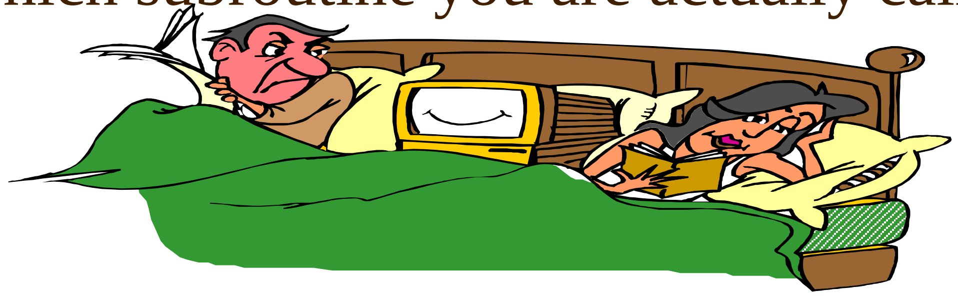
Problems with Dynamic Binding

- ✓ It lengthens execution time
  - Each call causes slight time delay
    - Tag of the variable must be examined
    - Correct subprogram must be found by searching inheritance tree from the “tag” upwards
- ✓ It is non-deterministic
  - I can bound it, but only as an upper limit or average case
- ✓ It makes verification difficult, as you can’t formally define which subroutine you are actually calling



Dynamic Binding Ada 95 does it better!!

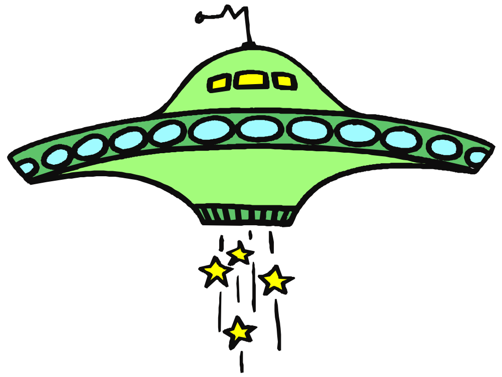
- Note that under Ada 95 and it's library, all dispatching calls are guaranteed to find a subprogram at run time
- Dispatching calls are only made when a procedure that has accepted a class-wide parameter passes this same parameter off to a non class-wide procedure

Actual Parameter Type	Formal Parameter Type	
	Specific	Class-Wide
Specific	static binding	class-wide procedure
Class-Wide	dispatching call	class-wide procedure

Class Wide Programming (Dispatching using pointers)

```
-- Vehicles held as a heterogeneous list using an access type.
GIVEN:  type Vehicle_Ptr is access all Vehicle'Class;
```

```
--control routine can manipulate the vehicles directly from the list.
procedure Move_All is
    Next : Vehicle_Ptr;
begin
    ...
    Next :=  Some_Vehicle;   --  Get next vehicle
    ...
    Start (Next.all);            --  Dispatch to appropriate Handle
    ...                          --  Note the dereferencing of pointer
end Move_All;
```



Dynamic Selection- Referencing a subprogram

An access type can refer to a subprogram; an access-to-subprogram value can be created by the ‘Access attribute and a subprogram can be called indirectly by dereferencing such an access value.

```
  type Trig_Function is access function (F : Float) return Float;
  T : Trig_Function;
  X, Theta : Float;
T := Sin'Access;

X := T(Theta);

X := T.all (Theta);
```

T can “point to” functions such as Sin, Cos and Tan

Indirectly call the subprogram currently referred to

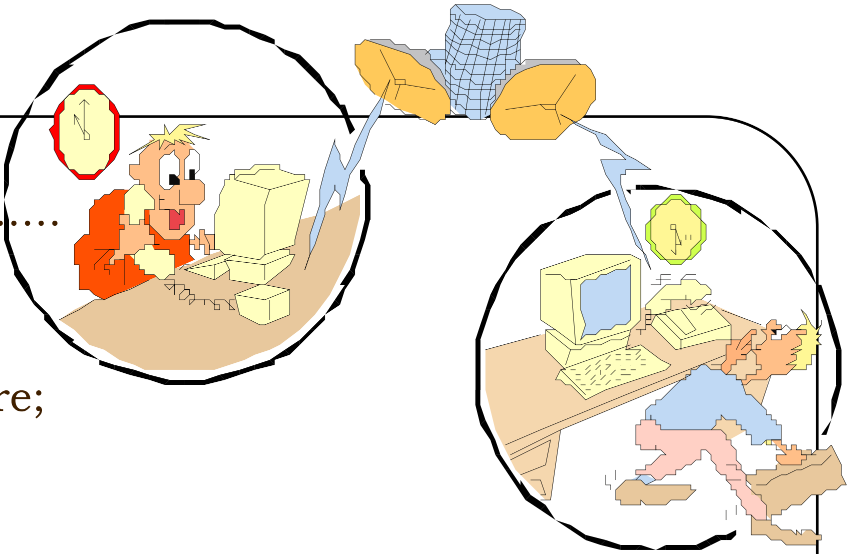
Calling a subprogram via a pointer (and default parameters)

```
type Message_Procedure is access procedure(M : in String := “Error!”);

procedure Default_Message_Procedure (M : in String);

Give_Message : Message_Procedure := Default_Message_Procedure’Access;
...
procedure Other_Procedure (M : in String);
...
Give_Message := Other_Procedure’Access;
...
Give_Message (“File not found.”);      --calls what Give_Message is pointing to
Give_Message.all;                      --ditto, but uses default!
```

