vii) ***Derived Types:*** They allow the definitions of new types driven from existing types. Even though the derived types inherit the parent type's operations, they are not compatible (logically distinctive)! To make them compatible we need to use explicit conversion.

```
type PERCEN is new INTEGER range 0..100;
P : PERCEN;   I: INTEGER;
P:= I;   -- illegal          also      I := P; -- illegal

But:   P := PERCEN(I);  -- is legal
       I := INTEGER(P);  -- is legal
```

PACKAGES

"***Packaging***" is a basic data abstraction and modularity mechanism in Ada for grouping logically related data values and their associated set of operations (ADT), forming an environment that can be accessed via ***mutual consent*** between the user and the implementer of the package.

i) Package Interface (Specification):

```
package  <NAME>  is
 -- visible section: objects and type declarations; procedure/function headers

 [ -- private section: optional section that has private types and all deferred
                       constructs concrete declarations. It is to be seen by the
                       compiler not by the users, for optimal space allocation
                       by the compiler.]
end  [(optional) <NAME>]
```

ii) Package Body (Implementation):

package body <NAME> **is**

[-- optional: 1) local declarations, invisible to the users
2) concrete implementation of the procedures and functions package operations (above).]
[--Optional section that with some Ada statements, e.g., for initialization]
[**exception** -- Optional section that with Ada exception handlers]

end **[** (optional) <NAME>];

Package's "Specification" and "Body" can be compiled separately (same as in Modula-2). A package can be lib module, or nested within a subprograms.

- Packages can be a lib modules and separate compilation units.
- "***Private***" types are similar to the Modula-2 "***opaque***" types, since the user can not access their concrete implementation for "**true**" ADTs.
- There is no explicit "import/export" commands, yet the idea of mutual consent (user/implementer) visibility still holds, all names defined at the package interface and outside the private section are visible to the package's users (i.e., **public** names).
- To invoke a lib package we use the "***with***" clause, in the declaration of other subprograms:

  with <package-name> ;

  and we utilize the "***use***" clause to make the package name visible in the code with no need for prefixing all of its operations with its name via the dot notation, i.e., <package-name>.<operation-name>.

  use <package-name>;

  Notice: the **with** must be before the **use**!

Generic Packaging Facility: (Static Genericity)

- A second order static polymorphism (power) to textually group a number of different versions of an ADT (e.g., based on elements' types or numbers) into one "generic" parameterized template to be instantiated at compile yielding concrete package definitions based on the provided actual values of the generic parameters.

generic -- (an example at the bottom of page 273, generic package "stack")
<list of dummy formal arguments>
package <NAME> is
<proc/func's headers (ops) containing some of the above arguments>
end <NAME>;

package body <NAME> **is** -- (example page 275, generic "stack")
<the concrete implementation of all names and operations that contains all of the above arguments>
end <NAME>**;**

- The instantiation process is a simple **textual replacement** of every instantiation statement with a copy of the generic package template after replacing every dummy generic parameter by its corresponding actual argument that has been provided in the **instantiation statement:**

package <INSTANT-PACKAGE-NAME> **is new** <GENERIC-PACKAGE-NAME>
-- (list of actual arguments values);
-- (for example: look near the top of page 274)

- It is completely carried out by the compiler, thus it's a **static** genericity of limited power since each instantiated package will use a separate set of operations' codes; no code sharing.

- The goal is to shorten the Ada programs helping in the code maintenance, and program coding. BUT, we will have a problem. A one page Ada code (filled with many generic instantiation statements) might "***explode***" into 100 pages of text (***code explosion*** problem in Ada), with huge compiled machine code due to the non-sharing of the operations between different instances with different type elements.
- A truly powerful instantiation would allow all versions to share the same codes of operations, which is not successfully done in Ada.
- Attempts to share operations codes in Ada have been failed due to the same back draw and insecure assumption that "same size" means "same type"; the same failure at Modula-2 side.

Internal Versus External ADT Implementation in Ada:

Internal Representation of types:

- Internal representation of an ADT resembles the object orientation view of ADTs (in OOLs). As shown in the "Stack1" example in pages 268-270, we find that the type part of the ADT (i.e., "*ST*" type) is **NOT** listed in the package interface (bottom of page 268), thus there is no stack type-name to be exported for use outside the package. Also, the stack data-structure's concrete implementation is listed ("***internally***") inside the body implementation (top of page 270). In the above internal representation of the stack ADT, we deal with the stack as an integrated "**object**", not as a type (set of operations) to be used to declare names of such type. Hence, in dealing with the stack as an object, we send it messages and expect that it will act on itself (how? we do not care -- **abstraction**!).

