      Sum := Val1 + Val2;
   end;
begin
   Ada.Text_IO.Put_Line ("Enter first integer: ");
   Ada.Integer_Text_IO.Get(Value1);
   Ada.Text_IO.Put_Line ("Enter second integer: ");
   Ada.Integer_Text_IO.Get(Value2);

   ComputeSum(Value1, Value2, Sum);
   Ada.Text_IO.Put("The sum is: ");
   Ada.Integer_Text_IO.Put(Sum);
   Ada.Text_IO.New_Line;
end SumProcedure;
```

Section 4
Ada Control Flow and Logic

If Statements

- If statements conditionally execute sequences of statements depending upon whether some conditions are true.
- Example 1

```
if A = B then
  A := B + 1;
else
  A := B - 1;
end if;
```

`then` required

`else` optional

`end if;` required

- Example 2

```
if A = B then
  A := B + 1;
  C := 10;
elsif C > 20 then
  A := C;
elsif C < 0 then
  A := -C;
else
  A := B - 1;
end if;
```

Note the lack of parentheses around condition.
Parentheses allowed, but not required.

Note the spelling of `elsif`

If Statements: EBNF Specification

```
if_statement ::=
  “if” condition “then”
     sequence_of_statements
  {“elsif” condition “then”
     sequence_of_statements}
  [“else”
     sequence_of_statements]
  “end if;”
```

Case Statements

- Case statements are another form of conditional execution.
- Example

```
case A is              -- Execute something depending on A's value
  when 1 =>            -- if A=1, execute Fly then Land
    Fly;
    Land;
  when 3 .. 10    => Put(A);    -- if A is 3 through 10
  when 11 | 14    => null;      -- if A is 11 or 14, do nothing
  when 2 | 20..30 => Swim;      -- if A is 2 or 20 through 30, swim
  when others     => Complain; -- if A is anything else
end case;
```

- `when others` matches any condition that doesn't match anything else
- No `break` statement is necessary
- Compiler will detect missing or duplicate cases at compile time and won't successfully compile when it detects such a thing. *You have to cover all the cases and cover each case only once.*

Case Statements: EBNF Specification

```
case_statement ::=
  “case” expresson “is”
    case_statement_alternative
    {case_statement_alternative}
  “end case;”

case_statement_alternative ::=
  “when” discrete_choice_list “=>”
    sequence_of_statements

discrete_choice_list ::= discrete_choice { |
discrete_choice }

discrete_choice ::= expression |
discrete_range | “others”
```

Simple Loops - 1

- There are a number of looping constructs
- A Simple Loop simply repeats “forever”.
  - Can have an `exit` or `exit when` clause to prevent “infinite” loop.
- Example 1 (inside a subprogram)

```
with Text_IO;
use Text_IO;
.
.
loop
  Put_Line(“Hello again!”);
end loop;
.
.
```

Simple Loops - 2

- Example 2

```
with Text_IO;
use Text_IO;
procedure Print_Squares is
  X : Integer
begin
  loop
    Get(X);
    exit when X = 0;
    Put(X * X);
    New_Line;
  end loop;
end Print_Squares;
```

Could also be written as

```
if X = 0 then exit; end if;
```

Iteration Schemes

- Other *iteration schemes* are built on this basic loop syntax
- While loop

```
while N < 103
loop
  Put(N);
  N := N + 1;
end loop;
```

- For loop

```
for Index in 1 .. 2000
loop
  Put(Index);
end loop;
```

Note the specification of a range.

For Loop Notables

- The *loop variable* (mentioned in the for statement) is both “local to the loop” and *cannot be modified by statements in the loop*.
- The loop variable is incremented by one unless otherwise specified.
  - Can specify reverse order using keyword `reverse`.

```
for Index in reverse 1 .. 2000
loop
  Put(Index);
end loop;
```

For Loop “Gotchas”

```
for Index in 2000 .. 1
loop
  Put(Index);
end loop;
```

- Does nothing – repeats zero times
  - `Index` is initially set to `2000` and `2000` is already greater than `1`.
  - So loop content never executes

```
for Index in reverse 2000 .. 1 loop
```

- Also does nothing because `2000 .. 1` is an empty list
- Doing nothing in reverse order still does nothing.

Nested Loops

```
for OuterCounter in 1 .. 3 loop
  Put(“OuterCounter: ”);
  Put(OuterCounter);
  New_Line;
  for InnerCounter in 1 .. OuterCounter loop
    Put(“   InnerCounter: ”);
    Put(InnerCounter);
    New_Line;
  end loop;
end loop;
```

Note the different `loop` position style.

- Output

```
OuterCounter:          1
   InnerCounter:          1
OuterCounter:          2
   InnerCounter:          1
   InnerCounter:          2
OuterCounter:          3
   InnerCounter:          1
   InnerCounter:          2
   InnerCounter:          3
```

Named Loops for Exiting

```
Search:
for I in 1 .. N loop
  for J in 1 .. M loop
    if condition_OK then
      I_Value := I;
      J_Value := J;
      exit Search;
    end if;
  end loop;
  -- outside of inner loop
end loop Search;
-- Control passes here
```

- If `exit` were used without the loop name, we would only be exiting the inner loop.
- But we've found what we're looking for and want out of the entire search.

Lab 4: Counting Digits - 1

- Goals
  - Use Ada looping to create a function which solves a perhaps non-obvious problem
- Steps
  1. Write an function that computes the number of decimal digits in a supplied non-negative Integer.
  2. If the supplied Integer is negative, just output an error message an return 0
  3. Use the fact that each time you do Integer division by `10`, you are effectively “cutting off” the lowest digit of the number.

Lab 4: Counting Digits - 2

- Algorithm
  - Assume you have at least 1 digit
  - Integer divide by 10 to cut off one digit: 512 / 10 = 51 and increment digit count
  - As long as remaining number (51) is at least 10, you've got another digit to cut off
  - Integer divide by 10 to cut off one digit: 51 / 10 = 5 and increment digit count
  - Now that the remaining number 5 is no longer at least 10, you are done.
- Steps (continued)

  4. Write a main procedure that prompts the user for a number and then computes and outputs the number of digits in the number

Lab 4 Solution

- Review in IDE

Section 5

More Ada Types and Defining Your Own Types

Type Float

- As you would expect, Ada has a predefined type for “real” numbers
- That predefined type is called `Float`
- `Float` is used when you don't care about the minimum or maximum range of the numbers, nor do you care about the minimum accuracy of a value.
- Ada will choose the most “natural” floating point type for the machine.
- `Float` is not appropriate when you **do** care about range or accuracy. We'll visit this later.
- `Float` has all the arithmetic operations (`+`, `-`, `*`, `/`, `**`) and comparison operations (`=`, `/=`, `>`, `>=`, `<`, `<=`) that you'd expect.

Operations on Mixed Types

- **Ada insists that types be correct in operations, and there are no predefined operations for mixing Integers with Floats. This is part of Ada philosophy: *It won't surprise you by carrying out an operation that it isn't clear that you intended.* So you have to make your intentions explicit.**

```
Integer I := 2;
Float A := 3.0;
Float B := I + A;
B := Float(I) + A;
```

- Must explicitly convert `I` to a `Float` to perform floating point arithmetic

Limitations of Floating Point Representation - 1

- Floating point numbers are stored as binary approximations of the intended value using a limited number of bits.
- Results are usually only approximately the value you'd expect. Results become even further “off” as more operations are performed and as the numbers get further from 0.
- You should almost never compare floating point values using =. They will seldom be actually exactly equal.
- This is not unique to Ada. It is part of the standard binary floating point representation of numbers.

Limitations of Floating Point Representation - 2

```
with Ada.Text_IO, Ada.Float_Text_IO; use Ada.Text_IO, Ada.Float_Text_IO;
procedure Pennies is
   Three_Pennies : constant Float := 0.03;
   Total_Dollars : Float := 0.0;
begin
   for Count in 1 .. 100_000 loop
      Total_Dollars := Total_Dollars + Three_Pennies;
   end loop;
   Put("Total_Dollars: ");
   Put(Total_Dollars);
end Pennies;
```

- What do you expect the resulting `Total_Dollars` value to be?
- $0.03 x 100,000 = $3,000
- But actual result is $3002.41
- ***A value as simple as 3 cents cannot be accurately represented as a binary floating point number.***

Fixed Point & Decimal Types

- One way to get the decimal (fractional) values and still maintain accuracy is to use a *Decimal Type* (which is a special form of a *Fixed Point Type*)

```
type Money is delta 0.01 digits 14;
```

- This is our first example of defining our own type.
- This type will accurately represent values from 0.00 to `999_999_999_999.99` (14 digits long).
- More generally a Fixed Point Type declaration will look like:

```
type F is delta D range L .. R;
```

- Where `D`, `L`, and `R` are Real Constants and `D > 0.0`
- For example `type T is delta 0.1 range –1.0 .. +1.0`
- If delta is a power of `10`, then we have a *Decimal Type* and can use `digits` in place of `range`.

Accurate Money Calculations

```
with Ada.Text_IO; use Ada.Text_IO;
procedure BetterPennies is
   type Money is delta 0.01 digits 14;
   Three_Pennies : constant Money := 0.03;
   Total_Dollars : Money := 0.0;
   package Money_IO is new Ada.Text_IO.Decimal_IO(Money);
begin
   for Count in 1 .. 100_000
   loop
      Total_Dollars := Total_Dollars + Three_Pennies;
   end loop;
   Put("Total_Dollars: ");
   Money_IO.Put(Total_Dollars);
end BetterPennies;
```

- Output:

```
Total_Dollars:         3000.00
```

Boolean Type

- Ada provides a predefined type called `Boolean` which can only have 2 values: `True` and `False`
- Comparison operators (`=, /=, >=,` etc.) have a result of type `Boolean`
- Booleans have their own set of binary (“infix”) operators too
- `and, or, xor` (exclusive or, either but not both)
- Also have a unary (“prefix”) operator `not`
- *Ada compiler can evaluate these in whatever order is most efficient for the machine.*
  - E.g. `if Func(A) > 0 or B /= 17 then` might evaluate the `B /= 17` first, find that to be true and never call the function `Func`.
- “Short circuit” versions of `and` and `or` are available: `and then` and or else
  - They always evaluate left-to-right and will stop evaluation when the answer is known
  - `if K /= 0 and then 1.0/Float(K) > B then`

Creating Types and Subtypes - 1

- A *Type* is a set of values and a set of *primitive operations.* We've seen these with the `Integer` type and the `Float` type and their primitive operations (`+, −, *,` etc.)
- We'll see that the term *primitive operations* can entail more that one or two character operators later.
- In Ada, an *object* of a particular type is a run-time entity that contains/has a value of the type.
- The entity could be a variable or a constant.
- Ada lets you create your own types and this is very often the heart of programming well in Ada
- As such, Ada has a very rich set of capabilities for creating your own types
- We use a *Type Declaration* to create a new type.

```
Type Column is range 1 .. 72;
Type Row is range 1 .. 24;
```

- Wherever you can declare a variable, you can declare a type.

Creating Types and Subtypes - 2

- Ada considers types to be different even if their underlying implementation happens to be the same
- Our `Column` and `Row` types from the previous slide are two distinct types
- A value of one type cannot be assigned to a variable of the other type

```
C : Column;
R : Row := 2;
C := R;  -- NOPE
```

- A value of one type cannot be added to a value of the other type

```
C := C + R;  -- NOPE
```

- Why do you think Ada's designers made it this way?
- What if the types are related to each other and you decide you want to be able to do these things?

That's what Subtypes are for

- A *subtype* is another name for an existing type that may have some additional constraints on it.

```
type Size is range 0 .. 99;
subtype Shoe_Size is Size range 1 .. 20;
subtype Hat_Size is Size range 5 .. 10;
subtype Ring_Size is Size;
```

- Variables of these types can be assigned to each other and operations can occur between mixtures of the types.
- But what happens when:

```
S : Size := 5;
T : Shoe_Size := 7;
H : Hat_Size;
H := S + T;
```

- Sometimes compiler can detect it. Sometimes not.
- `Constraint_Error` will be raised.

Introduction to Ada Strings

- Ada's type `String` is simple and efficient, but some operations may require a little more work than you are used to.
- A `String` is a one dimensional array of characters (We'll visit Arrays more later.)
- To create a `String` variable, you must give Ada a way of determining its length. There is no `String` termination character. `String` variables must have a known size/length.
- Two alternatives:

```
A : String (1 .. 50); -- Variable A holds 50 characters
B : String := “What is your name?” -- Variable B is
                                   -- String (1..18)
```

- Gotcha

```
C : String := “”; -- Variable C is an array of length 0
                  -- It can hold nothing.
```

A Little about String I/O

- As we've seen, the package `Ada.Text_IO` is use for Text/String IO in Ada.
- Writing out Strings is pretty straightforward using `Put` and `Put_Line`
- Reading in Strings is a bit more involved
- `Ada.Text_IO.Get` will get as many characters as the specified String can hold

```
Message : String (1..10);
Ada.Text_IO.Put(“Enter a message: “);
Ada.Text_IO.New_Line;
Ada.Text_IO.Get(Message);
```

- This will get 10 characters from the console input. It will wait until there are 10 characters entered. *You gotta enter at least 10.* If you enter more, those beyond 10 will become part of the input for the next `Get` call.
- ***This is often not what you want.***

A Little *More* about String I/O

- The `Get_Line` procedure can help
- But you need to understand what it does.
- `Get_Line` reads characters into a string variable until one of the following conditions is met:
  1. The string is filled or
  2. The line terminator (platform specific) is found.
- If the line terminator is found before the string is filled, then the remaining characters are not overwritten. They will still contain whatever they had in them before (*perhaps junk.*)
- `Get_Line` will also return an indication of the position of the last character put into the string by `Get_Line`.

```
Procedure Get_Line (Item: out String;
                    Last: out Natural);
```

- `Natural` is a subtype of `Integer` ranging from 0 to `Integer'Last`

Even More about String I/O

```
Message : String (1..80);
Last : Natural;
Ada.Text_IO.Put(“Enter a message: “);
Ada.Text_IO.New_Line;
Ada.Text_IO.Get_Line(Message, Last);
```

- The first `1..Last` characters in `Message` will contain what the user entered.
- The remaining slots in `Message` will be left alone.
- The line terminator character will not be placed in `Message`
- If the user entered Hello, then Message may contain `Hellocytaiytqtqiuagafh...ty`
- How can I get just the part the user entered?

String Slices and Concatenation

`Message(1..5)`

- This *slice* will contain the first 5 characters in `Message`

`Message(1..Last)`

- Will contain characters `1` through whatever the value of `Last` is.

`Message(1 .. 2) & Message(4 .. 5)`

- `&` is the String Concatenation operator

`Message(5 .. 4)`

- Will contain no characters
- `5 .. 4` is called a *null range* (whenever first is greater than last)
- The string of no characters is called the *null string*

Enumerations

- When a variable can have only one of a small set of values, an *enumeration type* can be created.
- Classic examples include:

```
type Day is (Monday, Tuesday, Wednesday, Thursday, Friday,
             Saturday, Sunday);
subtype Weekday is Day range Monday .. Friday;
subtype Weekend is Day range Saturday .. Sunday;
Today : Day;
Today := Sunday;
type Color is (Red, Amber, Green, Blue)
type Stone is (Amber, Beryl, Quartz)
type Mood is (Happy, Ecstatic, Sad, Blue)
```

- Note the overloaded literals: `Amber` and `Blue`.
- Compiler can often tell from context when you are using a literal. In a case where this is not possible, you can use `Color'(Amber)`, `Stone'(Amber)`, `Color'(Blue)`, or `Mood'(Blue)`
- The `Boolean` type is an Enumeration type with only two values – `True` and `False`

Attributes for working with Enumerations

- Ada defines *attributes* of types that can be useful for working with Enumeration types
  - `First, Last`
  - `Pred = Predecessor, Succ = Successor`
  - `Image` (from Enumeration to String), `Value` (from String to Enumeration)
  - `Pos` = Position (from Enumeration to Integer), `Val` (from Integer to Enumeration)
- These attributes are not unique to Enumerations, they work for any *discrete type*

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

`Day'First` is `Monday` and `Day'Last` is `Sunday`

`Day'Pred(Sunday)` is `Saturday` and `Day'Succ(Tuesday)` is `Wednesday`

`X : Day := Monday; Day'Image(X) is “MONDAY”`

`Day'Value(“Friday”)` is `Friday`

`Day'Val(3)` is `Thursday` and `Day'Pos(Tuesday)` is `1` (0 based)

Lab 5: Seasons - 1

- Goals
  - Create an Enumeration Type
  - Write and use functions that help you work with the enumeration type
  - Learn how to do I/O with Enumeration Types by converting them to and from Strings
  - Get some experience using `Ada.Text_IO.Get_Line`
  - Get some experience using string slices.

Lab 5: Seasons - 2

- Steps
  1. Create an Enumeration type (`Season_Type`) for the 4 seasons of the year.
  2. Write a function called `Next_Season` that returns the season that follows a given season.
  3. Write a function called `Last_Season` that returns the season the precedes a given season.
  4. Write a procedure called `Get` which retrieves a “line” of user input (`Get_Line`) and returns the corresponding `Season_Type` value through its output parameter.
  5. Write a procedure called Put which outputs to the console a String representation of a given `Season_Type` parameter.
  6. Write a main procedure that allows the user to enter a season name and displays the season before and after it.

  Note what happens when the user enters a string that is not one of the string representations of a Season?

Lab 5: Solution

- Review Solution in IDE

Section 6

More about Types:

Unbounded Strings, Arrays, Records, and Packages.

Unbounded Strings - 1

- Ada Strings can be tricky to work with.