Callbacks



```
procedure Call_Word_Processor is......

procedure Ring_Bell is ..........

type Action_Call is access procedure;

type Button_Type is
        record
                X_Pos : Integer;
                Y_Pos : Integer;
                Action_When_Left_Button_Pushed     : Action_Call;
                Action_When_Right_Button_Pushed  : Action_Call;
        end;

Button_1 : Button_Type := ( 100,  50,  Call_Word_Processor'access,
                                                   Ring_Bell'access );
```

Constructors/Destructors (Controlled types in Ada95)

```
package Ada.Finalization is   --system defined package

type Controlled is abstract tagged private;

procedure Initialize  (Object: in out Controlled);
procedure Adjust      (Object: in out Controlled);
procedure Finalize    (Object: in out Controlled);
```

Initialize-	**implicitly called after storage is allocated, and also after any default initialization.**
Adjust-	**implicitly called after a *copy* is performed. All tagged types are passed by reference. Thus, Adjust is called for function return and assignment.**
Finalize-	**implicitly called after object becomes inaccessible (leaving scope, deallocation of object) or before assignment.**

Multiple Inheritance

- ✓ There is no built-in multiple inheritance mechanism in Ada 95
  - – limited usefulness in language with good module and child libraries
  - – adds overhead to processing
  - – typically used to “program around” rather than to create a true hierarchy from several classes
- ✓ It can be simulated in two ways
  - – sibling inheritance - creating a record with components of each type you want to inherit from
  - – mix-in inheritance - using generics to extend a type

Mixing Inheritance

Create a generic that takes a tagged type as a parameter

```
generic
        type Parent_Type is tagged private;
package  Mixin is
…..
        type Mixed_Type is new Parent_Type with private;

end Mixin_Package;
```

```
type Original_Type is tagged private;
package Mixin_Package is new Mixin (Original_Type);
```

Type Mixin_Package.Mixin inherits from both Original_Type and from Mixed_Type. It is a member of both Original_Type'Class and Mixed_Type'Class.

Program Libraries

The Ada program library brings important benefits by extending the strong typing across the boundaries between separately compiled units.

Unfortunately, the flat nature of the Ada 83 library gave problems of visibility control; for example it prevented two library packages from sharing a full view of a private type.

A common consequence of the flat structure was that packages become large and monolithic. This hindered understanding and increased the cost of recompilation.

A more flexible and hierarchical structure was necessary.

Hierarchical Libraries

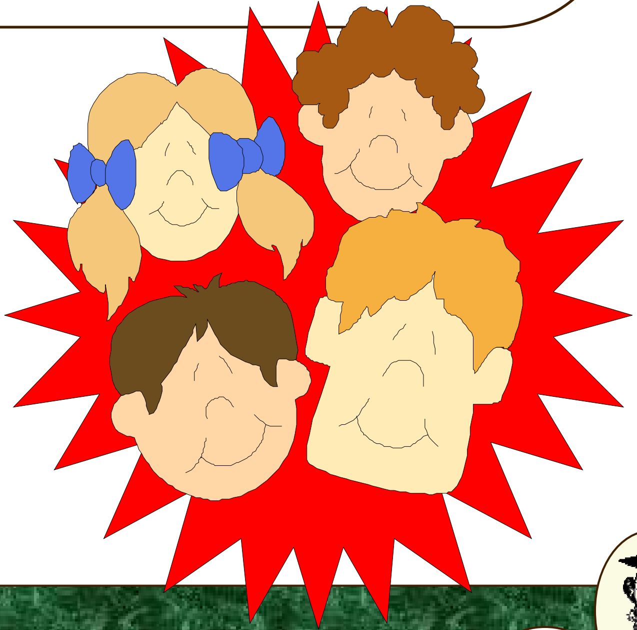
```
package Complex_Numbers is
    type Complex is private;
    function “+” (Left, Right  : Complex) return Complex;
    ... -- similarly “-”, “*” and “/”
    function Cartesian_To_Complex (Real,Imag:Float) return Complex;
    function Real_Part (X : Complex) return Float;
    function Imag_Part (X : Complex) return Float;
private
....
end Complex_Numbers;
```

Problem : some users of the above package need additional functionality - they need Polar Coordinates. Adding additional functions to the package requires all users to recompile, and adds functionality to ALL users, even those who don’t need it.