For example, at mid of page 269:

```
declare
use Stack1;   -- assuming it is visible and not a lib package
…..
Push(I);  -- a message sent to Stack1 ADT to push I in itself.
Pop(N);  -- a message sent to Stack1 ADT to pop its the top
            element into N.
```

- In order to declare another stack, with different **type element** or **size**, we need to write another package (**inefficient**!!), or use the generic package facility, ending up with duplicate sets of the same operations, one per every stack. The problem is that even though we view the stack as an OO **object**, it is **NOT** as **powerful** as a typical OOL's ***class***!! **(since, with OOL's classes we can instantiate dynamically many instants of one class)**
- The type "*ST*" is truly "***opaque***", since its name and implementation are hidden inside the body, not listed at the interface for exportation.

External Representation (Value) of types:

For more efficient packaging of ADTs in Ada, we can use the "value" approach of representing the ADT types. In this case we externally export the "type" (set of operations) as a ***value*** to the users, to be used to **declare** more than one ADTs that share the "same" set of operations in the original package. In the example of the "Stack_Type" package, page 276-277, we export the type part of the ADT ("***Stack***") by listing its name in the package interface (top of page 277), and we implement it in the "**private**" section, for compilation efficiency; though it is still an "**opaque**" type, i.e., the user can not access the details of the implementation (why?). At the user code, we can instantiate many versions of the same stack by simply declaring each with the same exported type '*Stack*":

d**eclare**

use Stack_Type; -- assume it is not a lib package

Stack1: Stack := **new** Stack;

Stack2: Stack := **new** Stack;

* -- instantiate 100 stacks that share the same set of operations, at declaration "**elaboration**" (run) time

*

Stack100: Stack := **new** Stack;

begin

Push (Stack87, I) -- in the instantiated stack87, push I.

We are not sending a message to anyone, instead we are invoking the "Push" operation (of the type Stack_Type) passing it our stack as an actual parameter value (the corresponding formal is in-out) to be modified by pushing "I".

It should be clear that we are dealing with the ADT stack as a type *Stack* (data-structure, in this case array) and its set of operations that work on them, defined in an Ada package, and we export such type to the user to be used as any other built-in type (e.g., INTEGER, FLOAT, etc).

Questions:

Is the above implementation considered ***polymorphic***? Justify.
Yes, since more than one instantiated stacks are aliasing on the same set of operations.

If the answer is yes, is it "dynamic" or "static" ***polymorphic***? It is only ***dynamic*** instantiation of elements of the type "ST", i.e., Stack1 to Stack100, since it is carried out at the declaration **elaboration** time.

Is it "genericity? **NO**, since all stacks are of the "***same***" type (INTEGER); only when all Stacks use the same set of operations, then it is ***dynamic genericity***.

Parameter Passing In Ada

Ada solved the problematic confusion of the tangled *user* and *compiler* **views** of parameters by making them "**orthogonal**" (i.e., independent).
The **user** informs the **compiler** of his/her view of the formal parameter, as follows:

i) "**in**": The programmer has in mind that such parameter is to be used to **input a value** into the subprogram, and nothing else, e.g., sending raw material, placed in the corresponding formal parameter, to be processed by the callee module. Thus, the compiler will issue an error when the user attempt to change the value of such formal parameter, e.g., when used as a left hand side of an assignment (*l-val*). The parameter acts as **constant.**

ii) "**out**": The programmer has in mind that such parameter is to be used to transfer some produced results by the recipient callee, via its corresponding formal parameter. Old versions of Ada disallowed its use in the right hand side of an assignment (*r-val*), or in any expression.

[Just a note:
Ada 95 allowed that based on its implementation as the "result" half of by "value-result" in FORTRAN! Hence, Ada 95 allows the utilization of its value, assuming its been modified before use (??!!), even when the compiler implement it by reference since there will not

be any intermediate change to its corresponding actual, until the return from the subprogram.]

iii) "**in out**": used to input a value to a subprogram, and also output a value from the subprogram.

In all of the aforementioned different user's views, the compiler attempts to implements the parameter passing efficiently, isolating (abstracting/hiding) the implementation details from the user views. If the actual parameter is large size **composite** data structure (e.g., big array) the compiler will pass a **reference** to the actual parameter into its corresponding formal, otherwise (i.e., simple name **scalar** actual) it passes its **value**. Notice that in this case passing a reference of an ***in*** parameter is not a threat to security since the compiler guards against any attempts to change it; also in case of ***out*** parameters, any change will be effective only after exiting the callee (the result-half of value-result), hence there is no unwanted side effects of global and by reference, as in other languages.

Question: Do we have a problem in Ada 95 implementation of the "out" parameter passing? YES/NO! (explain).