```
X : String (1..10);
X := “Hello”;
```

- Doesn't work. Gives a Warning at compile time. Will raise a `Constraint_Error` at run time.
- `“Hello”` is not 10 characters long.
- The `Unbounded_String` Type will make our lives a bit easier.
- The `Unbounded_String` type is defined in the package `Ada.Strings.Unbounded`
- The package also provides a number of useful operations on `Unbounded_String`
- For further information, you can visit the Ada Reference Manual (RM) section A.4.5 online.

Unbounded Strings - 2

- Function `To_Unbounded_String` takes a `String` and produces an `Unbounded_String`
- Function `To_String` takes an `Unbounded_String` and produces a `String`
- Function `Length` takes an `Unbounded_String` and returns the number of characters currently stored in it
- Procedure `Append` takes two arguments and appends the second to the end of the first. The first argument must be an `Unbounded_String`. The second can be another `Unbounded_String` or a `String`.
- Function `Element` extracts from a given `Unbounded_String` the character at a given position (index). The leftmost character in the `Unbounded_String` is in position 1
- Procedure `Replace_Element` lets you modify a character at a given position.

Unbounded Strings - 3

- Function `Slice` returns a “slice” of an `Unbounded_String` and returns a `String` (not an `Unbounded_String`)
- Procedure `Insert` takes a `String` and inserts it into an `Unbounded_String` before a specified index
- Procedure `Delete` takes an `Unbounded_String` and two indexes and deletes the characters between those two index positions
- Comparison operations, such as `=, <, >` are also defined in the package, as well as `&` for concatenation.
- Other utilities are included such as `Translate`, `Trim`, `Head`, `Tail`, `Index`, `Find_Token`.
- You can assign (`:=`) one `Unbounded_String` to another without the lengths being the same.

Arrays - 1

- Very much like arrays in many other programming languages, an array type in Ada can contain many components with the same subtype.
- Array indices are not required to start at 0 or 1. You get to define the bounds `(-5 .. 5), (1..100), (-100 .. -3)`. Whatever makes sense for what you are doing.
- Array access is checked at run-time to make sure you are not trying to access using an index that is out-of-bounds. When reading from or writing to the array, Ada will not quietly allow you to go beyond the bounds of an array. An Exception will be raised.
- Ada supports multi-dimensional arrays
- You can supply a *pragma* to ask the compiler to *pack* arrays. This is asking the compiler to try to store the array in a memory efficient way. Think of arrays of Booleans.
- Referring to a value in an array looks like calling a function. If you change the array to a function, code that uses the array won't have to change.
- You can define array types that don't have completely fixed boundaries, Unconstrained Arrays. We'll talk about them more later

Arrays - 2

```
type Table is array(1 .. 100) of Integer;
My_Table : Table;
My_Table(1) := 205;

type Schedule is array(Day) of Boolean;
My_Schedule : Schedule;
My_Schedule(Friday) := False;

A : array (Integer range 1 ..6) of Float;
A : array (1..6) of Float; -- equivalent since a
                           -- range of 1..6 implies
                           -- type Integer
A(1) := 3.14159;

Hours_Worked : array (Day range Mon .. Fri) of Float;
-- Equivalent if Weekday is the Mon .. Fri subtype of Day
Hours_Worked : array (Weekday) of Float;

type Grid is array(-100 .. 100, -100 .. 100) of Float;
Temperature : Grid;
Temperature(0, 0) := 100.0;

type Packed_Bool_Array is array (1 .. 8) of Boolean;
pragma Pack (Packed_Bool_Array)
-- Compile will try to compress the entire array; maybe into a single byte
```

Records

- Types can be a complex collection of other types
- The primary method for putting such types together is a *record* (analogous to a C struct)

```
type Date_Type is
   record
      Day   : Integer range 1 .. 31;
      Month : Integer range 1 .. 12;
      Year  : Integer range 1 .. 10_000 := 2015;
   end record;

Ada_Birthday : Date_Type;

Ada_Birthday.Month := 12;
Ada_Birthday.Day := 10
Ada_Birthday.Year := 1815

type Complex_Type is
   record
      Real_Part, Imaginary_Part : Float := 0.0;
   end record;

X : Complex_Type;
X.Real_Part := 1.0; X.Imaginary_Part := 12.0;
```

Packages - 1

- Packages are Ada's main way of dividing code into separately compilable modules.
- Packages potentially have 3 parts that generally reside in 2 files
  - The package specification – the visible part – in a .ads file
  - The private part – also in the .ads file
  - The package body

```
package A_Sample_Package is
   type Range_10 is range 1 .. 10;
end A_Sample_Package;
```

- No private part.
- No body necessary.

Packages - 2

```
package Package_With_Private is
   type Private_Type is private;

private
   type Private_Type is array (1 .. 10) of Integer;

end Package_With_Private;
```

- Has a private part. Users cannot know that Private_Type is an array.
- We'll talk about this more later.
- Still doesn't need a body.

Packages - 3

```
package Package_With_Body is
   type Basic_Record is private;
   procedure Set_A (This : in out Basic_Record;
                    An_A : in Integer);
   function Get_A (This : Basic_Record) return Integer;

private
   type Basic_Record is
      record
         A : Integer;
      end record;

   type Private_Type is array (1 .. 10) of Integer;

end Package_With_Body;
```

- This is just the specification.
- This one needs a body to define the Set_A and Get_A functions.

Packages - 4

```
package body Package_With_Body is
   procedure Set_A (This : in out Basic_Record;
                    An_A : in Integer) is
   begin
      This.A := An_A;
   end Set_A;

   function Get_A (This : Basic_Record) return Integer is
   begin
      return This.A;
   end Get_A;

end Package_With_Body;
```

- This is the package body.

Lab 6: Contacts - 1

- Goals
  - Gain experience with `Unbounded_String`, records, arrays, and packages
- Steps
  1. Create a package specification for a package named `Contacts` (`Contacts.ads`)
     - You should not need to create a package body for this lab
  2. Your package should contain a type named `Address_Type` which is a record containing 4 `Unbounded_String` objects (Street, City, State, and Zip)
  3. It should also contain a type named `Date_Type` which is a record containing day number, month number, and year number. Similar to what we saw a few slides back.
  4. It should also contain a type named `Contact_Type` which is a record containing
     - `Name : Unbounded_String`
     - `Address : Address_Type`
     - `Birthday : Date_Type`
  5. Finally, it should contain a type named `Contact_Book` which is an array of 100 `Contact_Type` objects.

Lab 6: Contacts - 2

- Steps continued

6. Create a main procedure in a separate file (e.g. `testcontacts.adb`)
7. Declare a variable of type `Contact_Book`, create a date, address, and name and populate an element in the `Contact_Book`
8. Populate at least one more element in the `Contact_Book` and try printing out some contact information from your `Contact_Book`.

Lab 6: Contacts - 3

- Review Solution in IDE

Section 7

Private Types, Limited Private Types, and a Little Bit of Object Oriented Programming

Private Types - 1

- If you simply declare (in a package specification) record types as we've seen and done so far, then any other code that uses *with* to get access to your package and types can directly access the parts of the record.
- We did this in the lab at the end of the previous section.
- In general, this is not good practice because you'd like to reserve the right to change your implementation without breaking other code that depends on that implementation. (This is unnecessarily revealing implementation details.)
- To hide these details we'll need to hide the implementation details by making the type *private*.
- By declaring a type to be *private* you make it such that code outside the package which uses the type cannot directly access the attributes of the type.
- **You also disable the normal operations on that type (e.g. +, −, etc.)**
- After declaring a type as private, you then list the subprograms that you want to permit as publicly accessible operations on that type.
- Some terms commonly applied: *information hiding, encapsulation*

Private Types - 2

- You can also make the implementations of various subprograms (procedures and functions) that work with the type private.
- If you are familiar with Object Oriented Programming, you should start to see the underpinnings of defining a “class” here. What we are really doing is defining an Abstract Data Type (ADT), a precursor to the notion of a “class” as it is found in C++, Objective-C, Java, Python, etc.
- Example package specification (in a file named `Keys.ads`)

```
package Keys is
   type Key is private;
   Null_Key : constant Key;         -- “deferred” constant
   procedure Get_Key(K : out Key); -- get a new key value
   function “<”(X, Y : in Key) return Boolean; – True if X before Y
private                                 -- here starts the private part
   Max_Key : constant := 2 ** 16 – 1;  -- a private constant
   type Key is range 0 .. Max_Key;     -- now we've defined the type
                                        -- privately
   Null_Key : constant Key := 0;       -- now we've completed our
                                        -- constant
end Keys;
```

Private Types - 3

```
package Keys is
   type Key is private;
   Null_Key : constant Key;        -- “deferred” constant
   procedure Get_Key(K : out Key); -- get a new key value
   function “<”(X, Y : in Key) return Boolean; -- True if X before Y
private                            -- here starts the private part
   Max_Key : constant := 2 ** 16 - 1;  -- a private constant
   type Key is range 0 .. Max_Key;     -- now we've defined the type

   Null_Key : constant Key := 0; -- now we've completed our constant
end Keys;
```

- Key is actually a numeric type, but users of the type Key can't use the normal operations of a numeric type (`+`, `-`, `/`, `*`, `<`, `>`, etc.). That is they can't behave as if they know it is a numeric type.
- We can define an operator “<” for our private type. “<” is a valid “name” for a function
- The assignment operator (`:=`) and the equality comparison operators (`=`, `/=`) are still available for a private type.

Private Types - 4

- We will need a package body to provide the implementation.

```
package body Keys is
   Last_Key_Used : Key := 0;

   procedure Get_Key(K : out Key) is
   begin
      Key := Last_Key_Used + 1;
      Last_Key_Used := Key;
   end Get_Key;

   function “<”(X, Y : in Key) return Boolean is
   begin
      return X < Y;
   end “<”;
end Keys;
```

Limited Private Types

- What if we don't even want the default assignment and equality operations for our private type? Then we can declare it to be a *limited private* type.

```
package Keys is
   type Key is limited private;
   Null_Key : constant Key;        -- “deferred” constant
   procedure Get_Key(K : out Key); -- get a new key value
   function “<”(X, Y : in Key) return Boolean; – True if X before Y
   function “=”(Left, Right : in Key) return Boolean;
private                                 -- here starts the private part
   Max_Key : constant := 2 ** 16 – 1;   -- a private constant
   type Key is range 0 .. Max_Key;      -- now we've defined the type
privately
   Null_Key : constant Key := 0;        -- now we've completed our
                                        -- constant
end Keys;
```

- Defining the “=” operator implicitly defines the “/=” operator.
- We cannot override the assignment operator :=. To achieve something like this, we'll need to learn about User-Controlled Initialization, Finalization, and Assignment. This comes later.

Object Oriented Programming Overview - 1

- Software development has always involved using approaches to manage complexity.
- One approach, called *Functional Decomposition,* is the breaking down of a program's overall “function” into smaller parts (subprograms). This is so common now we barely think of it, but it was at one time a major innovation.
- In an *Object-Based* approach, a system is decomposed (broken down) into of “objects” that contain both data and a set of operations that can be performed on that data. The system is then described and programmed in terms of these objects. The objects represent real or abstract “things” important to the problem being solved.
- These objects become Types. In other programming languages they are not usually explicitly called “types”. In Ada they are.

Object Oriented Programming Overview - 2

- One of the classic examples is to think about creating a Type to represent a *Vehicle* that our system somehow uses.
- We define a set of *attributes* of the type Vehicle that are important for our program's purpose, and a set of operations that access and manipulate that data for a Vehicle.
- All the information hiding we've already seen with private types and private operation definitions is essential to making this arrangement beneficial.

Object Oriented Programming Overview - 3

- One essential (probably *the* essential) thing that an *Object Oriented (OO)* approach adds to an *Object Based* approach is the idea of one type *inheriting* data and operations from another type and then extending or redefining both the data and the operations.
- If our Vehicle type has a data attribute indicating the number of wheels and an operation *drive_to(location).* Then if we were able to create a Bus type which inherits from Vehicle (sometimes referred to as extending Vehicle), then the Bus would have the number of wheels data attribute and the drive_to(location) operation via inheritance. The type might need to redefine the drive_to(location) operation (override) and add new attributes.

Object Oriented Programming Overview - 4

- Grady Booch (a well known Software Engineer, author, and OO advocate) defined OO programming as, “OO programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.”
- It is difficult to use OO programming if the programming language does not support inheritance and polymorphism (dynamic dispatching of operation calls based upon run-time types of variables.)

OO Programming in Ada - 1

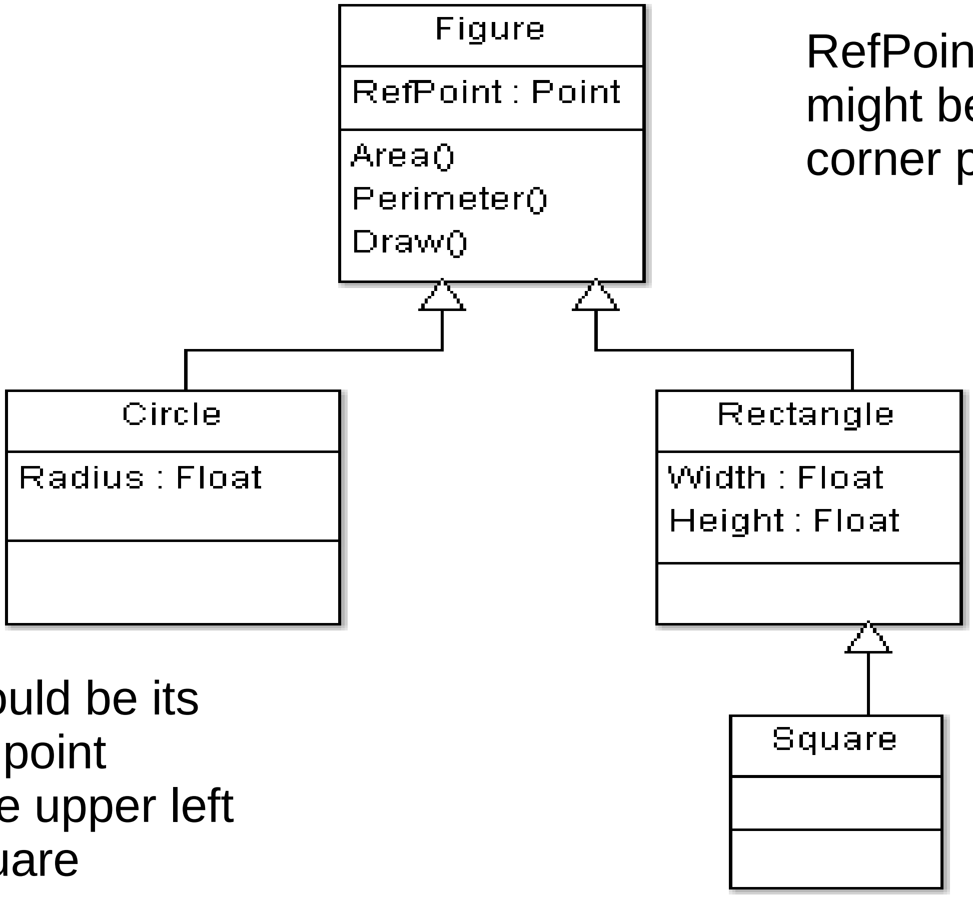
- Inheritance allows us to define new types as extensions of existing types.
- The new types inherit all the operations and data of the types they extend.
- New types are called *children* or *derived types.* The types extended are called *parents* or *base types.* (Other terminologies are sometimes used.)
- Derived types can add new data and new operations.
- A different/enhanced set of attributes and operations may be available for a derived type than are available for the base type.
- ***The system must be able to invoke a different version of a function or procedure at run-time based on the type of object used. (This is known as: Dynamic Binding, Dynamic Dispatching, or Polymorphism)***
- How does Ada support this? (Note: These features were added in Ada 95.)
- ***Tagged types.***
- A tagged type is one that carries around with it a “tag”. A tag is a hidden component of the type that contains an indication of the actual type of the object and thus allows the system to distinguish between types at run-time.

OO Programming in Ada - 2

- Another classic example: Graphical Figures to be drawn on screen.
- We create a package called Figures and define different types of figures (circles, squares, rectangles, etc.)
- We start out creating a Figure type and the data (e.g. a reference point indicating where on the screen the Figure is to be drawn) and a set of operations that we need for the type (e.g. Area, Perimeter, Draw)
- We observe that a Circle is a kind of Figure and should have all those attributes and operations.
- We observe that a Rectangle is a kind of Figure and should have all those attributes and operations.
- We further observer that a Square is a kind of Rectangle and should have all the same attributes and operations of a Rectangle.

OO Programming in Ada - 3

- A simple (UML) representation of this family of types



RefPoint of a Rectangle might be its upper left hand corner point.

RefPoint of a Circle could be its center, but could be a point outside the circle in the upper left hand corner of the square enclosing the circle.

OO Programming in Ada - 4

```
package Figures is
   type Point is
      record
         X, Y: Float;
      end record;

   type Figure is tagged
      record
         RefPoint : Point;
      end record;

   function Area(F : Figure) return Float;
   function Perimeter(F : Figure) return Float;
   procedure Draw(F : Figure);

   type Circle is new Figure with
      record
         Radius : Float;
      end record;
```

OO Programming in Ada - 5

```
   function Area(C: Circle) return Float;
   function Perimeter(C: Circle) return Float;
   procedure Draw(C: Circle);

   type Rectangle is new Figure with
      record
         Width, Height : Float;
      end record;

   function Area(R: Rectangle) return Float;
   function Perimeter(R: Rectangle) return Float;
   procedure Draw(R: Rectangle);

   type Square is new Rectangle with null record;

end Figures;
```

- Tagged types have allowed us to solve the problem of extending a type.
- But we haven't seen how this gives us any real dynamic dispatching.

OO Programming in Ada - 6

- We now have a hierarchy of types derived from the type Figure.
- It is important to note that an operation cannot be taken away nor can an attribute be removed by extending a record.
- We can convert a value from type Circle to type Figure and from type Figure to type Circle, but we must be very explicit about it.