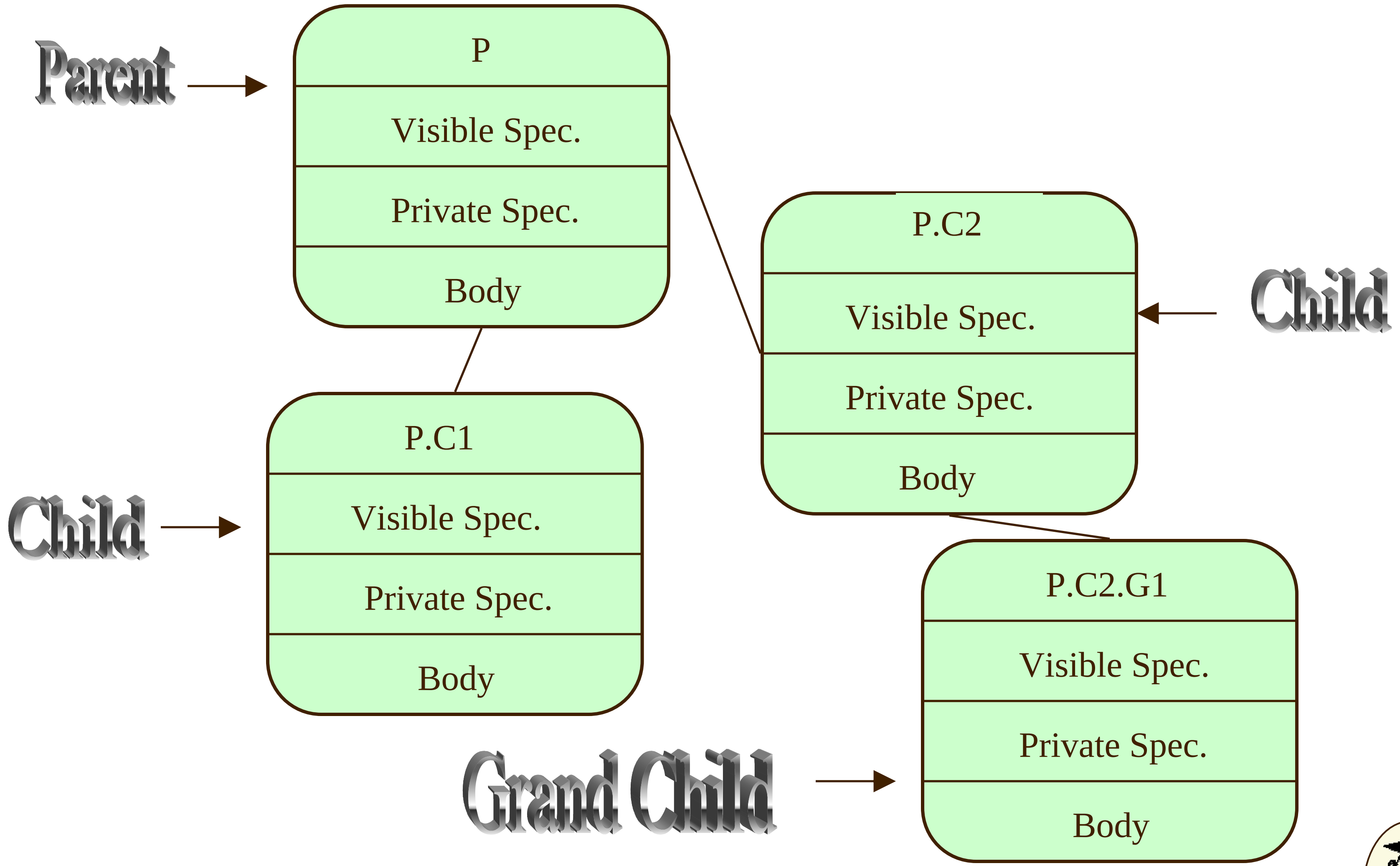
Public Children

```
package Complex_Numbers.Polar is
    procedure Polar_To_Complex (R, Theta : Float) return Complex;
    function "abs" (Right : Complex ) return Float;
    function Arg ( X : Complex) return Float;
end Complex_Numbers.Polar;
```

Solution - create a public child. Only those users who specify "with Complex_Numbers.Polar" gain the functionality. Other users are unaffected (and need not recompile).



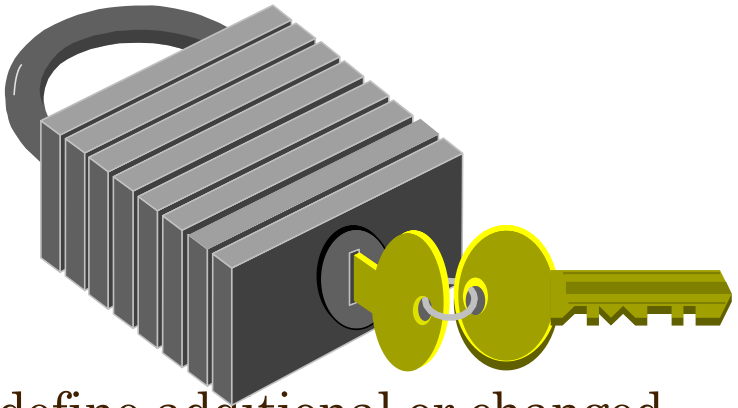
Visibility Rules of a Child Package



Meyer’s Modules

✓ **Closed Module**

❒ is executable, its behavior is well defined, it’s been V&V.