```
P  : Point := (1.0, 0.5);
F1 : Figure := (RefPoint => P);
F2 : Figure;
C1 : Circle := (RefPoint => P, Radius => 5.0);
C2 : Circle;

F2 := Figure(C1); -- F2 effectively ignores the Radius of
                  -- the Circle C1
C2 := (F1 with Radius => 41.2); -- the with clause supplies
                                -- values for extra
                                -- attributes of the record
```

OO Programming in Ada - 7

- *One very important restriction: Any subprogram can only involve one tagged type as either one of its parameters or its return type.* More formally: “A given subprogram shall not be a dispatching operation of two or more distinct tagged types.”
- It is very important to realize that if we declare a variable to be of type `Figure` for example, it will always be a `Figure`. It cannot actually become a `Circle`.
- When the `Area` function is called for a `Figure`, `Area(F : Figure)`, it will always be the `Area` operation that takes a `Figure` as it's first parameter.
- We can't assign a `Circle` to a variable that is defined to be of type `Figure` and then get the `Circle` version of `Area` to be invoked when `Area` is called with that object as a parameter.
- So where is my Dynamic Binding, Dynamic Dispatching, or Polymorphism?

OO Programming in Ada - 8

- Unlike most other OO programming languages, Ada carefully distinguishes between an individual type such as Figure, Circle, Rectangle, or Square and a *set of types such as Figure and all of its derived types.*
- A set of such types is known as a *class.*
- The *class* rooted at the type Figure is denoted as `Figure'Class`.
- The *subclass* of Figure'Class which is rooted at Rectangle is denoted as `Rectangle'Class.`
- If you want a function or procedure to work on any type that is in the set of types known as Figure'Class, you've got to declare the function as such. For example:

```
procedure Print_Area(F : in Figure'Class) is
begin
   Put(“Area = “);
   Put( Area(F) );  -- Calls the correct area-computing
                    -- function based on the Type that F
                    -- actually is at run-time
   New_Line;
end Print_Area;
```

“in” is optional

OO Programming in Ada - 9

```
Print_Area(F1); -- prints the Figure Area
Print_Area(C1); -- prints the Circle Area
```

- Now Dynamic Dispatching based on the actual types of F1, and C1 is happening.
- Notice that thinking in terms of C++ or Java, the object “on which” the procedure Print_Area is being called is in the Ada 95 case just one of the parameters to the procedure call.
- Typically it is the first parameter, but it doesn't have to be.
- If it is the first parameter, and you are using Ada 2005 or Ada 2012 you can move that first parameter to the front of the procedure or function name to get:

```
C2.Print_Area;
F2.Print_Area;
```

- This makes things look a lot more similar to what is the predominant OO programming style. But this moving of the first parameter is not available in Ada 95.

The `Ada.Calendar` Package - 1

- As is true in general for Object-Based or Object-Oriented programming languages, much of the development effort goes into creating Types (with their attributes and operations).
- Previously, we have created our own `Date_Type` with Day, Month, and Year components.
- But there is a better, “built-in”, way to deal with Dates and Times in Ada.
- The `Ada.Calendar` package, which is part of the Ada Standard Library that must be provided by all Ada compilers. (Analogous to the standard C library.)
- In GPS, you can see the package's specification by typing or finding `Ada.Calendar` in an editor window, right clicking on `Calendar`, and selecting “Goto Declaration of Calendar.”
  - This will open an editor window with the actual source code for the `Ada.Calendar` package specification.
- The primary Type made available by the `Ada.Calendar` package is `Time`.
- A Time represents an exact point in time (Year, Month, Day, and Seconds).

The `Ada.Calendar` Package - 2

- You can create an object of type `Time` using the `Time_Of` function (sort of like a constructor in other OO programming languages, but not quite the same because it isn't allocating any memory)

```
function Time_Of (Year    : Year_Number;
                  Month   : Month_Number;
                  Day     : Day_Number;
                  Seconds : Day_Duration := 0.0)
return Time;
```

- Note the Default value supplied for Seconds. You don't have to supply that parameter's value when calling the function. It will assume the value 0.0 which means midnight.
- So it can be used to simply represent a Date.

The `Ada.Calendar` Package - 3

- The Ada.Calendar package also supplies:
  - Functions for getting constituent parts of a Time.

```
function Year (Date : Time) : return Year_Number;
function Month (Date : Time) :
   return Month_Number;
function Day (Date : Time) :
   return Day_Number;
function Seconds (Date : Time) :
   return Day_Duration;
```

  - A function to return the current time `Clock`
  - Comparison functions (<, <=, >, >=)
  - Addition and subtraction functions that work with a given `Time` and a given `Duration`.

Lab 7: Better Contacts - 1

- Goals
  - Improve our Contacts lab by using private types
  - Gain experience using the Time type from the Ada.Calendar package
  - Start writing our code in a more “Ada Way”
- Steps
  1. Create a package specification for a package named `Contacts` (`Contacts.ads`)

     You can start from scratch or start with your `Contacts.ads` from the previous lab.
  2. Define 3 ***private*** types:
     - `Address_Type, Contact_Type, and Contact_Book`.
  3. Address_Type should have the same 4 Unbounded_String attributes (components of the record) as before:
     - `Street, City, State, and Zip`

Lab 7: Better Contacts - 2

- Steps Continued

4. `Contact_Type` should have 3 attributes
   - `Name : Unbounded_String`
   - `Address : Address_Type`
   - `Birthday : Ada.Calendar.Time`
5. `Contact_Book` type should be defined as follows:

```
type Contact_List is array (1 .. 100) of Contact_Type;
type Contact_Book is
   record
      Contacts : Contact_List;
      Num_Contacts : Integer range 0 .. 100 := 0;
   end record;
```

6. `Contact_Book` now not only has an array for storage, but an attribute which is used to keep track of how many contacts we have. But this information is all “hidden” from a user of the type.

Lab 7: Better Contacts - 3

- Steps Continued
  - Note that we cannot do the following:

```
type Contact_Book is
   record
      Contacts : array (1 .. 100) of Contact_Type;
      Num_Contacts : Integer range 0 .. 100 := 0;
   end record;
```

  - This definition of Contacts is referred to as an *anonymous array*
  - It defines an array type without giving that type a name.
  - Anonymous arrays are not allowed as components of records.
  - The solution is to define the array as a type with a name (`Contact_List`) and use that named type.
  - We can define this type in the private part of your package specification.

Lab 7: Better Contacts - 4

- Steps continued
  7. Create a “constructor” for `Address_Type`

```
function Make_Address(Street : in String;
                      City   : in String;
                      State  : in String;
                      Zip    : in String) return Address_Type;
```

Note that the input parameters are `Strings` not `Unbounded_Strings`. You'll have to do some conversions in your implementation of this function.

For testing purposes, `Address_Type` should also have a `Put` procedure which outputs the address information to the console (standard output)

```
function Put(Address : in Address_Type);
```

Lab 7: Better Contacts - 5

- Steps continued
  8. Create similar `Make_Contact` and `Put` operations for `Contact_Type`

```
function Make_Contact(Name     : in String;
                      Address  : in Address_Type;
                      Birthday : in Time) return
Contact_Type;

function Put(Contact : in Contact_Type);
```

Lab 7: Better Contacts - 6

- Steps Continued

9. Create three operations for the `Contact_Book` type

```
function Get_Contact_Count(Contacts : in Contact_Book)
   return Integer;

procedure Add_Contact(Contacts : in out Contact_Book;
                      Contact : in Contact_Type);

function Get_Contact(Contacts : in Contact_Book;
                     Num : in Integer) return Contact_Type;
```

10. Provide implementations for all of these subprograms (functions and procedures) in a separate file (the package body) file named `contacts.adb`.

Lab 7: Better Contacts - 7

- Steps Continued

11. As with the previous Contacts lab, the “main” should be in a separate file (e.g. `testcontacts.adb`) which contains one procedure.
12. Declare

```
My_Contacts : Contact_Book;
Birth_Date : Time;
An_Address : Address_Type;
Contact : Contact_Type;
```

13. Populate: A birth date, an address, and a Contact
    - Use the populated birth date, address, and a name string to populate the Contact
14. Add the Populated Contact to the Contact_Book
15. Repeat the process for at least one other contact with different information
16. Use a for loop to cycle through the contacts and display then using the Put you've defined for the Contact_Type

Lab 7: Better Contacts - 8

- Review a Solution

Section 8

Toward a Standard Object Oriented Ada Format

A standard format for OO Ada Type Definitions

- We didn't use any tagged types, classes, or dynamic dispatching in the previous lab.
- We're going to use those features in the next lab.
- We'll start to follow a “standard” approach for each OO Type definition.

```
with PACKAGE_NAME_OF_PARENT;
use  PACKAGE_NAME_OF_PARENT;
package PACKAGE_NAME is
   type MY_TYPE is tagged private; -- or new Parent_Type with
                                   -- private
   type MY_TYPE_Access is access all MY_TYPE'Access;
     -- We'll take about Access types in a later lesson
   -- primitive operations go here
   package Constructors is
      function Make_MY_TYPE( . . . )
   end Constructors;
private
   -- Define details of MY_TYPE here
end PACKAGE_NAME;
```

- For the package name, it is conventional to use the plural of the type name.
- This standard is not “writ in stone.” But is a good starting place.

What's with the “inner” package Constructors

- All operations on tagged types that are either a predefined operations on the type (e.g. +, -, *, etc. for Integers) or an operation (function, procedure, or operator) that is declared just below the type in the same package specification and has a parameter of that type are called *primitive operations.*
- ***Each primitive operation of a tagged type is inherited by all types derived from the type.***
- If we make “constructors” as we've done so far and allow them to be primitive operations, then the “constructors” for the parent type will be inherited by the child type.
- This is really not what we want.
- By putting the “constructors” (the Make functions) in what is called an “inner package” we prevent them from being inherited by derived types.

Example OO Ada Type Definition

```
with Creatures;
use  Creatures;

package Players is
   -- Player is a type of (extention of) Creature
   type Player is new Creature with private;
   type Player_Access is access all Player;

   -- There is probably a procedure Look(C : in Creature);
   -- defined for the type Creatures.Creature
   procedure Look(P : in Player);
   package Constructors is
      function Make_Player(...) return Player;
private
   type Player is new Creature with
      record
         Logged_In : Boolean;
      end record;
end Players;
```

Bounded Strings - 1

- The `Unbounded_String` type gives the maximum amount of flexibility when using strings.
- But this flexibility comes at the price of lower efficiency and some uncertainty.
- If you are doing programming for embedded systems or for interacting with a database (in which often only a limited number of characters can be stored in a field), you may find that `Unbounded_String` objects become a problem to deal with.
- Somewhere between the rigidity of the `String` type (a fixed size array of characters) and the flexibility of the `Unbounded_String` type, there is the `Bounded_String` type.
- The `Bounded_String` type is defined in the `Ada.Strings.Bounded` package.

Bounded Strings - 2

- But the `Ada.Strings.Bounded` package does not define a simple, single type named `Bounded_String` that we can use.
- Instead it contains a generic package called `Ada.Strings.Bounded.Generic_Bounded_Length`
- The `Generic_Bounded_Length` package is defined such that we must *instantiate* a copy of the package with a specified maximum string length before we can use any of the types and subprograms it defines.
- This is using Ada's Generics mechanism. Like that found in C++ templates or Java Generics.
- Ada's Generics mechanism existed in Ada 83. Before Ada added tagged types in Ada 95.
- Example instantiation of the `Generic_Bounded_Length` package

Bounded Strings - 3

- Example instantiation of the `Generic_Bounded_Length` package

```
with Ada.Strings.Bounded;
package Person is
   type Employee is abstract tagged private;
   procedure Display(Person_In : Employee);
   package String25 is new
      Ada.Strings.Bounded.Generic_Bounded_Length(25);
   use String25;
private
  ...
end Person
```

- Note the creation of a new package which is an instantiation of the `Generic_Bounded_Length` package: `String25`
- Think of this as asking the compiler to write a new package for you based on the `Ada.Strings.Bounded.Generic_Bounded_Length` package. Substituting 25 where ever the maximum length is used in the “generic” or “template” package `Generic_Bounded_Length`.

Bounded Strings - 4

- Now we can use `String25.Bounded_String` (or just `Bounded_String` with a use clause) as a type.Example instantiation of the `Generic_Bounded_Length` package
- The Bounded_String type then has operations like:

```
function To_Bounded_String(Source : String;
                           Drop : Truncation := Error)
                           return Bounded_String;

function To_String (Source : Bounded_String) return String;

function Append(Left : Bounded_String; Right : Bounded_String;
                Drop : Truncation := Error) return Bounded_String;
function Append(Left : Bounded_String; Right : String;
                Drop : Truncation := Error) return Bounded_String;
function Append(Left : String; Right : Bounded_String;
                Drop : Truncation := Error) return Bounded_String;
```

- Also has concatenation &, Index, comparison, find_token, etc. operations
- `type Truncation is (Left, Right, Error)` – serves as an indication of what should happen if the maximum is exceeded by an operation.

Lab 8: OO Contacts - 1

- Goals
  - Improve the design of our Contacts Types by using Bounded Strings
  - Follow standards for file names and type placement
- Steps
  1. Move the Address_Type into its own specification (.ads) and body (.adb) files
     - Name the new package Addresses
     - Use a bounded string with an upper bound of 80 characters for the attributes of the Address_Type
  2. Move the Contact_Type into its own specification and body files
     - Name the new package Contacts
     - Use a bounded string with an upper bound of 80 characters for the Name attribute of the Contact_Type;
  3. Move the Contact_Book type into its own specification and body files
     - Name the new package ContactBooks
  4. Fix up the with and use clauses in the main procedure
     - If you've done the preceding changes correctly, there should be no further changes to the main procedure

Lab 8: OO Contacts - 2

- Review Solution

Section 9

Extending a Tagged Type

Where is our Tagged Type? Where is our Dynamic Dispatcing? - 1

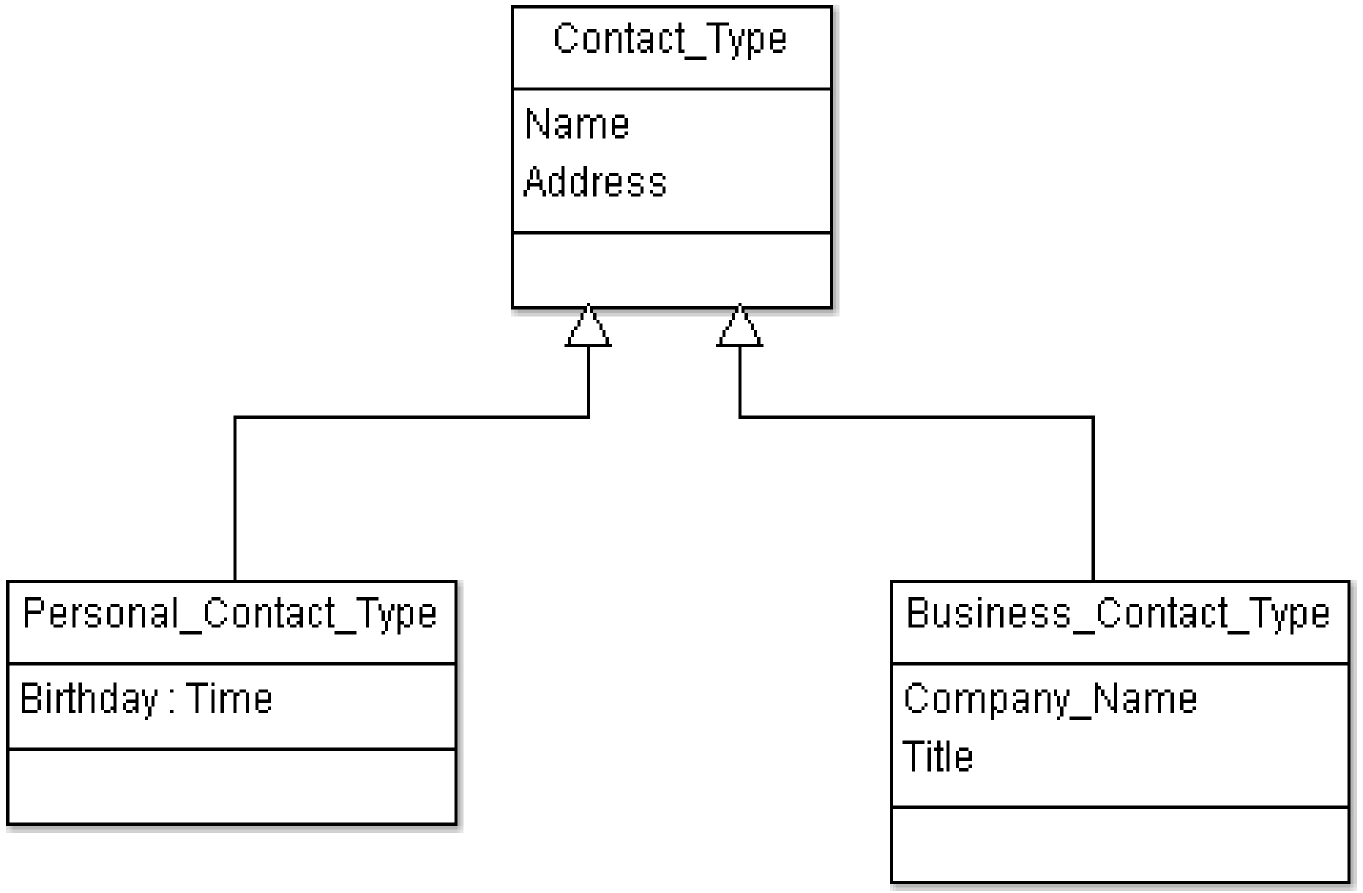
- We *still* haven't used any tagged types
- We *still* haven't used any dynamic dispatching
- We *still* haven't created any types which are extensions of another type
- So...now here we go.

Where is our Tagged Type? Where is our Dynamic Dispatcing? - 2

- Let's assume that we want to have different attributes for “personal” contacts and “business” contacts.
- For personal contacts, we'll want to keep the person's birthday as an attribute.
- For business contacts, we don't really care about the person's birthday. But we do want to have the name of the company the person works for and the person's title.

Our Type Hierarchy

- Here's the UML class diagram



Lab 9: More OO Contacts - 1

- Goals:
  - Make our Contact_Type a tagged type so we can extend it
  - Create two extensions of `Contact_Type`: `Personal_Contact` and `Business_Contact`.
  - Move our “Constructors” to an inner package so they don't get inherited.
- Steps
  1. See our *Figures, Circle, Rectangle, Square* example earlier in the notes for help.
  2. Also see our standard type format for the type Player with it's inner package for Constructors.
  3. Make `Contact_Type` a tagged type so we can extend it
  4. Remove the Birthday attribute from `Contact_Type` and with and use clauses referring to `Ada.Calendar`.

Lab 9: More OO Contacts - 2

- Steps:

5. Move the `Make_Contact` function into an inner package named `Constructors`
   - This will have to happen in both the specification and the body
6. Change any calls to `Make_Contact` in the main procedure to `Constructors.Make_Contact` and remove the BirthDate parameter
7. Run the program to test your changes so far.

Lab 9: More OO Contacts - 3

- Steps:
  8. Create a `Personal_Contact_Type` in a `PersonalContacts` package as an extension of the Contact_Type

     ```
     type Personal_Contact_Type is new Contact_Type with private;
     ```
  9. Add a `BirthDate` attribute for the `Personal_Contact_Type`
  10. Create a `Make_Personal_Contact` procedure in an inner package of `PersonalContacts` named `Constructors`
  11. Create a `Put` function for `Personal_Contact_Type`

  – See next slide for `PersonalContacts` package body

Lab 9: More OO Contacts - 4

- Steps:

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
package body PersonalContacts is
  package body Constructors is
    function Make_Personal_Contact(...)
      return Personal_Contact_Type is
      Contact : Contact_Type;
      Personal_Contact : Personal_Contact_type;
    begin
      Contact := Contacts.Constructors.Make_Contact(Name, Address);
      Personal_Contact := (Contact with BirthDate => BirthDate);
      return Personal_Contact;
    end Make_Personal_Contact;
  end Constructors;

  procedure Put(Personal_Contact : in Personal_Contact_Type) is
    Contact : Contact_Type;
  begin
    Contact := Contact_Type(Personal_Contact);
    Put(Contact);
    Put(Year(Personal_Contact.BirthDate));
    Put(Month(Personal_Contact.BirthDate));
    Put(Day(Personal_Contact.BirthDate));
  end Put;
end PersonalContacts;
```

Do what you have to do to construct something of type Contact_Type.

Then add the BirthDate to it.

Output what you would have for something of type Contact_Type.

Then add the BirthDate info to it.

Lab 9: More OO Contacts - 5

- Steps:

12. Back in your test main procedure, create an object of `Personal_Contact_Type` and use your `Put` function to display its contents.

13. Do not try to add the `Personal_Contact_Type` object to your `Contact_Book` just yet. We're not quite ready for that because our `Contact_Book` can't hold `Personal_Contact_Type` objects.

14. To use the `Contacts.Constructors` package and the `PersonalContacts.Constructors` package, you'll have to invoke the constructors as:

```
Contacts.Constructors.Make_Contact(...)
PersonalContracts.Constructors.Make_Personal_Contact(...)
```

Lab 9: More OO Contacts - 6

- Steps:

15. Back in your Contacts package, create a procedure named Put_Contact
    - It should have a specification like:

```
procedure Put_Contact(C : in Contact_Type'Class);
```

    - Its body should look like:

```
procedure Put_Contact(C : in Contact_Type'Class) is
begin
   Put(C);
end Put_Contact;
```

    - This is a procedure with dynamic dispatching. Notice that it can take any member of the `Contact_Type'Class` as it's parameter and will dispatch to the correct version of Put for the type that is actually passed in.

16. Change all your calls to `Put` in your “main” that you are using to output a contact to calls to `Put_Contact`.
17. Compile and run.

Lab 9: More OO Contacts - 7

- Steps:

18. Finally, create another type which extends `Contact_Type`.
    - This extension will be `Business_Contact_Type` and will be modeled on your `Personal_Contact_Type`.
19. Add `Company_Name` and `Title` attributes to your `Business_Contact_Type`
20. Create a `Business_Contact_Type` object in your "main" and display it using `Put_Contact`
21. Compile and run.

Section 10
Abstract Types and Subprograms

Abstract Types - 1

- Given our Type Hierarchy of:
  - `Contact_Type`
  - `Personal_Contact_Type`
  - `Business_Contact_Type`
- Does it really make any sense to allow the creation of objects of type `Contact_Type`?
- You can argue the point, but let's assume not. `Contact_Type` is intended to be extended, not used directly.
- We can enforce this by making it an *abstract* type.

```
type Contact_Type is abstract tagged private;
```

Abstract Types - 2

- If there are operations on our abstract type that we know we need to have defined for any type derived from the abstract type, but can not or do not want to define the operation for the abstract type, then we can provide the specification of the operation and add the designator *abstract* to the operation
- E.g.

```
type Set is abstract tagged private;
function Empty return Set is abstract;
function Union(Left, Right : Set) return Set is abstract;
function Intersection(Left, Right : Set) return Set is
                                                   abstract;
```

- In fact, any function that returns an abstract type must itself be abstract.

Abstract Types - 3

- Once a type is abstract, do we want to provide a “Constructor” for it?
- No-one would be able to use the “Constructor” anyway.
- But, can we do away with it's functionality?
- We've used it in the Constructors for the extension types.
- We need the functionality of initializing the attributes that are part of this root type.
- ***But any function that returns an abstract type must itself be abstract.***
- We can solve this by turning the “constructor” into a procedure that takes a parameter of type `Contact_Type` to initialize.

Lab 10: Abstract Contacts - 1

- Goal
  - Enforce our notion that objects of type `Contact_Type` should not be declared or created.
  - Turn our “Constructor” for the `Contract_Type` type into an initialization procedure.
- Steps
  - Once our `Contract_Type` is abstract, our `Contact_Book` type will not work. Its fundamental component is an array of things of an abstract type.
  - To make things easy for us for now, remove both the specification and body of the `ContactBooks` package.
  - Close down GPS, and do this outside of GPS. You may want to copy these files somewhere outside of your project for safe-keeping before deleting them.
  - Then restart GPS and remove all uses and references to things in the `ContactBooks` package from your main. The compiler can help you find these.

Lab 10: Abstract Contacts - 2

- Steps
  - In our `Contacts` package specification, declare `Contact_Type` to be abstract
  - Remove the `Contacts.Constructors` inner package and turn its `Make_Contact` **function** into an `Initialize_Contact` **procedure** that takes an **in out** `Contact_Type` parameter along with the two other parameters containing values to use for initialization.

```
procedure Initialize_Contact(C       : in out Contact_Type;
                             Nam     : in String;
                             Address : in Address_Type);
```

  - In the `Contacts` package body, the `Initialize_Contact` procedure will simply assign the values of the two parameters to the appropriate parts of the type.

Lab 10: Abstract Contacts - 3

- Steps
  - The implementation of the `Make_Business_Contact` function will no longer be able to use an object of type `Contact_Type` or call a function named `Make_Contact_Type`.
  - Replace the call to `Make_Contact_Type` with a call to `Initialize_Contact` which will initialize the attributes that a `Business_Contact_Type` inherits from a `Contact_Type`.
  - Then initialize the attributes that are added by the `Business_Contact_Type`. For example:

```
function Make_Business_Contact(Name : in String;...)
  return Business_Contact_Type is
  Business_Contact : Business_Contact_Type;
begin
   Initialize_Contact(Business_Contact, Name, Address);
   Business_Contact.Business_Name :=
     To_Bounded_String(Business_Name);
   usiness_Contact.Title := To_Bounded_String(Title);
   return Business_Contact;
end Make_BusinessContact;
```

Lab 10: Abstract Contacts - 4

- Steps
  - The implementation of the `Put` function in the `BusinessContacts` package will no longer be able to use an object of type `Contact_Type`.
  - The new implementation of the Put function should look like:

```
procedure Put(Business_Contact : Business_Contact_Type) is
begin
   Contacts.Put(Contact_Type(Business_Contact));
   Put_Line(To_String(Business_Contact.Business_Name));
   Put_Line(To_String(Business_Contact.Title));
end Put;
```

Lab 10: Abstract Contacts - 5

- Steps
  - The changes to the `Make_Personal_Contact` function and the `Put` procedure in the `PersonalContacts` package will be very similar to those you just made to in the `BusinessContacts` package
  - Your main procedure should now create at least one of each of your contact types and put them to the display using the dynamic dispatching version of `Put_Contact`.
  - Compile, run, and verify.

Lab 10: Abstract Contacts - 6

- Steps
  - Finally, let's do one last "clean-up" step for this lab.
  - Some consider the inner `Constructors` packages to be a particularly "messy" way to deal with initialization.
  - Our changes to the `Contacts` package so far in this lab suggest an alternative.
  - Instead of `Make_`*`my_type_name`* **functions** thought of as "constructors" use `Initialize_`*`my_type_name`* **procedures**
  - They will be relatively easy to write, can exist for abstract tagged types, and are easy to call in extensions of those abstract tagged types.

Lab 10: Abstract Contacts - 7

- Steps
  - So the entire inner package `PersonalContacts.Constructors` can be replaced with one `Initialize_Personal_Contact` procedure that looks like:

```
procedure Initialize_Personal_Contact(
   C : in out Personal_Contact_Type;
   Name      : in String;
   Address   : in Address_Type;
   BirthDate : in Time) is
begin
   Initialize_Contact(C, Name, Address);
   C.BirthDate := BirthDate;
end Initialize_Personal_Contact;
```

Lab 10: Abstract Contacts - 8

- Steps
  - And the entire inner package `BusinessContacts.Constructors` can be replaced with one `Initialize_Business_Contact` procedure that looks like:

```
procedure Initialize_Business_Contact(
   C : in out Business_Contact_Type;
   Name          : in String;
   Address       : in Address_Type;
   Business_Name : in String;
   Title         : in String) is
begin
   Initialize_Contact(C, Name, Address);
   C.Business_Name := Business_Name;
   C.Title         := Title;
end Initialize_Business_Contact;
```

Section 11

A Bit More Control
and
Text File I/O

User-Controlled Initialization, Finalization, & Assignment - 1

- There are three distinct fundamental activities that occur to all objects as a program is running.
  - Initialization – carried out immediately after creation
  - Finalization – carried out immediately before destruction
  - Assignment – when one object's value is assigned to another object
- If you want more control over these operations for a type that you create, you can make your type an extension of either the type `Controlled` or the type `Limited_Controlled`.
- These types are in the `Ada.Finalization` package.
- The type `Controlled` gives you “hooks” into all three operations.
- The type `Limited_Controlled` gives you “hooks” into only Initialization and Finalization

User-Controlled Initialization, Finalization, & Assignment - 2

```
type Controlled is abstract tagged private;
procedure Initialize(Object : in out Controlled);
procedure Adjust    (Object : in out Controlled);
procedure Finalize  (Object : in out Controlled);

type Limited_Controlled is abstract tagged limited private;
procedure Initialize(Object : in out Limited_Controlled);
procedure Finalize  (Object : in out Limited_Controlled);
```

- Once you've created your own type that is an extension of one of these types, you can override:
  - Initialize – will be called just after something of your type is declared and any default initialization occurs
  - Finalize – will be called just before an object is destroyed (e.g. goes out of scope)
  - Adjust – will be called just after an object is copied from one variable to another (assigned)
  - Gotcha – Limited_Controlled is `limited`.
    - Thus it does not get an assignment operator and cannot be copied/assigned

User-Controlled Initialization, Finalization, & Assignment - 3

- Suppose type `T` is an extension of `Controlled` and consider the following code

```
procedure Do_This is
  A : T;      -- Create A in memory, Initialize(A)
begin
  A := E;     -- Finalize(A), copy value, Adjust(A)
  ...
end Do_This; -- Finalize(A)
```

- Notice that when A is declared it takes up memory and is initialized
- When you assign a new value to A, the current value that A is being overwritten (destroyed) so Finalize is called before it is overwritten, then the new value is copied, then we get a call to Adjust to give us the opportunity to “adjust” or “change” A before the code moves on to whatever is next
- When the procedure reaches its end, the variable A is going “out of scope” (i.e. going away). So we get a call to Finalize to allow us to “clean up” before the space is actually freed up
- You do not have to write code that explicitly calls Initialize, Finalize, or Adjust

Character Types

- Ada provides a type for holding and manipulating a single “element of text”, a `Character`.
- Something of type Character can only hold one of 256 possible characters (a superset of ASCII called Latin-1)
- The Latin-1 character set is sufficient for handling most languages that are written using “Latin-based” characters (English, French, Spanish, etc.)
- Ada provides another type for holding and manipulating a single “element of text”, a `Wide_Character`.
- Wide_Character is helpful if you need to handle non-Latin alphabets (Chinese, Arabic, etc.)
- Character constants are written surrounded by single quotes: `'Y'`
- The package `Ada.Characters.Handling` contains a large set of operations to help you manipulate objects of type `Character`
- The package Ada.Wide_Characters.Handling contains similar operations for objects of type `Wide_Character`

Ada.Characters.Handling Examples

```
function Is_Control(Item : in Character) return Boolean;
function Is_Letter(Item : in Character) return Boolean;
function Is_Lower(Item : in Character) return Boolean;
function Is_Upper(Item : in Character) return Boolean;
function Is_Digit(Item : in Character) return Boolean;
function Is_Decimal_Digit(Item : in Character) return
                         Boolean renames Is_Digit;
function To_Lower(Item : in Character) return Character;
function To_Upper(Item : in Character) return Character;
function To_Lower(Item : in String) return String;
function To_Upper(Item : in String) return String;

function Is_Character(Item : in Wide_Character)
                      return Boolean;
function To_Character(Item : in Wide_Character;
                      Substitute : in Character := ' ')
                      return Character;
```

A Little More about Strings - 1

- We've already seen that Strings are fixed length, one-dimensional arrays of Characters.
- We've also seen String objects passed to subprograms
- Ada *attributes* can be used to retrieve properties of objects
- The `'Size` attribute can be used to get the size of an object ***in bits***

```
Str : String := “Hello World”;
Num_Bits : Integer := Str'Size;
```

- The `'Length` attribute can be used to get the *character length* of a String

```
Str : String := “Hello World”;
Num_Chars : Integer := Str'Length;
```

- The `'First` and `'Last` attributes will give you the first and last acceptable index in any array.

A Little More about Strings - 2

- Consider the following code

```
with Ada.Text_IO; use Ada.Text_IO;
procedure ReverseString is

   procedure Print_Reverse( S : in String ) is
   begin
      for I in reverse 1 .. S'Length loop
         Put(S(I));
      end loop;
      New_Line;
   end Print_Reverse;

   Demo : String := “A test”;

begin
  Print_Reverse(Demo);
end ReverseString;
```

- Will this work to print the reverse of “A test”?
- But … do you see anything that might be wrong with it?

A Little More about Strings - 3

- Consider

```
with Ada.Text_IO; use Ada.Text_IO;
procedure ReverseString is
   procedure Print_Reverse( S : in String ) is
   begin
      for I in reverse 1 .. S'Length loop
         Put(S(I));
      end loop;
      New_Line;
   end Print_Reverse;
   Demo : String(4 .. 9) := “A test”;
begin
  Print_Reverse(Demo);
end ReverseString;
```

- This will raise a CONSTRAINT_ERROR !
- You do not have any guarantee that a String you are passed will have an index that starts with 1.