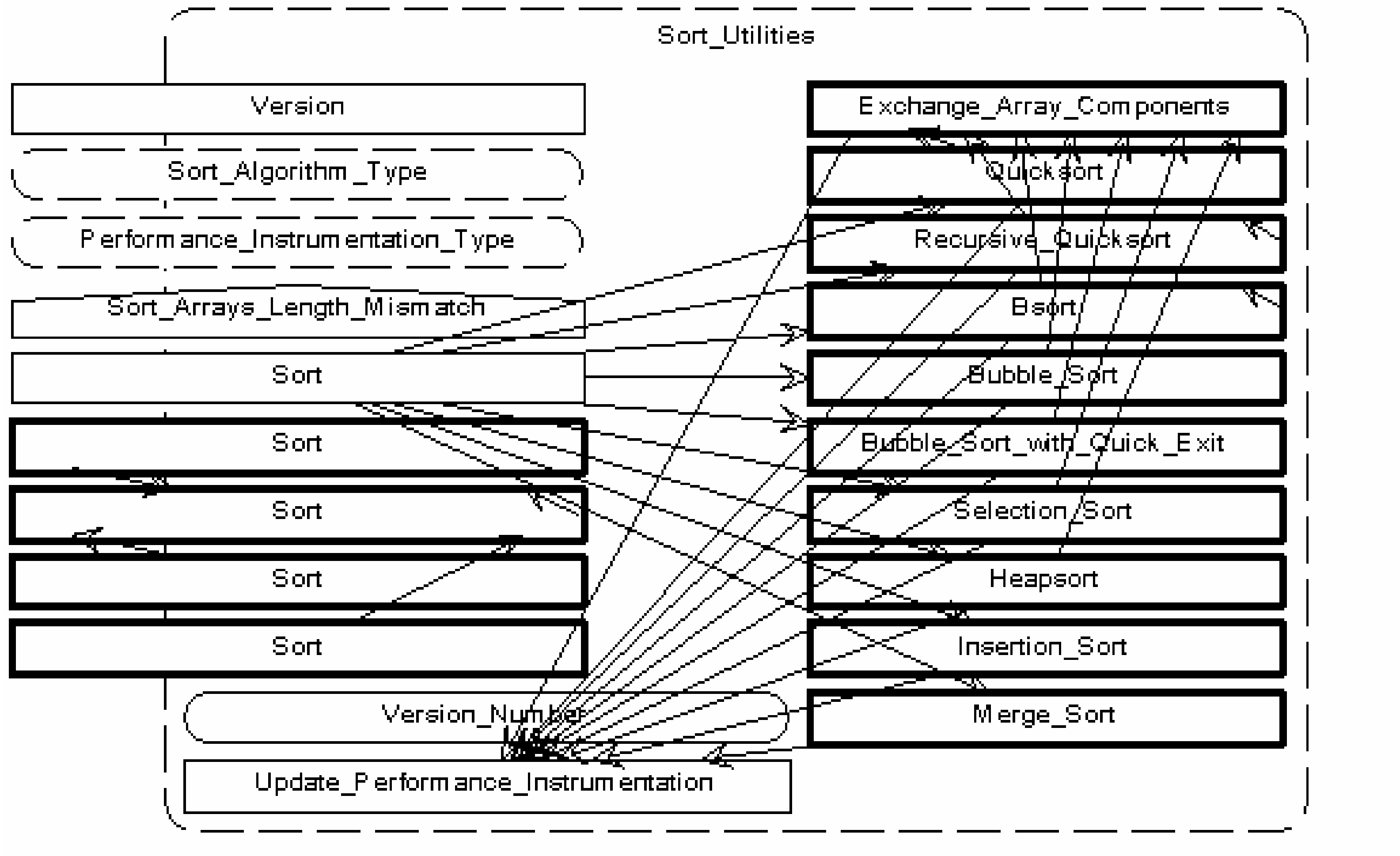


✓ **Open Module**

❒ is extendable, it’s possible to define additional or changed functionality with out changing the close module.