A Little More about Strings - 4

- So the right way to do this is to use the attributes 'First and 'Last

```
with Ada.Text_IO; use Ada.Text_IO;
procedure ReverseString is
   procedure Print_Reverse( S : in String ) is
   begin
      for I in reverse S'First .. S'Last loop
         Put(S(I));
      end loop;
      New_Line;
   end Print_Reverse;
   Demo : String(4 .. 9) := “A test”;
begin
  Print_Reverse(Demo);
end ReverseString;
```

Simple Text File I/O

- We've used Ada.Text_IO to write to and read from the “console”.
- Now we'll use it to write to and read from text files.
- Operating System files are represented by the type `Ada.Text_IO.File_Type`
- Before a text file can be read from or written to it must be opened or created.

```
procedure Create(File : in out File_Type;
                 Mode : in File_Mode := Out_File;
                 Name : in String := “”;
                 Form : in String := “”);
procedure Open(File : in out File_Type;
               Mode : in File_Mode;
               Name : in String;
               Form : in String := “”);
procedure Close(File : in out File_Type);
```

- Mode can be In_File, Out_File, or Append_File
- Form is optional and operating system specific
- All the `Get*` and `Put*` operations can take a parameter of type File_Type as their first parameter.

Simple Text File I/O Example

```
with Ada.Text_IO; use Ada.Text_IO;
procedure SimpleFileOutput is
   procedure Print_Reverse( File_Name : in String; S : in String ) is
      New_File : File_Type;
   begin
      Create(New_File, Out_File, File_Name);
      for I in reverse S'First .. S'Last loop
         Put(New_File, S(I));
      end loop;
      New_Line(New_File);
      Close(New_File);
   end Print_Reverse;
   Demo : String(4 .. 9) := "A test";
begin
   Print_Reverse("MyNewFile.txt", Demo);
end SimpleFileOutput;
```

Line Endings & File Endings

- Different operating systems use different characters or sequences of characters to indicate the end of line and the end of file. `Ada.Text_IO` provides cross platform operations that work with whatever is appropriate for your OS

```
procedure New_Line(File : in File_Type;
                   Spacing : in Positive_Count := 1);
procedure New_Line(Spacing : in Positive_Count := 1);

-- move to the next line to read
procedure Skip_Line(File : in File_Type;
                    Spacing : in Positive_Count := 1);
procedure Skip_Line(Spacing : in Positive_Count := 1);

-- return True if input is at the end of a line
function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;

-- return True if input is at the end of the file
function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;

- - Tell me what line I'm on
function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;
```

Access to the 3 standard I/O locations

- Various Operating systems and programming languages support the notion of the Standard Output (a.k.a. stdout), Standard Input (a.k.a. stdin), and Standard Error (a.k.a. stderr) I/O locations.
- Ada.Text_IO has three functions that give you access to these “Files”
  - Current_Input
  - Current_Output
  - Current_Error
- Ada.Text_IO has three procedures that allow you to change the locations of the Standard I/O locations
  - Set_Input
  - Set_Output
  - Set_Error
- To send all error messages to a file named “Error.txt”

```
Error_File : File_Type;
Create(Error_File, Out_File, “Error.txt”);
Set_Error(Error_File);
```

Access to command line parameters and exit codes

- The Ada.Command_Line package provides access to command line positional parameters and the ability to set the exit status code for your program

```
package Ada.Command_Line is
   function Argument_Count return Natural;
   function Argument(Number : in Positive) return String;
   function Command_Name return String;
   type Exit_Status is implementation specific integer type;
   Success : constant Exit_Status;
   Failure : constant Exit_Status;
   procedure Set_Exit_Status (Code : in Exit_Status);
end Ada.Command_Line;

if Argument_Count > 0 then
   for Arg in 1 .. Argument_Count loop
      Put_Line(Argument(Arg));
   end loop;
   Set_Exit_Status(Success);
else
   Set_Exit_Status(Failure);
end if;
```

Lab 11: Simple File I/O - 1

- Goal
  - Experiment with extending the `Ada.Finalization.Controlled` type
  - Send simple output from your program to a file
- Steps
  - First make a copy of your code for Lab 10 somewhere for safe-keeping. You'll need it as a starting point for Lab 12.
  - Make your `Address_Type` an extension of `Ada.Finalization.Controlled`

```
type Address_Type is new Controlled with private; -- in spec

type Address_Type is new Controlled with          -- in private
   record . . .
```

  - Override the procedures `Initialize`, `Finalize`, and `Adjust`
  - Build and run your application
  - See if you can identify each point in your main procedure where one of your Initialize, Finalize, and Adjust procedures are being called
  - Maybe add some other Put calls in your code to help you with this.

Lab 11: Simple File I/O - 2

- Steps
  - Send all or some of your output to a newly created file named `AddressOutput.txt`
  - Note that in our slides we saw two ways we might consider doing this.
  - We could change all the `Put*` calls to use a `File_Type` parameter, including the `Put` procedures in all our types, and create versions of all our Put procedures that take a `File_Type` parameter.
  - But we might consider just redirecting standard output to our file using a call to `Set_Output`
  - However, trying to do this lab by simply redirecting the standard output may result in a PROGRAM_ERROR exception at run time with a message that says “finalize/adjust raise exception”
  - This will happen if you are a “good citizen” and remember to put a call to Close at the end of your main procedure to close your output file.
  - This is because you've closed the output file, then some of your Address_Type variables will be destroyed when you hit the end of your procedure. These objects will try to write to standard output which is a closed file.

Lab 11: Simple File I/O - 3

- Steps (Optional)
  - Modify your main procedure so that it takes the name of the file to create as a command line parameter.
  - After you've made this change to your code and compiled it, the easiest way to run your program and pass it a command line parameter is to use the OS Shell console feature within GPS
  - Select Tools → Consoles → OS Shell
  - Then run your program within the Shell like:

```
> testcontacts MyFile.txt
```

Section 12
Access Types

Declaring Access Types

- Access types are the Ada form of pointers (C/C++) or references (Java). They “point to” or “hold a reference” to an object of a specified type.
- Declaring access variables is generally done by first defining a type.
- If the type you are declaring is to be an access to an object of type `T`, then by convention the name of the type you define should be `T_Access`.
- For example, if you have defined a type `Node` and need to define a type which is an access to a `Node`, you would write:

```
type Node_Access is access Node;
```

- Think: `Node_Access` is a pointer to a Node.

Access Type Rules

- To make access types safer than general pointers, Ada establishes some rules with regard to access types.
  - There is a special access type value, `null`, which represents the case that the access variable doesn't refer to an object. You can compare an access type value to `null` using `=` or `/=`
  - All access type variables are automatically initialized to `null` unless you have explicitly initialized them to something else
  - All operations that “dereference” an access value (follow the pointer) first check to see if the access value is `null`. If it is, they raise a `Constraint_Error`.
  - Normally, access types can only refer to objects that have been created *dynamically* (i.e. created using the new keyword we'll see in a moment, existing on the *heap* instead of the *stack*). If you want an access type to be able to reference locally declared objects, you need to add the keyword `all` to your type declaration.

    ```
    type Integer_Access is access all Integer;
    ```
  - Arithmetic on access variables (i.e. “pointer arithmetic”) is generally not permitted. (You can jump through some hoops to make this happen if you absolutely have to. But it is ***strongly*** discouraged.)

Access Variables and Unbounded Types - 1

- If we want to store a collection of objects but we don't know ahead of time the maximum number of objects in the collection we cannot use an Array for storage. A type that can store varying amounts of objects with no fixed limit is referred to as an *Unbounded Type*.
- Unbounded Types are typically created using access types (pointers)
- The classic example of this is a singly linked list
- To create a singly linked list of Integers for example, you would define a “node” to hold both one of your list values (an Integer) and an access to the next “node” in the list.

```
type List_Node is
   record
      Data : Integer           -- the value “at” this node
      Next : List_Node_Access; -- ref to next node in list
   end record;
```

- How would we define List_Node_Access?

```
type List_Node_Access is access List_Node;
```

Access Variables and Unbounded Types - 2

- But this won't work!

```
type List_Node is
   record
      Data : Integer            -- the value “at” this node
      Next : List_Node_Access; -- ref to next node in list
   end record;

type List_Node_Access is access List_Node;
```

- Because we are using `List_Node_Access` before it is defined.
- Reversing the order won't work either, because then we would be using `List_Node` before it is defined.
- To solve this “Chicken and Egg” problem we have to use something called an *incomplete type declaration*

Access Variables and Unbounded Types - 3

- The standard declaration of a `List_Node` and `List_Node_Access` using an incomplete type declaration looks like:

```
type List_Node; -- the incomplete type declaration telling
                -- the compiler you'll finish it later

type List_Node_Access is access List_Node;

type List_Node is
   record
      Data : Integer           -- the value “at” this node
      Next : List_Node_Access; -- ref to next node in list
   end record;
```

- Now we can declare variables of our access type

```
A_Node : List_Access;
```

Access Value Operations - 1

- Of course you need to be able to
  - create a new List_Node object and get an access (pointer) to it
  - assign one List_Node access value to a variable
  - get access to the List_Node that is referenced by an access value.
- The `new` operator creates an object in what Ada refers to as the *general storage area.* This is what you might refer to as the heap. The salient feature of things allocated on the heap is that they don't go away because you leave the enclosing block in which they were declared (as is true of other declared variables).

```
A_Node := new List_Node;
```

Access Value Operations - 2

- The standard “dot” notation is used to get access to a component of the record. This looks just as if the variable were not an access type at all.

```
Val : Integer; B_Node : List_Node_Access;
A_Node.Data := 1;
Val := A_Node.Data;
A_Node.Next := B_Node;
```

- Sometimes you don't want to retrieve or set just one of the attributes of the object being referenced. You want the entire object. Then you use the *psuedo-component* `all`.

```
Found_Node : List_Node;
-- the following is not assigning a pointer
-- it is assigning/copying the entire node pointed to by A_Node
Found_Node := A_Node.all;
```

- This is often necessary when you have an access value and need to pass it to a subprogram that takes the type being referenced

```
A_Node : List_Node_Access := . . . ;
procedure My_Procedure(Input : in List_Node);
My_Procedure(A_Node); -- won't compile; won't work
My_Procedure(A_Node.all); -- proper call
```

Passing Access Values as Parameters

- To do any significant Object-Oriented programming you will want to pass around access type values.
- Passing them by declaring parameters to be of your access type is allowed, but doesn't enable the necessary polymorphism/dynamic dispatching that you might think it would.
- A procedure defined like:

  ```
  procedure Do_This(Contact : in Contact_Access_Type);
  ```

  will always only take a parameter of type `Contact_Access_Type`. Pointers to things of type `Contact_Type`.
- It won't allow you to pass in access values that “point to” `Personal_Contact_Type` or `Business_Contact_Type`.
- Ada is picky that way.

Another Way to Pass Access Parameters

- Instead of supplying a parameter mode of *in, in out,* or *out* to a parameter, a parameter can be given the mode *access*

```
procedure Do_This( Contact : access Contact_Type );
```

- This means that the `Do_This` procedure takes an access to a `Contact_Type` object as its parameter value.
- Now, as long as `Contact_Type` is a tagged type, `Do_This` is an overridable operation using `Contact_Type`. Extension types (e.g. `Personal_Contact_Type` and `Business_Contact_Type`) can override this operation.

```
procedure Do_This( Contact : access
Personal_Contact_Type );
procedure Do_This( Contact : access
Business_Contact_Type );
```

- When Do_This is called, the correct version of Do_This will be called based upon the actual type of the Contact parameter passed in. We've got our *polymorphism*, our *dynamic binding*, our *dynamic dispatching*.

And Finally...Another Way to Pass Access Parameters

- We could also write a subprogram with an access mode parameter where that parameter is a class.
- Remember: In Ada a Class is a very specific thing – a set of types rooted at the specified type

```
procedure Do_That( Contact : access Contact_Type'Class );
```

- This means that the `Do_That` can take as its parameter an access type to any object that is of type `Contact_Type` or any of its descendents.
- ***BUT … there will be no dynamic dispatching going on here***.
- Whether you call `Do_That` and pass it an access to a `Contact_Type`, an access to a `Personal_Contact_Type`, or an access to a `Business_Contact_Type`, it's always the same `Do_That` procedure that will be called.
- Recall our procedure

Calls not dynamically dispatched.

```
procedure Put_Contact(C : in Contact_Type'Class) is
begin
   Put(C);
end Put_Contact;
```

This call is dynamically dispatched.

Freeing Memory Allocated with `new`

- Ada was designed to *permit but not require* automatic garbage collection
- Most Ada compilers do not perform automatic garbage collection. (There are Ada compilers that compile into Java bytecode and thus the programs are executed in a JVM which does have automatic garbage collection. But, in general, you should not expect it.)
- Ada provides you with a generic procedure named Unchecked_Deallocation for freeing memory from objects that you know you don't need anymore.
- We'll talk more about Ada Generics later. For now, simply understand that you must define (or instantiate) a procedure based on Unchecked_Deallocation that will be used to free the memory used by types that you define. By convention the name of the procedure you create this way is `Free`.
- The syntax for that is:

```
procedure Free is
  new Unchecked_Deallocation(List_Node, List_Node_Access);
```

- With Free defined this way, you can call Free like:

```
Free(A_List_Node_Access);
```

- After `Free` returns, the memory will have been released, and `A_List_Node_Access` will have a value of `null`.

Creating a Type that can access any Type in a Class

- There may be times when you want to define an access type which can reference any type that is a member of a class
- An access type that can reference any type that is a member of the class Contact_Type'Class would be defined as:

```
type Any_Contact_Access is
   access all Contact_Type'Class;
```

Run-Time Type Identification (RTTI)

- There may be times when you need to “query the tag” of an object at run time. That is, to determine which type the object belongs to. This, of course, only works for tagged types.
- You can determine whether an object belongs to a certain class of types, or to a specific type, by means of the membership test, `in`

```
procedure Do_Something(C : Contact_Type'Class) is
begin
   if C in Personal_Contact_Type then
      . . .
   elsif C in Personal_Contact_Type'Class then
      . . .
   end if;
end Do_Something;
```

Lab 12: A Flexible ContactBook - 1

- Goals
  - Create a Contact Book that can hold both our types of contacts (Personal_Contact_Type and Business_Contact_Type)
- Notes
  - We won't concern ourselves with making our Contact Book hold an infinite number of contacts. We'll accept that we can define a maximum number of Contacts
- Steps
  - Start with your work from Lab 10
  - If you haven't already done so, create access types for all types in the class `Contact_Type'Class`.
  - You should have previously removed the `ContactBooks` package
  - We're going to create a new `ContactBooks` package
    - `contactbooks.ads`
    - `contactbooks.adb`

Lab 12: A Flexible ContactBook - 2

- Steps
  - Create a Contact_Book_Type in your ContactBooks package specification
  - Make it a private type
  - Specify that it will have two procedures
    - One to add a contact to the book that allows the contact to be of any type in the class Contact_Type'Class
    - One to display all the contacts in the Contact Book
  - See the following slide for the ContactBooks package specification.

Lab 12: A Flexible ContactBook - 3

Notice that the use of “pointers” is hidden from the client code that uses our Contact_Book_Type.

```
with Contacts; use Contacts;

package ContactBooks is
   type Contact_Book_Type is private;
   procedure Add_Contact(Contacts : in out Contact_Book_Type;
                         Contact : in Contact_Type'Class);
   procedure Put_All(Contacts : in Contact_Book_Type);

private
   type Any_Contact_Access is access all Contact_Type'Class;
   type Contact_List is array (1 .. 100)
      of Any_Contact_Access;

   type Contact_Book_Type is
      record
         Contacts : Contact_List;
         Num_Contacts : Integer range 0 .. 100 := 0;
      end record;
end ContactBooks;
```

An array of “pointer” to any type derived from Contact_Type.

Lab 12: A Flexible ContactBook - 4

- Steps
  - Write the body
  - In the `Add_Contact` procedure, you will have to use RTTI to determine which type of contact you've been passed to add to your contact book.
  - You'll then have to use new to allocate memory and create an appropriate type of contact in which to copy the information from the contact you've been passed.
  - If your newly created pointer to a `Personal_Contact_Type` is stored in a variable named PC, and the Contact you've been passed is a `Personal_Contact_Type`, your assignment of the passed in values to the `Personal_Contact_Type` pointed to by PC will look like:

```
PC.all := Personal_Contact_Type(Contact);
```

  - Store a “pointer” to the created contact in the next available slot in your array of pointers.
  - Don't forget to appropriately increment your counter of contacts.

Lab 12: A Flexible ContactBook - 5

- Steps
  - It is in the Put_All procedure that dynamic dispatching will occur.
    - Loop through your array of pointers
    - Dereference each pointer to get the actual contact being pointed to (use `.all`)
    - Call your dynamically dispatched Put operation to perform a type specific put of the object
  - Change your “main” procedure to
    - Declare a `Contact_Book_Type` object
    - Use `Add_Contact` to add contacts to the contact book
    - Use `Put_All` to display the contents of the contact book
  - Compile and test run

Section 13
Exceptions

Exception Basics - 1

- Like most modern programming languages, Ada provides a mechanism for dealing with errors and other exceptional situations.
- When such situations arise an Exception is said to be *raised*. We've seen cases where Constraint_Error would have been raised.
- There are 4 predefined language exceptions: `Constraint_Error`, `Program_Error`, `Storage_Error`, and `Tasking_Error` (Ada 83 also had `Numeric_Error`, but Ada95 redefined `Numeric_Error` to be the same as `Constraint_Error`)
- Many packages that are part of the Ada environment define their own exceptions
  - For example Ada.Text_IO defines Status_Error, Mode_Error, Name_Error, End_Error, etc.
  - These are not part of the Ada Language, but part of the standard set of packages required

Exception Basics - 2

- The default action when an exception is raised is to halt the program.
- Depending upon your compiler and the compiler switches used, you might get a print out of the name of the exception and some indication of where in the code the problem occurred.
- You can change this default behavior (i.e. handle an exception yourself)
- You can also define your own exceptions and raise and handle them yourself.
- Generally, you should not use exceptions to handle expected conditions.