Closed Module + Open Module = Inheritance

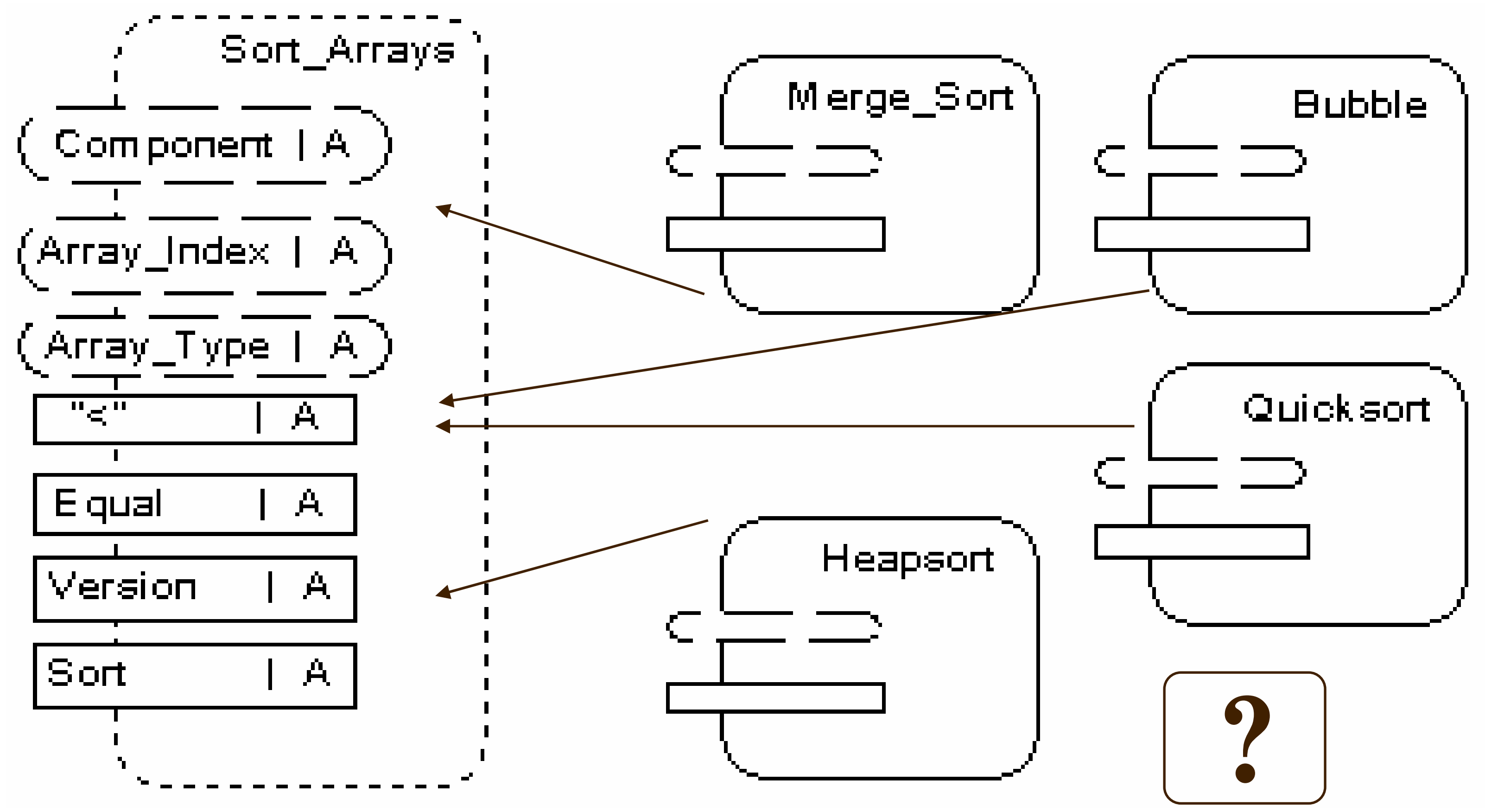
Closed Module



generic_package: Sort_Utilities
WARNING: Due to overloading, some calls to these objects may not be resolved correctly: Sort_Utilities.Sort,

Abstract Type

Open Module



Hierarchical Libraries

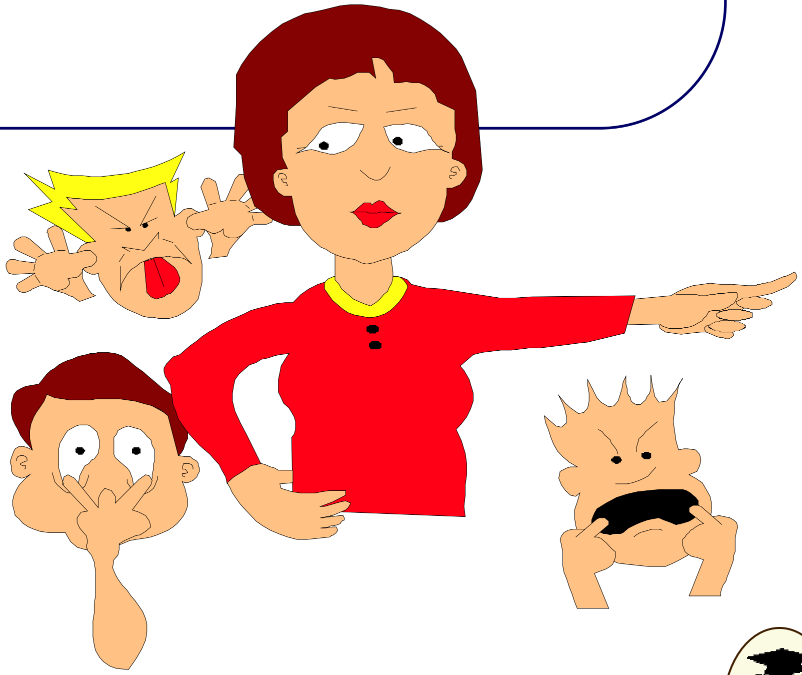
```
package Complex_Numbers is
    type Complex is private;
    function "+" (Left, Right  : Complex) return Complex;
    ... -- similarly "-", "*" and "/"
    function Cartesian_To_Complex (Real, Imag : Float) return Complex;
    function Real_Part (X : Complex) return Float;
    function Imag_Part (X : Complex) return Float;
private
....
end Complex_Numbers;
```

Problem - my package is becoming cluttered and unmanageable. I need to “modularize” it to make it easier to maintain.

Private Children

```
private package Complex_Numbers.Polar is
     procedure Polar_To_Complex (R, Theta : Float) return Complex;
     function “abs” (Right : Complex ) return Float;
     function Arg ( X : Complex) return Float;
end Complex_Numbers.Polar;
```

Solution - create a private child. Only the package body of Complex_Numbers is able to “with Complex_Numbers.Polar”. Users of the parent package Complex_Numbers are unable to access any private children.



Golden Rules for Packages

- A specification never needs to *with* its parent; it may *with* a sibling (if compiled first) except that a public child specification may not *with* a private sibling; it may not *with* its own child(it has not been compiled yet
- A body never needs to *with* its parent; it may *with* siblings (private or not); it may *with* its own child.
- A private child is never visible outside the tree rooted at its parent.
- The private part and body of any child can access the private part of its parent (and grandparent...).
- In addition, the visible part of a private child can also access the private part of its parent (and grandparent ...).
- A with clause for a child automatically implies with clauses for all its ancestors.
- A use clause for a unit makes the child units accessible by simple name (this only applies to child units for which there is also a with clause).

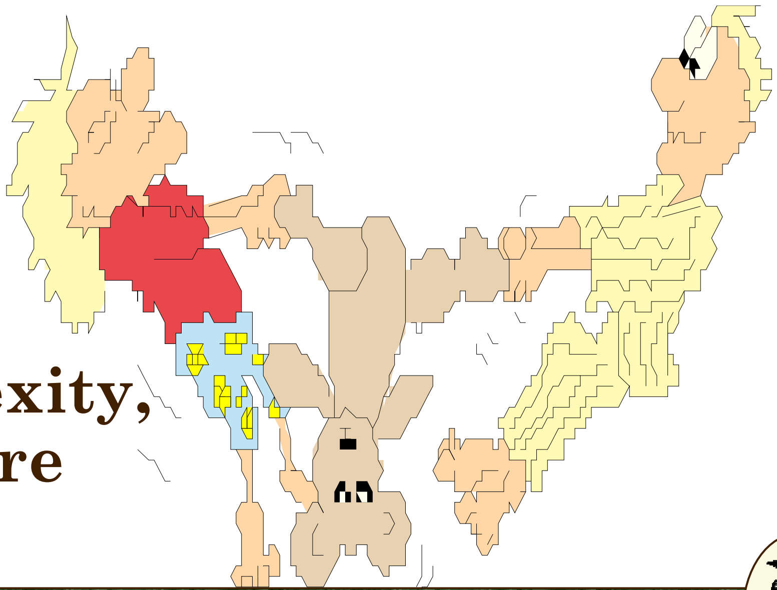
Summary - Child Packages

Public Children allow a large subsystem to be modularized from an user perspective. Users may "with" selected children as needed. Only necessary parts of the larger system need be "withed".

Private Children allow a large system to be functionally decomposed internally. Users of the system are unaware of the modularity.

Both uses reduce recompilation costs.

Both uses reduce complexity, and made the system more manageable.



Summary - OO features of Ada 95

ADTs

- allows non-changeable specification and predefined operations to be shared
- permits method extensions

Tagged types

- used to create parent class for inheritance
- allows code sharing between related classes
- permits attribute & method extensions
- allows class wide variables and dynamic dispatching

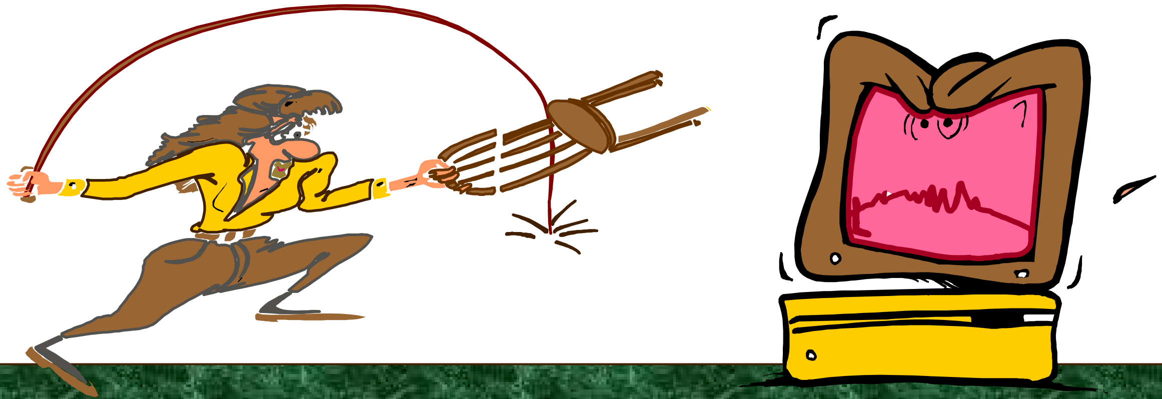
Summary - OO features of Ada 95

Child Packages

- Allows sharing of private views
- Manages namespace
- creates specific views or “schemas” for clients