Declaring and Raising Exceptions

- To declare an exception is like declaring a variable of type exception

```
The_World_Is_Falling_Apart : exception;
```

- In Ada, exceptions are not part of the type hierarchy.
  - You cannot extend the exception type
  - Exceptions cannot be parameters passed to subprograms
  - Exceptions cannot be components of records
- Exceptions are raised either by the failure of a language-defined check (e.g. indexing an array beyond its bounds) or by the execution of a raise statement.

```
raise The_World_Is_Falling_Apart;
```

Handling Exceptions

- When and exception is raise, Ada will abandon what it was doing and look for a matching exception handler in the sequence of statements where the exception was raised.
- If no matching handler is found, it returns from the current subprogram and tries to find a matching exception handler in the calling subprogram, and so on up the call tree until it finds a matching exception handler or exits the entire program.
- Example exception handler

```
procedure Open_Or_Create(File : in out File_Type;
                         Mode : in File_Mode;
                         Name : in String) is
begin
   Open(File, Mode, Name);
exception
   when Name_Error =>
      Create(File, Mode, Name);
   when others =>
      Put_Line(“something went wrong”);
      raise;
end Open_Or_Create;
```

others means any exception not listed

raise without an exception name re-raises the exception currently being handled

Lab 13: Too Many Contacts - 1

- Goals
  - Define your own Too_Many_Contacts exception to be raise if the user of your Contact Book tries to add a Contact to an already full book
  - Raise your Too_Many_Contacts exception when appropriate
  - Handle the Too_Many_Contacts exception in your main subprogram
- Steps
  - In the public part of your `ContactBooks` package, declare an exception named `Too_Many_Contacts`.
  - In the private part of your `ContactBooks` package, define a constant named `Maximum_Contacts` and use it in place of the numeric maximum values in the declarations.

    ```
    Maximum_Contacts : constant Integer := 100;
    ```
  - In the implementation of your `Add_Contact` procedure, raise the `Too_Many_Contacts` exception if the user tries to add more than the maximum allowed contacts

Lab 13: Too Many Contacts - 2

- Steps
  - Add an exception section to the bottom of your main procedure and handle the `Too_Many_Contacts` exception by writing out a message and re-raising the exception.
  - Change the Maximum_Contacts value to something pretty low (like 2) and then try to add a contact beyond that upper limit
  - Compile and test run

Section 14
Generics

What is a Generic?

- It is often useful to create a more generic version of a subprogram or package and then use that generic version to create more specific subprograms or packages for actual use.
- You write the logic independently of the type of object the logic is to work with. That is, create a *template* of the logic that will be used.
- In C++, this capability is called *templates*. In Ada it is called *generics.*
- Consider the following example procedure

```
-- declaration
procedure Swap(Left, Right : in out Integer);
-- body
procedure Swap(Left, Right : in out Integer) is
  Temp : Integer;
begin
  Temp  := Left;
  Left  := Right;
  Right := Temp;
end Swap;
```

- This is fine, but it will only let you swap Integers. Whereas the logic would be fine for just about any type.

A Generic Swap procedure

- A generic version of Swap would look like

```
-- declaration
generic
  type Element_Type is private;
procedure Generic_Swap(Left, Right : in out Element_Type);
-- body
procedure Generic_Swap(Left, Right : in out Element_Type) is
  Temp : Element_Type;
begin
  Temp  := Left;
  Left  := Right;
  Right := Temp;
end Generic_Swap;
```

- Notice that we've listed the generic item of our algorithm after the keyword generic. The items in this list are called the *generic formal parameters*.
- This code cannot be used “as is” because the `Generic_Swap` procedure does not have an actual type to work with.

Using a Generic procedure

- To use a generic procedure, we must ask the compiler to use this template that we've provided to essentially write a real procedure substituting a real type for the generic type.
- This process is called *instantiation* and the new procedure created is called an *instance* of our generic procedure.

```
procedure Swap is new Generic_Swap(Integer);
procedure Swap is new Generic_Swap(Float);
procedure Swap is new Generic_Swap(Unbounded_String);
```

- We have asked the compiler to write and compile for us 3 new Swap procedures.
- Now we can use them

```
A, B : Integer; A := 5; B := 7;
Swap(A, B);
```

Specifying Generic Formal Parameters

- What can be used as a generic formal parameter (the list right after the keyword `generic`)?
  - Values or variables of any type (called *formal objects*)
    - `Maximum_Size : Integer`
  - Any type (called *formal types*)
    - `type Element_Type is private;`
  - Other instances of generic packages (called *formal packages*)
    - allow one generic package to instantiate another
- Formal type specifications allow you to not only name the type (as we did with Element_Type), but to also specify what “kind of type” is permitted.

Formal Type Specifications

```
-- any type that has assignment (:=) and equal-to (=)
type type_name is private;

-- any type at all
type type_name is limited private;

-- any tagged type with assignment (:=)
type type_name is tagged private;

-- any tagged type
type type_name is tagged limited private;

-- any “discrete” type (e.g. an enumeration, Integer, Boolean)
type type_name is (<>);
```

Example Generic Package Specification

- Generic packages are produced and used more often than stand-alone generic procedures. One of the best ways to understand generics is to see a full fledged generic package. The following is a fairly simple one borrowed directly from the Ada 95 Reference Manual (RM) section 12.8.
- Here is the Generic Package Specification

```
generic
   Size : Positive;
   type Item is private;
package Generic_Stack is
   procedure Push(E : in  Item);
   procedure Pop (E : out Item);
   Overflow, Underflow : exception;
end Generic_Stack
```

Example Generic Package Body

```
package body Generic_Stack is
   type Table is array (Positive range <>) of Item;
   Space : Table(1 .. Size);
   Index : Natural := 0;

   procedure Push(E : in Item) is
   begin
      if Index >= Size then
         raise Overflow;
      end if;
      Index := Index + 1;
      Space(Index) := E;
   end Push;

   procedure Pop(E : out Item) is
   begin
      if Index = 0 then
         raise Underflow;
      end if;
      E := Space(Index);
      Index := Index - 1;
   end Pop;
end Generic_Stack;
```

Instantiating Our Generic Package

- To use our Generic_Stack package we must instantiate it.

```
package Stack_Int is new Generic_Stack(Size => 200,
                                       Item => Integer);
```

- Now we can use our new Stack_Int like

```
Stack_Int.Push(7);
```

- Recall that we already saw this when we were using bounded strings.

A Few Details to Note About Our Generic Package Body

- Positive is a standard predefined subtype of Integer
  `subtype Positive is Integer range 1 .. Integer'Last;`
- Natural is a standard predefined subtype of Integer
  `subtype Natural is Integer range 0 .. Integer'Last;`
- Using `Size => 200` and `Item => Integer` is an example of parameter passing by name instead of position which sometimes seems verbose, but sometimes makes things much clearer.
- Valid alternatives include:

```
package Stack_Int is new Generic_Stack(200, Integer);
package Stack_Int is new Generic_Stack(Item => Integer;
                                       Size => 200);
```

- Passing by name is allowed for subprograms too!

Generic Data Object vs Generic Data Type

- Our `Generic_Stack` package allows us to instantiate and use a number of different types of stacks.
- But it doesn't allow us to declare variables of a stack type that can be used, passed around as parameters to subprograms, or declared to be part of other types (a record that contains a stack for example).
- Our Generic_Stack is said to be a *Generic Data Object*.
- We can make things more flexible by instead creating a *Generic Data Type*.

Generic Data Type Specification

```
generic
   Size : Positive;
   type Item is private;
package Generic_Stack is
   type Stack is limited private;
   procedure Push(S : in out Stack;
                  E : in Item);
   procedure Pop (S : in out Stack;
                  E : in Item);
   Overflow, Underflow : exception;
private
   type Stack is array(1 .. Size) of Item;
end Generic_Stack;
```

- Note that in this package we've created a type that is dependent upon the generic formal parameters. Once this package is instantiated, we'll have a type that we can actually use to declare variables.
- This is called a *Generic Data Type*

Using a Generic Data Type

- To use the Generic Data Type version of `Generic_Stack`, we instantiate it as before, and then create variables of the new type

```
package Stack_Int is
   new Generic_Stack(Size => 200,
                     Item => Integer);

use Stack_Int;
. . .
My_Stack : Stack; -- Stack_Int's Stack type
. . .
Push(My_Stack, 7);
```

Lab 14: Generic Contact Book

- Goal
  - Create a generic version of your `Contact_Book_Type` for which the maximum number of contacts that can be stored in the book varies from instantiation to instantiation
- Steps
  - Add a generic formal parameter list to the top of your `ContactBooks` package specification.

    ```
    Generic
       Maximum_Size : Positive;
    ```
  - Remove any previously used maximum size constant from both the specification and the body and replace them with the generic parameter `Maximum_Size`
  - Rename your `ContactBooks` package to `Generic_ContactBooks`. Don't forget to do this in both the specification and the body. You'll need to rename the files correspondingly.
  - In your main procedure, instantiate your `Generic_ContactBooks` package to create a ContactBooks package with a specified maximum size.
  - Try running your application with the size big enough and with it too small to hold what we want to put in it as we did when testing our exception.

Section 15

Multi-Tasking Basics

Tasking Basics

- Ada includes built-in support for concurrent processing using Ada *tasks.*
- Ada tasks run concurrently and can interact with each other.
- It is sometimes helpful to imagine that each Ada task is running on a separate CPU or even a separate computer.
- Independent tasks can be started (activated) or stopped (terminated) and can communicate with each other through special mechanisms built-in to the language.

Tasking Caveats

- If the machine your program is running on has only a single CPU, then Ada cannot actually make the independent tasks run simultaneously. It then must simulate multiple simultaneous tasks. This is much like a multi-process or multi-user OS running on a single CPU must simulate simultaneous processes. This involves some overhead.
- Tasking can be over-used or under-used. Using many unnecessary tasks can make your program slow, hard to understand, and hard to maintain. Use tasks for things that naturally should happen simultaneously in your domain.
- For tasks to work really well, the underlying operating system must have some reasonable support for threading or light-weight processes.
- Every Ada program contains at least 1 task. The “main” task is referred to as the *environment* task.

A Simple Counting Task

```
-- File: counting.ads
package Counting is
   task type Counter (Task_Number : Integer; Count_To : Positive) is
      entry Start;
   end Counter;
end Counting;

-- File: counting.adb
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;

package body Counting is
   task body Counter is
   begin
      accept Start do
         for I in 1 .. Count_to loop
            Put(Task_Number); Put(“: ”); Put(I);
            New_Line;
            delay 1.0; -- wait a second before proceeding
         end loop;
      end Start;
   end Counter;
end Counting;
```

Running a Counting Task

```
with Counting;
use  Counting;

procedure Count is
   Task_1 : Counter(Task_Number => 1, Count_To => 20);
begin
   Task_1.Start;
end Count;
```

- Output

```
         1:           1
         1:           2
         1:           3
         1:           4
         . . .
         1:          20
```

Running Multiple Tasks

```
with Counting;
use  Counting;

procedure Count is
   Task_1 : Counter(Task_Number => 1, Count_To => 20);
   Task_2 : Counter(Task_Number => 2, Count_To => 10);
begin
   Task_1.Start;
   Task_2.Start;
end Count;
```

- Output

```
         1:         1
         1:         2
         . . .
         1:        20
         2:         1
         3:         2
         . . .
         5:        10
```

- This hardly seems concurrent or parallel.

Running Multiple Tasks

- What we did wrong was we programmed `Start` as if it were a subprogram and not a point of connection between tasks. While `Start` can take parameters like a procedure, `Start` is actually an *entry*, not a subprogram.
- An entry is where one task can "call" another task. The calling task will generally wait (block) until the called task reaches an accept statement for the entry being called. When the calling task is requesting an entry and the called task is ready to accept that entry, we have a rendezvous between the two tasks and parameters can be passed.
- Consider the task body on the following slide

A Better Counting Task

```
-- File: counting.ads
package Counting is
   task type Counter (Task_Number : Integer; Count_To : Positive) is
      entry Start;
   end Counter;
end Counting;

-- File: counting.adb
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;

package body Counting is
   task body Counter is
   begin
      accept Start;

      for I in 1 .. Count_to loop
         Put(Task_Number); Put(“: ”); Put(I);
         New_Line;
         delay 1.0; -- wait a second before proceeding
      end loop;
   end Counter;
end Counting;
```

Running Multiple Tasks

```
with Counting;
use  Counting;

procedure Count is
   Task_1 : Counter(Task_Number => 1, Count_To => 20);
   Task_2 : Counter(Task_Number => 2, Count_To => 10);
begin
   Task_1.Start;
   Task_2.Start
end Count;
```

- Output

```
         1         2: :        1         1
         1         2: :        2         2
         1         2: :        3         3
         1:        6
         2:        6
         1:        7
         1:        8
         2:        7
```

- Well, now we've got concurrent. So much so that our output from the two tasks is all jumbled together.

A Little More Task Control: Protected Types

- To give us a little more control we can use a semaphore or locking technique by taking advantage of a *protected type* data object.
- A protected type contains data that tasks can only access through a set of protected operations defined by you (the developer)
- Protected operations are 1 of 3 kinds
  - Protected functions
    - Provide read only access to the protected type's data
    - Since access is read only, multiple tasks can call protected functions simultaneously
  - Protected procedures
    - Provide exclusive read-write access to protected type's data
    - When one task is running a protected procedure of a protected type, no other task can interact with that same protected type
  - Protected entries
    - Like protected procedures but with a Boolean expression that forms a barrier
    - The entry is only open for acceptance of a rendezvous if the barrier is True

Using a Protected Type as a Simple Lock

- Consider adding the following code inside the body of our `Counting` package.

```
protected type Lock is
   entry Seize; -- protected entry
   procedure Release; -- protected procedure
private
   Locked : Boolean := False;
end Lock;

protected body Lock is
   entry Seize when not Locked is
   begin
      Locked := True;
   end Seize;

   procedure Release is
   begin
      Locked := False;
   end Release;
end Lock;

Counting_Lock : Lock;
```

Using a Protected Type as a Simple Lock

- Now we have a `Counting_Lock` which one task can `Seize` and cause other tasks to block until the `Counting_Lock` is released. In our `Counting` package, we modify our task body for the `Counter` task as

```
task body Counter is
   begin
      accept Start;

      for I in 1 .. Count_To loop
         Counting_Lock.Seize;
         Put(Task_Number); Put(": "); Put(I); New_Line;
         delay 0.25;
         Counting_Lock.Release;
      end loop;

   end Counter;
```

- Between the `Seize` and `Release`, other tasks will block at there own attempts to `Seize`. Thus each Task will get to do its output uninterrupted by other output.