Multiple Inheritance

- Adding existing properties to existing components



OOSE - Myths and Facts

- ✓ Myth - OO can be inserted into programming projects currently under development
- ✓ Myth - OO will provide reusable code, thus lowering my overall system cost
- ✓ Myth - OO will work with existing methodologies and systems

OOSE - Myths and Facts

- Fact - OO must be part of the overall lifecycle to recognize savings

- Fact - Reuse must be designed into code. It cost more to make reusable code.

- Fact - OO requires supporting CASE tools and extensive training to realize its full benefit

THE BOTTOM LINE

The insertion of OOSE into an organization requires time, money, planning, and commitment!!

Inserting OOSE
Step 1 -Planning and Pre-Insertion Stage

- ✓ Planning - note that Software Process Planning is a KPA (Key Process Area) under Level 2 of the Capability Maturity Model (CMM)
  - – Transition Plan
  - – Software Development Plan
    - who, what, when, how much, risks

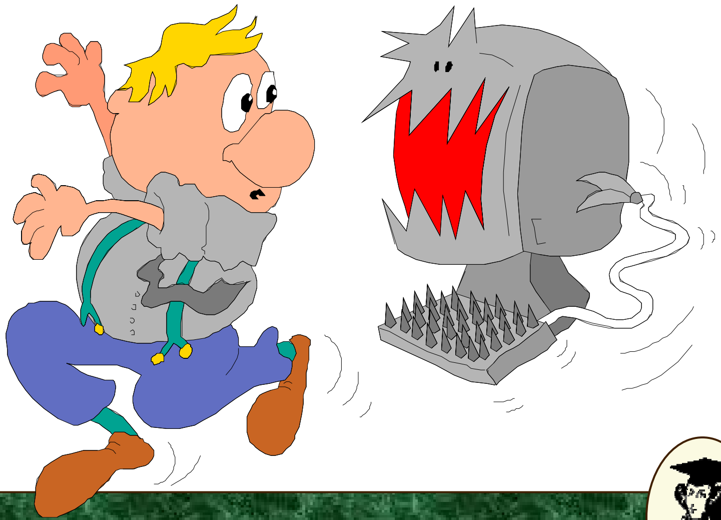
●Changing existing culture

–Understand the culture change

–Demonstrate management commitment

•provide time, tools, and strict enforcement

–Sell OOSE to those who will use it



Inserting OOSE
Step 2 - Insertion State

- Selecting an OO technique
  - Full life cycle, appropriate domain, target language
- Selecting a CASE tool
  - robust, adequate features
- Staffing and Organizing the project
  - Domain Analyst for reuse, OOP Guru, prototyping expert
- Training the team
  - Adequate time, dedicated time, uncompressed schedule
- Dealing with Legacy Systems
  - reverse-engineer or modify them to fit new OO systems
- Budgeting for Reuse

Inserting OOSE
Step 3 - Project Management

- Analyze, model and prototype
  - Tendency is to over-apply OO for first project
- Effective project tracking and controlling
  - objects, classes
- Define and document the development process
- Collect software metrics
- Inspect OO software products
  - review for quality, develop good metrics, inspect quality of documentation and graphics
- Integrate the documentation

Ada Predefined Packages

Ada.Finalization (Control Types)

Ada.Exceptions

Numeric, String, Wide String

Controlled Types

```
package Ada.Finalization is

type Controlled is abstract tagged private;

 procedure Initialize  (Object: in out Controlled);
procedure Adjust      (Object: in out Controlled);
 procedure Finalize    (Object: in out Controlled);
```

A type derived from Controlled can have an user-defined ***Adjust***, ***Finalize,*** and ***Initialize*** routines. Everytime an object of this type is assigned, released (via exiting scope or freeing up a pointer) or created, the appropriate routine will be called.