More Tasking Details

- Of course protected types can store more data than just a simple Boolean. They are intended to be the primary means of controlling access to data that is shared between Tasks.
- A task can reach a point at which it will `accept` one of any number of entries. This is done with a `select` statement. It is very common for tasks to be written as a “tight loop” that simply executes a select statement for various entries.

```
task body Sample is
  V : Item := Initial_Value;
begin
   loop
      select
         accept Read(X: out Item) do
            X := V;
         end;
      or
         accept Write(X: in Item) do
            V := X;
         end;
      or
         accept End do
            exit;
         end;
      end select;
   end loop;
```

Selective Entry Calls

- Entry calls can be set to “time out” if the task they are trying to communicate with is not ready to accept the requested entry

```
select
   Task.Read(X);
or
   delay 0.15;
   -- could not read in allotted time
end select;
```

- Or if you don't want to wait at all, leave out the delay

```
select
   Task.Read(X);
or
   -- could not read immediately
end select;
```

Selective Accepts

- Task accepts can similarly be made to “time out”

```
select
   accept Request_Value(X : out Integer) do
      -- do work
   end Request_Value;
or
   delay until 0.15;
   -- no-one wanted the value within the allotted time
end select;
```

- Or immediately go on if no task is waiting on the entry

```
select
   accept Request_Value(X : out Integer) do
      -- do work
   end Request_Value;
or
   -- no other task was immediately wanting to rendezvous
   -- so I'm gonna just do something else
end select;
```

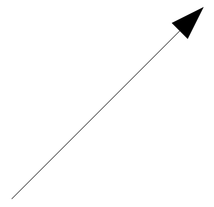
Lab 15: Counting Tasks

- Goal
  - Get some experience with simple tasking in Ada
- Steps
  - Implement “A Better Counting Task” as shown on the slide with that title
  - Implement a main procedure in a separate .adb file as shown on the “Running Multiple Tasks” slide.
  - Build and run your application
  - Put a protected type “Lock” in your Counting package and modify your tasks to use the lock as shown on the slides titled “Using a Protected Type as a Simple Lock”

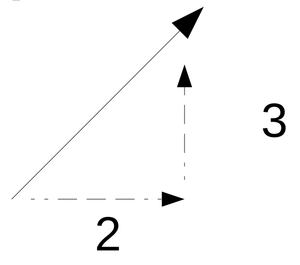
Section 16
Operator Overloading

Review of Vector Arithmetic - 1

- A Vector has a direction and a magnitude (length)
- Example:

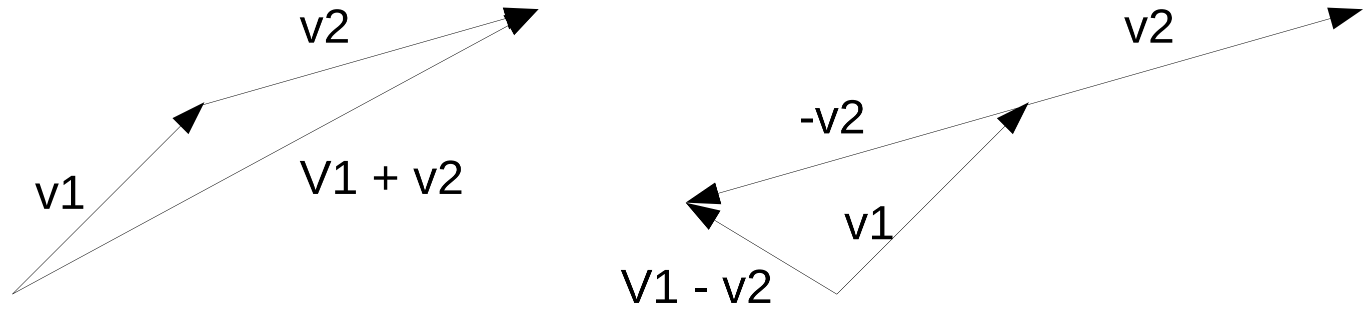


- Usually represented as a set of coordinates
- For example (2, 3)



Review of Vector Arithmetic - 2

- Adding two vectors is a matter of adding their corresponding coordinates
- Subtracting two vectors is a matter of subtracting their corresponding coordinates
- Example: v1 = (2, 3) v2 = (4, 1)
- v1 + v2 = (2 + 4, 3 + 1) = (6, 4)   v1 – v2 = (2 – 4, 3 – 1) = (-2, 2)



- Length of vector | v1 | = sqrt($x^2 + y^2$) = sqrt(4 + 9) = sqrt(13)
- Multiplying a scalar (simple number) times a vector: 2 * (2, 3) = (4, 6)
- Inner Product of two vectors is the “sum of the products of the corresponding coordinates”: v1 * v2 = (2 * 4) + (3 * 1) = 8 + 3 = 11

A Vector Type with Overloaded Operators - 1

```
package Vectors is

   type Vector is array (Integer range <>) of Float;

   function "+"(Left : Vector; Right : Vector) return Vector;
   function "-"(V : Vector) return Vector;
   function "-"(Left : Vector; Right : Vector) return Vector;
   function "*"(S : Float; V : Vector) return Vector;
   function "*"(Left : Vector; Right : Vector) return Float;

end Vectors;
```

A Vector Type
with Overloaded Operators - 2

```
package Vectors is

   function "+"(Left : Vector; Right : Vector) return Vector is
      Result : Vector(Left'Range);
   begin
      for I in Left'Range loop
         Result(I) := Left(I) + Right(I);
      end loop;
      return Result;
   end "+";


   function "-"(V : Vector) return Vector is
      Result : Vector(V'Range);
   begin
      for I in V'Range loop
         Result(I) := -V(I);
      end loop;
      return Result;
   end "-";
```

A Vector Type with Overloaded Operators - 3

```
 function "-"(Left : Vector; Right : Vector) return Vector is
   Neg_Right : Vector(Right'Range);
   Result : Vector(Left'Range);
begin
   Neg_Right := -Right;
   Result := Left + Neg_Right;
   return Result;
end "-";

function "*"(S : Float; V : Vector) return Vector is
   Result : Vector(V'Range);
begin
   for I in V'Range loop
      Result(I) := S * V(I);
   end loop;
   return Result;
end "*";
```

A Vector Type with Overloaded Operators - 4

```
   function "*"(Left : Vector; Right : Vector) return Float is
      Result : Float := 0.0;
   begin
      for I in Left'Range loop
         Result := Result + (Left(I) * Right(I));
      end loop;
      return Result;
   end "*";

end Vectors;
```

Main for testing Vector Type - 1

```
with Ada.Float_Text_IO, Ada.Text_IO, Vectors;
use  Ada.Float_Text_IO, Ada.Text_IO, Vectors;

procedure OperatorOverloading is
   Vector1, Vector2, Vector3 : Vector(1..2);
   Result  : Float;
begin
   Vector1 := (2.0, 3.0); Vector2 := (4.0, 1.0); New_Line(2);
   Vector3 := Vector1 + Vector2;
   Put("After addition: Vector3 = ");
   Put("(");Put(Vector3(1));Put(", ");Put(Vector3(2)); Put(")");
   New_Line(2);

   Vector3 := -Vector1;
   Put("After negation: Vector3 = ");
   Put("(");Put(Vector3(1));Put(", ");Put(Vector3(2)); Put(")");
   New_Line(2);
```

Main for testing Vector Type - 2

```
   Vector3 := Vector1 - Vector2;
   Put("After subtraction: Vector3 = ");
   Put("(");Put(Vector3(1));Put(", ");Put(Vector3(2));Put(")");
   New_Line(2);

   Vector3 := 2.0 * Vector1;
   Put("After scalar multiplication: Vector3 = ");
   Put("(");Put(Vector3(1));Put(", ");Put(Vector3(2)); Put(")");
   New_Line(2);

   Result := Vector1 * Vector2;
   Put("After inner product multiplication: Result = ");
   Put(Result);
   New_Line(2);

end OperatorOverloading;
```

Ada Operators

- The following Ada operators can be overridden by using the operator string in double quotes

```
abs   and   mod   not   or    rem   xor
 =     /=    <     <=    >     >=
 +     -     *     /     **    &
```

Lab: Currency

- Goal
  - Write and Test an Ada Package that defines a Currency Type with overloaded operators
- Steps
  - For this lab you have a supplied starting point
  - It defines a package `Currencies` and another package `Currencies.IO` and a test “main” procedure.
  - Review the implemented subprograms in `currencies.adb`. Pay particular attention to the `Add` function, the `Subtract` function, and the implemented `"*"` functions.
  - Search for TODO: in the source code in `currencies.adb` and `currencies-io.adb`.
  - Implement the overloaded operator functions marked with TODO:, Review and run the main test procedure.

Section 17
AUnit – Unit Testing Framework

Overview of Unit Testing

- Testing is, of course, an important part of software development.
- Unit testing is a testing method by which individual “units” of source code (sets of one or more program “modules”) are tested to determine if they are fit for use.
- A “unit” is intended to be a very small testable part of an application.
- A unit test consists of a set of “test cases” that exercise the unit and generally define a set of input data, an operation or small set of operations to be performed, and the expected results.
- General Philosophy: Unit tests should be able to be performed automatically and detection of success or failure of any test case should be automatic – i.e. not require any human interpretation or evaluation.
- Encourages writing small, easily testable chunks of code.
- Encourages running the full test suite whenever any changes are made to the code base.

Unit Test Framework

- A Unit Test Framework is a set of code intended to simplify the process of unit testing by helping software developers write easily repeatable, easily evaluated tests.
- Thus encouraging a philosophy of “test early, test often, write lots of tests, run lots of tests.”
- While simultaneously discouraging individual code “ownership.”
- Anybody on the team is free to modify any code, because the test suites are always running to protect us from mistakes.
- If a change “breaks” something without making any tests fail, the “fault” is not in the code that changed. The “fault” is in the lack of a test that indicates what behavior is right and what behavior is in error.

The xUnit Family of Unit Test Frameworks

- SUnit is a unit testing framework written for creation of unit tests for the programming language Smalltalk.
- Originally described by its author (Kent Beck, the creator of Extreme Programming) in 1989.
- Other unit testing frameworks were modeled on SUnit. SUnit is described as the “mother of all unit testing frameworks”.
- Unit testing frameworks based on the design of SUnit are referred to as members of the xUnit family of unit test frameworks.
- Notable unit testing frameworks based on the design of SUnit include: JUnit (Java), CUnit (C), CppUnit (C++), COBOLUnit (COBOL), FUnit (Fortran), PyUnit (a.k.a unittest) (Python).
- AUnit is Ada's xUnit family member.
- AUnit comes pre-installed as part of the GNAT GPL binary.

xUnit: Unit Testing Concepts - 1

- xUnit design establishes a set of concepts and design patterns that are used in all the xUnit family of testing frameworks.
- A *Test Runner* – a main application whose job it is to run a *Test Suite* and report on their success or failure.
- A *Test Suite* – a set of related *Test Cases*. A *Test Suite* has the same interface as an individual *Test Case* and can therefore be thought of and worked with as a single unit, as if it were a single *Test Case*. (The Composite Design Pattern.)
- A *Test Suite* is usually created that corresponds to the basic unit of division in the programming language being used. In Java or C++ this usually means a *Test Suite* for each class. In Ada this usually means a *Test Suite* for each package. *Test Suites* are combined to form larger *Test Suites*. This can be done because each *Test Suite* can behave like a single *Test Case*.

xUnit: Unit Testing Concepts - 2

- A *Test Case* – an individual test to be run that unambiguously succeeds or fails. A subprogram (method, procedure, subroutine, etc.) which generally invokes/tests a single operation and checks the result of that operation by *Asserting* what the result should be.
- Test *Set Up* routine – a subprogram called before each Test Case is run to provide any necessary initialization of data for the test.
- Test *Tear Down* routine – a subprogram called after each Test Case is run to perform any necessary post-Test cleanup
- *Fixtures* – objects available to an entire Test Suite for use in testing. Generally configured for each test by the Set Up routine.
- *Scaffolding* – any code we write that is not part of an actual test case, but is there to support the tests we do write
  - *Set Up*, *Tear Down*, stubbed in replacements for pieces used by the tests but not currently under test (*Mock Objects*).

xUnit: Unit Testing Concepts - 3

- *Assertions* – in your *Test Cases*, you write *Assertions* for things that must be true for the *Test Case* to succeed.
- Along with a condition that you “assert” must be true, you provide a message to be output if the *Test Case* fails.
- The message should explain what test has failed and give enough information that a tester can find out what test failed, not necessarily what implementation caused the failure.

```
AUnit.Assertions.Assert (Boolean_Expression,
String_Description);
```

- Writing *Set Up*, *Tear Down*, and *Assertions* in Test Cases is where most of your work will be done in creating Unit Tests.

AUnit: Setting Up Unit Tests – GNATtest

- Getting everything setup so that:
  - There is a Test Runner that has a Test Suite to run
  - That Test Suite consists of Test Suites for each of your packages
  - The Test Suite for each package consists of Test Cases for each accessible operation for the package
  - There are Set Up and Tear Down routines available
  - There are *Assertion* routines available with messages to be output on failure
- can be a tedious and long task
- Fortunately we have GNATtest to do most of the tedious work for setting this up using AUnit.

GNATtest - 1

- Start with a package (or set of packages) for which you want to create unit tests
- Example:

```
package Math is
   type Int is new Integer;
   function “+” (I1, I2 : Int) return Int;
   function “+” (I1, I2 : Int) return Int;
end Math;

package body Math is
   function “+” (I1, I2 : Int) return Int is
   begin
      return 0;  -- very common to have stubbed in operations
   end “+”;
   function “-” (I1, I2 : Int) return Int is
   begin
      return 0;
   end “+”;
end Math;
```

GNATtest - 2

- Start with a GPS project that contains the packages to be tested.
- Select: Tools →GNATtest → Generate unit test setup
  - Sometimes get gnattest: initialization failed because of failure to create a temporary directory. If so, just repeat Generate unit test setup.
- Creates new project hierarchy:
- `Test_Driver`
  - `AUnit`
  - `test_`*`<your_original_project>`*
    - `AUnit`
    - *`<your_original_project>`*
- This is a dependency hierarchy.
- The `Test_Driver` project depends upon `AUnit` and the project for testing your packages (`test_`*`<your_original_project>`*)
- The `test_<your_original_project>` project also depends on `AUnit` and on *`<your_original_project>`*.

GNATtest - 3

- `Test_Driver` project contains
  - `test_runner.adb` – main procedure
  - `gnattest_main_suite` – the suite of tests to be run by test_runner
  - *`<your_package>`*`-test_data-tests-suite` – a suite consisting of the test cases for each package to be tested
- `test_`*`<your_original_project>`* contains
  - *`<your_package>`*`-test_data` – Set Up and Tear Down
  - *`<your_package>`*`-test_data-tests` – Test Cases
- It is in the Set Up, Tear Down, and Test Cases where you do your work.

GNATtest - 4

- Example Test Case

```
package body Math.Test_Data.Tests is

--  begin read only
   procedure Test_Plus (Gnattest_T : in out Test):
   procedure Test_Plus (Gnattest_T : in out Test) renames Test_Plus;
--  id:2.2/395e90b00be8ad74/Plus/1/0/
   procedure Test_Plus (Gnattest_T : in out Test) is
   -- math.ads:4:4:”+”
--  end read only
      pragma Unreferenced (Gnattest_T);
   begin

      AUnit.Assertions.Assert
        (Gnattest_Generated.Default_Assert_Value,
         “Test not implemented.”);

--  begin read only
   end Test_Plus;
--  end read only
```

Suppress some warnings.

This is what you replace with your own test assertions.

GNATtest - 5

- Example Set Up and Tear Down

```
package body Math.Test_Data is

   procedure Set_Up (Gnattest_T : in out Test) is
      pragma Unreferenced (Gnattest_T);
   begin
      null;
   end Set_Up;

   procedure Tear_Down (Gnattest_T : in out Test) is
      pragma Unrefrenced (Gnattest_T);
   begin
      null;
   end Tear_Down;

end Math.Test_Data;
```

Notice that the tests themselves are in a child package of the test data. This makes the test data available for all the tests.

Called before each test case.

Called after each test case.

This package is a child of the package being tested. So it has access to the “internals” of the package being tested. All public and private declarations of the parent are visible to the child.

GNATtest - 6

- Prepare *scaffolding*

```
package Math.Test_Data is

   I1, I2 : Int;

--  begin read only
   type Test is new Aunit.Test_Fixtures.Test_Fixture
--  end read only
   with null record;


   procedure Set_Up (Gnattest_T : in out Test);
   procedure Tear_Down (Gnattest_T : in out Test);

end Math.Test_Data;
```

Notice we are in the specification not the body.

GNATtest - 7

- Prepare *scaffolding*

Notice we are in the body.

```
package body Math.Test_Data is

   procedure Set_Up (Gnattest_T : in out Test) is
   begin
      I1 := 3;
      I2 := 2;
   end Set_Up;

   procedure Tear_Down (Gnattest_T : in out Test) is
   begin
      null;
   end Tear_Down;

end Math.Test_Data;
```

GNATtest - 8

- Modify Test Case

```
package body Math.Test_Data.Tests is

--  begin read only
   procedure Test_Plus (Gnattest_T : in out Test):
   procedure Test_Plus (Gnattest_T : in out Test) renames Test_Plus;
--  id:2.2/395e90b00be8ad74/Plus/1/0/
   procedure Test_Plus (Gnattest_T : in out Test) is
   -- math.ads:4:4:”+”
--  end read only
      pragma Unreferenced (Gnattest_T);
   begin

      AUnit.Assertions.Assert
        (I1 + I2 = 5, “Simple addition failed.”);

--  begin read only
   end Test_Plus;
--  end read only
```

GNATtest - 9

- Compile and run Test_Drive project
- Make sure to leave the `Default test behavior` set to `Fail`
- Then Execute the test
- See **Messages** tab:

```
C:\tbb\GoogleDrive\projects\sluwfc\AdaFundamentals\Ex_AUnit\gnatte
st\harness\test_runner --passed-tests=hide –skeleton-default=fail
math.ads:4:4: corresponding test FAILED: Simple addition fails.
(math-test_data-tests.adb:25)
math.ads:6:4: corresponding test FAILED: Test not implemented.
(math-test_data-tests.adb:45)
2 tests run: 0 passed; 2 failed; 0 crashed.

[2015-04-18 10:14:25] process terminated successfully, elapsed
time: 00.24s
```

Lab1: GNATtest - 1

- Start with Lab17_1_AUnit1
- Open up the `gnattest/harness/test_driver.gpr` project instead of the `ex_Aunit.gpr` project.
- Build and run the project
- Implement “+” and “-” correctly.
  - Convert the Int objects to Integer objects, do the math, and convert back.
- Write more tests for “Plus” in Test_Plus
- Write tests for “Minus” too
- Compile and run Test_Drive project until all succeed.

Lab1: GNATtest - 2

- Write a “*” operator in the Math package
- There won't be a test for the “*” operator.
- Generate tests for the “*” operation by:
  - Tools → GNATtest → Exit from harness project
  - Select Ex_AUnit project then choose
    Tools → GNATtest → Generate unit test setup
  - Open body of tests for the math package (may be asked to reload from disk)
  - Notice that your existing test cases have been left “as is” (not overwritten)
  - But a new test case, Test_Multiply has been created with a “Test not implemented.” message

Lab2: GNATtest - 3

- Write Unit tests for the `Currencies` package that was created as the Operator Overloading lab
- The start point for this lab is the solution for the Operator Overloading lab
- Open the `lab_OperatorOverloading.gpr` project
- Create the test harness: Tools → GNATtest → Generate unit test setup
- Open `test_lab_operator_overloading` project
- For now just make the tests for Currencies.IO null.
  - Replace code with either `null;` or `AUnit.Assertions.Assert(True, “null test”);`
-

Lab2: GNATtest - 4

- Implement tests for `Currencies`
  - `Test_Init_Currency` – make sure initialized currency has expected properties
  - `Test_Init_Currency` – try to initialize a currency with 101 cents
    - Write separate procedure (e.g. `Test_Raising_Exception`) that internally calls `Init_Currency(C, 200, 101)`
    - Verify that calling the `Test_Raising_Exception` procedure raises an exception using

      ```
      Aunit.Assertions.Assert_Exception(Test_Raising_Exception'Access,
                                        "should raise");
      ```
  - `Test_1_Make_Currency` – testing make currency from a Float
  - `Test_2_Make_Currency` – testing make currency from Dollars and Cents
  - `Test_3_Make_Currency` – testing make currency from Pennies
  - `Test_Make_Float` – test creating a float from a currency
  - `Test_Dollars`, `Test_Cents`, `Test_IsPositive` – go ahead and make these null tests
  - `Test_Less_Than`, `Test_Greater_Than`, `Test_Less_Than_Or_Equal`, `Test_Greater_Than_Or_Equal`, `Test_Unary_Plus`, `Test_Unary_Minus`, `Test_Abs`
  - `Test_Plus` (binary), `Test_Minus` (binary)
  - `Test_1_Multiply` – Left : Integer; Right : Currency
  - `Test_2_Multiply` – Left : Currency; Right : Integer
  - `Test_3_Multiply` – Left : Float; Right : Currency
  - `Test_4_Multipy` – Left : Currency; Right : Float

Section 18

A Very Brief Introduction to the Ada95 Booch Components

Ada95 Booch Components Overview

- A container library, supporting Bags, Collections, Maps, Queues, Rings, Sets, Stacks, and Trees
- Main web site: http://booch95.sourceforge.net/index.html
- Download site: http://sourceforge.net/projects/booch95/
- It often saves time, money, and testing effort to use pre-existing components rather than try to implement your own.
- Not by Grady Booch. Instead a port of Grady Booch's C++ components to Ada
- From the Booch 95 web-site: “Many people, faced with the BCs (Booch Components) for the first time, choose Lists (Single or Double) as their standard Container. This is probably *not* the best choice. Lists are complex entities which would be useful if you were implementing a list-processing engine like Lisp. ***You'll be a lot better off using Collections!***

Preparing to use the Ada95 Booch Components

- Download the ZIP file from: http://sourceforge.net/projects/booch95/
- Unzip the zip file (`bc-20130322.zip` as of this writing) to create a `bc-20130322` directory
- In the `bc-20130322` directory create a sub-directory named `lib-static`
- Open the `bc.gpr` project file and build the library
  - You can ignore the style warning about the a misplaced “then”
- Start with Lab_Booch_StartingPoint and make that project dependent upon the Booch project
  - Open lab .gpr file
  - Right click on Project and select Project → Dependencies
  - Add From File, find the `bc.gpr`, Apply
  - See the BC project added as a dependency to your lab project

Instantiation of a Container

- You don't put your own items directly into a Booch Component.
- Instead you create a `Container` to hold your item. A `Container` is a *controlled tagged record* that encloses your item.
- For example, if you want to put individual characters in a Booch Component, you instantiate a new package to create a type that will contain the characters.

```
with BC.Containers.Lists.Single;
with BC.Support.Standard_Storage; -- allocates from the heap
package CharacterContainers is
   new BC.Containers (Item => Character);
```

- This instantiates a new package called `CharacterContainers` which provides a type that can *contain* the items we want (things of type `Character`)
- The `CharacterContainers` package also provides us with access to iterators that allow us to traverse a Booch component

Iterating over a Booch Component

- Create an iterator with the `New_Iterator` function
- Manipulate an iterator with the following subprograms
  - `Reset` – start a new traversal at the first item
  - `Next` – continue to another item in the component
  - `Is_Done` – True if there are no more items
  - `Current_Item` – return the current item

Use of Booch Components

- Review the following Examples
  - `Ex_Booch_Character_List`
  - `Ex_Booch_Character_Collection`
  - `Ex_Booch_PersonalContact_Collection`
  - `Ex_Booch_ContactClass_Collection`
    - IMPORTANT NOTE: THIS DOES NOT WORK!
    - TALK ABOUT WHY.
  - `Lab_Booch_Solution`

Types of Booch Components - 1

- Bags – Unordered group of items. Duplicates are counted but not actually stored.
- Collections – Ordered group. Duplicates allowed and stored.
- Deques – Double-ended queues
- Single Linked Lists – Sequence of 0 or more items with head and pointer to next item.
- Double Linked List – Sequence of 0 or more items with head and pointer to next and previous item.
- Maps – A Set with relationships between pairs of items (Set of Keys, Key ==> Value)
- Queues – First in, First out list
- Ordered (Priority) Queues – A Sorted List, items removed from the front
- Rings – a Deque with only one endpoint
- Sets – Unorderd group. Duplicates not allowed.
- Stacks – Last in, First out list

Types of Booch Components - 2

- Stacks – Last in, First out list
- AVL Trees – Balanced binary trees
- Binary Trees – List with 2 successors per item
- Multiway Trees – List with an arbitrary number of children per item
- Directed Graphs – Groups of items with one-way relationships
- Undirected Graphs – Groups of items with two-way relationships
- Smart Pointers – Access types that automatically deallocate themselves

Section 19

Ada Program Structure

Some Program Structure Guidelines - 1

- Real world Ada programs are seldom as simple and short as those shown as examples and produced as lab exercise in a course like this.
- They are generally composed of a set of many interrelated Ada packages.
- Some recommendations are given in the publication *Ada Quality and Style: Guidelines for Professional Programmers* (a link to the online version of this publication is provided in the “Ada Related References and Links” section later in these materials.)

Some Program Structure Guidelines - 2

- A few of guidelines selected from the Quality and Style publication.
  - Packages should serve a single purpose. (Try to make each package a coherent unit, not a just a collection of objects and subprograms.)
  - You can (and often should) define more than one type in a package. If two types are closely related feel free to put them in the same package. In our Contacts example, it might very well be considered appropriate to put the Contact_Type, the Personal_Contact_Type, and the Business_Contact_Type all in one package. Of course, a type and its access type should also be defined in the same package.
  - Put only what is absolutely necessary for use of the package into the package specification. Hide as many implementation details as possible. The private part of a specification is “exposed” to users of the package in that they can see it, but not exposed in that they can use it.
  - Avoid defining a variable in a package specification. This creates a global variable that anyone using a with clause can access and manipulate.
  - Try to use with clauses only when they are needed. In particular, if a with clause is not needed in a package specification, but the implementation (in the package body) needs the with clause, put the with clause in the body and leave it out of the specification.
  - Don't be to free with using use clauses. Sometimes it's just clearer to go ahead and write the full specification of a type or subprogram.

Child Packages

- Starting with Ada95, the capability to create *Child Packages* was added.
- A child package is a package that is conceptually part of its parent package, but the code for a child package is not in the same file as the parent package.
- The syntax for created child packages is simple:

```
package X.Y is
. . .
end X.Y;
```

- The package `X.Y` is a child of the package `X`.
- A child package has automatically “withed” its parent package.
- In the private parts of the child package you have access to the private parts of the parent package. You can use private subprograms, access private parts of records, use privately declared types, etc. from the parent package.

Section 20
Miscellaneous

Interfacing with other Languages

- Ada has facilities for “importing” subprograms (functions, procedures, subroutines) from other programming languages so they can be called by Ada programs.
- It also has facilities for “exporting” subprograms so that they can be called from other programming languages.
- Ada “bindings” can be made such that libraries of code written in other programming languages (for creating GUIs or accessing relational databases or interacting more directly with the Operating System) can be easily callable from Ada and appear to be available as Ada packages.
- As has been mentioned, Ada can be compiled to run in the Java Virtual Machine.

Language Defined Attributes

- For every type, Ada predefines a set of operations or attributes. These are those things we've seen accessed via the single quote or “tick mark” used after the type name (e.g. `'Class`, `'First`, `'Last`, `'Range`)
- Some useful attributes that we haven't seen (or have seen only briefly) are:
  - `X'Access` where `X` is a subprogram gives us an access value “pointing” to the subprogram
  - `X'Min` and `X'Max` are defined for any scalar type X and return the minimum and maximum values
  - `X'Round` will round a number of type `X` to the nearest integer. If X is “half-way” it rounds away from 0.
  - `X'Size` will tell you how many bits are used to store something of type X
  - `X'Bit_Order` will tell you what the bit ordering is for an integer type (Little Endian, Big Endian)
-

Recursion

- Ada support a subprogram calling itself – recursion
- As is true with all recursive algorithms, you have to be sure to provide a mechanism to end the recursion. Infinite recursion is predictably a bad thing.
- The classic recursion example is always the computation of the factorial of an integer... n!

```
function Factorial(A : in Integer) return Integer is
begin
   if A < 0 then
      raise Constraint_Error;
   elsif A = 0 then
      return 1;
   else
      return A * Factorial(A - 1); -- recurse
   end if;
end Factorial;
```

Declare Blocks

- At any point in a subprogram you can declare new variables, but to do so you must use a declare block and the declared variable will only exist within that declare block.

```
if Contact in Personal_Contact_Type then
   declare
      PC : Personal_Contact_Type;
   begin
      . . .
   end;
elsif Contact in Business_Contact_Type then
   declare
      BC : Business_Contact_type;
   begin
      . . .
   end;
end if;
```

Pragmas

- Ada allows for compiler directives to be specified as *pragmas.* Compilers are not obliged to support all pragmas.
- There standard form is
  `pragma` *`Name (Parameter list);`*
- Pragmas allow you to ask a compiler
  - To “inline” all calls to a particular subprogram
  - To optimize certain parts of the code for time or space
  - To suppress certain run-time checks (generally a bad idea)
  - To pack the storage representation of a type into the smallest possible space
  - To define the priority to a task
  - Default Parameters

SPARK

- SPARK is a formally defined programming language based on Ada.
- It is essentially a subset of Ada with a constraint language added which is intended for development of high integrity, highly reliable software systems.
- In addition to removing some constructs that are deemed Ada potentially ambiguous, it adds the notion of Contracts to the language which allows for the verification that a subprogram meets its Contracts with the invoker.
- SPARK is used in high profile safety-critical systems (aviation systems), Air Traffic Control applilcations, medical and space systems.

Supplemental Labs

- Implement and try out the recursive `Factorial` function from the Recursion slide.
- Make your `PersonalContacts` package a child package of the `Contacts` package (`Contacts.Personal`) and your `BusinessContacts` package a child package of the Contacts package (`Contacts.Business`)
- Use Declare blocks to change the implementation of ContactBooks.Add_Contact so that it
  - Only declares a variable of type Personal_Contact_Type_Access if the Contact being added is actually a Personal_Contact_Type
  - Only declares a variable of type Business_Contact_Type_Access if the Contact being added is actually a Business_Contact_Type
- Write a simple procedure to find out how many bits are used on your system to store an `Integer`, a `Long_Integer` (if defined for your platform), and a `Short_Integer` (if defined for your platform). Define a type Byte as follows
  `type Byte is mod 256;` and determine how many bits are used to store such a type.

Supplemental Labs

- Write a protected type for a CircularBuffer (Ring Buffer of Integers to be used as a FIFO (First In First Out)
  - Allow it to store up to N (e.g. 8) values
  - Store the values in an Array
  - Have a Next_Input_Location value that keeps track of the next available slot in the Circular Buffer
  - Have an entry named Put which places a new Integer at the next available slot in the buffer
  - If you go beyound the bound of the array then “circle back” and put the next item at the beginning of the buffer
  - Have an entry named Get which retrieves the next avilable value
  - Have a Next_Output_Location value that keeps track of the next slot from which to retrieve a value from the CircularBuffer
  - Have a Count value that keeps track of how many things are in the buffer
  - Your Put entry will have a barrier (when Count < N)
  - Your Get Entry will have a barrier (when Count > 0)
  - Create a GetNumberOfItems entry to retrieve the Count value
  - Write a package that contains a declaration of one of your CircularBuffer objects, a Task which puts some number of Integers into the CircularBuffer (a “producer” task), a task which retrieves values from the buffer (a “consumer” task).
  - Write a main procedure that starts up two instances of your producer task and one instance of your consumer task
  - How will you make the consumer task stop?

Appendix 1

Ada Related References and Links

Ada 95 References - 1

- Ada Home: The Web Site for Ada
  - http://www.adahome.com
  - http://www.adahome.com/Resources/References
  - http://www.adahome.com/Resources/refs/rm95.html
    Ada95 Language Reference Manual (LRM)
- AdaCore – commercial Ada software
  - http://www.adacore.com
  - http://www.adacore.com/developers/documentation
    Developer's Documentation
- Ada Information Clearinghouse
  - http://www.adaic.org
  - http://www.adaic.org/ada-resources/tools-libraries
  - http://www.adaic.org/ada-resources/standards

Ada 95 References - 2

- Ada Quality and Style: Guidelines for Professional Programmers
  - http://www.adahome.com/Resources/refs/aqs.html
- Ada Rationale
  - http://www.adahome.com/Resources/refs/rat95.html
- Ada Annotated Reference Manual
  - http://www.adaic.org/resources/add_content/standards95aarm/AARM_HTML/AA-TTL.html

Ada 95 References - 3

- The Ada Lovelace Tutorial by David A. Wheeler
  - http://www.adahome.com/Tutorials/Lovelace
  - http://www.dwheeler.com/lovelace
  - Portions of these materials are modeled on the excellent and somewhat famous *Lovelace Tutorial*

Appendix 2

Setup for Ada 95 Fundamentals Course
Windows OS
using GNAT Programming Studio (GPS)

Download GNAT Ada GPL 2014

- Visit: http://libre.adacore.com
- Click on the *Download* icon
- Select the *Free Software or Academic Development* option
- Provide email address only if you want to
- Select *Build Your Download Package*
- Above the list of tools that are available as part of GNAT GPL 2014, select your platform: *x86 Windows (32 bits)*
  - Apparently the 64 bit tools are only available as part of the commercial GNAT Pro tools
- Leave the pull down to the right of the platform choice at *GNAT GPL 2014*
- Click the arrow to the left of the *GNAT GPL 2014* product in the list and check the `gnat-gpl-2014-x86-windows-bin.exe` file.
- This will include an Ada Compiler and GNAT Programmers Studio (GPS) an Ada IDE
- Select *Download Selected Files*

Unpack and Install GNAT Ada GPL 2014

- The downloaded .zip file will be named `AdaCore-Download-YYYY-MM-DD_TTTT.zip` (where `YYYY-MM-DD_TTTT` is replaced by the date and time of the download, apparently in US Eastern Time.)
- Unzip the downloaded file to a temporary location.
- There will be a directory structure like: `x86-windows\adagpl-2014\gnatgpl` containing the file `gnat-gpl-2014-x86-windows-bin.exe`. This `.exe` is an installation binary.
- Run the installation binary.
- Be sure to accept the default installtion location proposed by the installation binary (`C:\GNAT\2014`*).* If you change it, then a number of example GPS project files (`.gpr` files) will need to be modified outside of the GPS environment before they are usable.
- At the end of the installation process, leave the *Add install location to search path* and *Associate GNAT GPL with source and project files* check boxes selected.
- Click finish.

Test the GPS and Ada installation - 1

- Start GPS
  - Under Windows 7: Start → GNAT → 2014 → GPS
- Select *Create new project with wizard* then *OK*
- Select *Single Project* then *Forward*
- Name: `HelloWorld`
- Directory: *your choice*
- Select *Forward*
- Select Language *Ada* and *Forward*
- Select *Forward* without changing default options until *Forward* is no longer selectable
- Select *Apply*
- Allow directory to be created if necessary
- Close “Tip of the Day” dialog

Test the GPS and Ada installation - 2

- Select File → New
- Insert the following content

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
   Put_Line ("Hello WORLD!");
end Hello;
```

- Save file as `Hello.adb` in project directory

Test the GPS and Ada installation - 3

- Select Project → Edit Project Properties
- Select *Main files* tab along left hand side
- Select + *Add* button and check box next to `Hello.adb`
- Select OK twice
- Select Build → Project → Build All → Execute
- Look for `process terminated successfully, elapsed time: XX.XXs` message in Messages tab at bottom of window
- Select Build → Run → Hello → Execute
- Look for Hello WORLD! Output in Run:Hello tab at bottom of window = This means successful installation!!!
- Put a shortcut to GPS on the Desktop please.

Install Alternative Editor (for editing files outside of GPS)

- An alternative editor that also recognizes and does syntax highlighting for Ada will come in handy.
- Geany (www.geany.org) is a good alternative
- Visit www.geany.org/Download/Releases
- From that page, download and run geany-1.24_setup.exe
  - Accept default installation location
- Next visit http://plugins.geany.org/downloads.html
- From that page, download and run geany-plugins-1.24_setup.exe
  - Accept default installation location
- Put a shortcut to Geany on the Desktop please.