Controlled Type Example

```
with Ada.Finalization;

package Bathroom_Stalls is

type Stall is new Ada.Finalization.Controlled with private;

private

  type Stall is new Ada.Finalization.Controlled with
  record
    ID : integer;
  end record;

procedure Initialize  (Object : in out stall);

procedure Adjust      (Object : in out stall);

procedure Finalize    (Object : in out stall);

end Bathroom_Stalls;
```

Controlled Type Example

```
with Ada.Text_IO;
use Ada.Text_IO;
package body Bathroom_Stalls is

Stall_No                   : Natural   := 1;
Stalls_In_Use              : Natural   := 0;

procedure Initialize (Object : in out  Stall) is
  begin
    object.id := Stall_No;
    put ("In Initialize, object # " & integer'image(object.id) );
    Stall_No  := Stall_No + 1;
    Stalls_In_Use := Stalls_In_Use + 1;
    Put_Line(".  There are "&integer'image(Stalls_In_Use)
             & " People in stalls.");
end Initialize;

procedure Adjust (Object : in out Stall)
      renames Initialize;

procedure Finalize (Object : in out Stall) is
  begin
    put("In Finalize,   object # " & integer'image(object.id) );
        Stalls_In_Use := Stalls_In_Use - 1;
    Put_Line(".  There are "&integer'image(Stalls_In_Use)
             & " People in stalls.");
end Finalize;

end Bathroom_Stalls;
```

Controlled Type Example

```
with bathroom_stalls;
procedure stalls is

A,B:  bathroom_stalls.stall; --initialize called twice

begin
   declare
   D:  bathroom_stalls.stall; --initialize
   begin
      A := D;    --finalize, then adjust (initialize)
   end;
A := B;          --finalize, then adjust (initialize)
end;
```

```
In Initialize, object #  1.  There are 1 People in stalls.
In Initialize, object #  2.  There are 2 People in stalls.
In Initialize, object #  3.  There are 3 People in stalls.
In Finalize,   object #  1.  There are 2 People in stalls.
In Initialize, object #  4.  There are 3 People in stalls.
In Finalize,   object #  3.  There are 2 People in stalls.
In Finalize,   object #  4.  There are 1 People in stalls.
In Initialize, object #  5.  There are 2 People in stalls.
In Finalize,   object #  2.  There are 1 People in stalls.
In Finalize,   object #  5.  There are 0 People in stalls.
```

Summary

- ✓ Initialize, adjust and finalize executed automatically
- ✓ Used when data structures to initialize and finalize with a certain procedure
- ✓ User of data structure does not need to call set-up or clean-up procedures

Generics

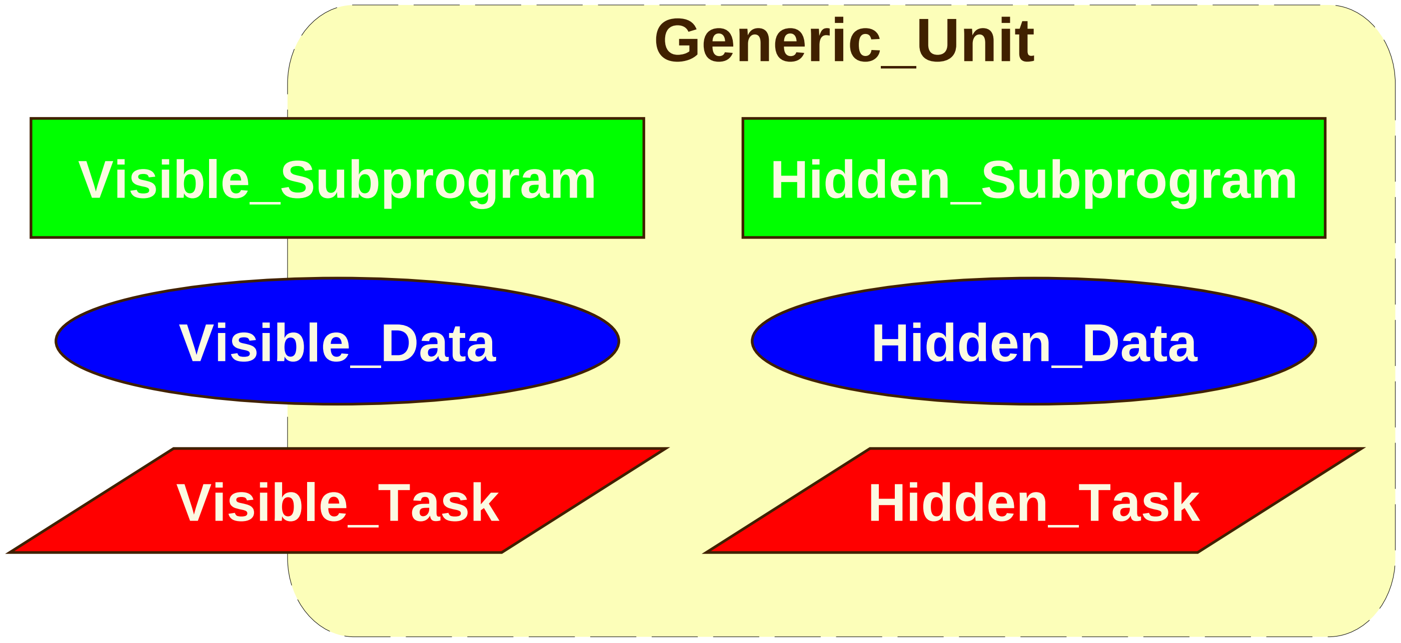
"A Generic is like a Twinkie without the filling"

Don Cutler
HQ SAC 1987

Generics Construction

- Design
- Instantiation
- Generic Parameters:
- Type, Objects, Subprograms, and Packages

Generics



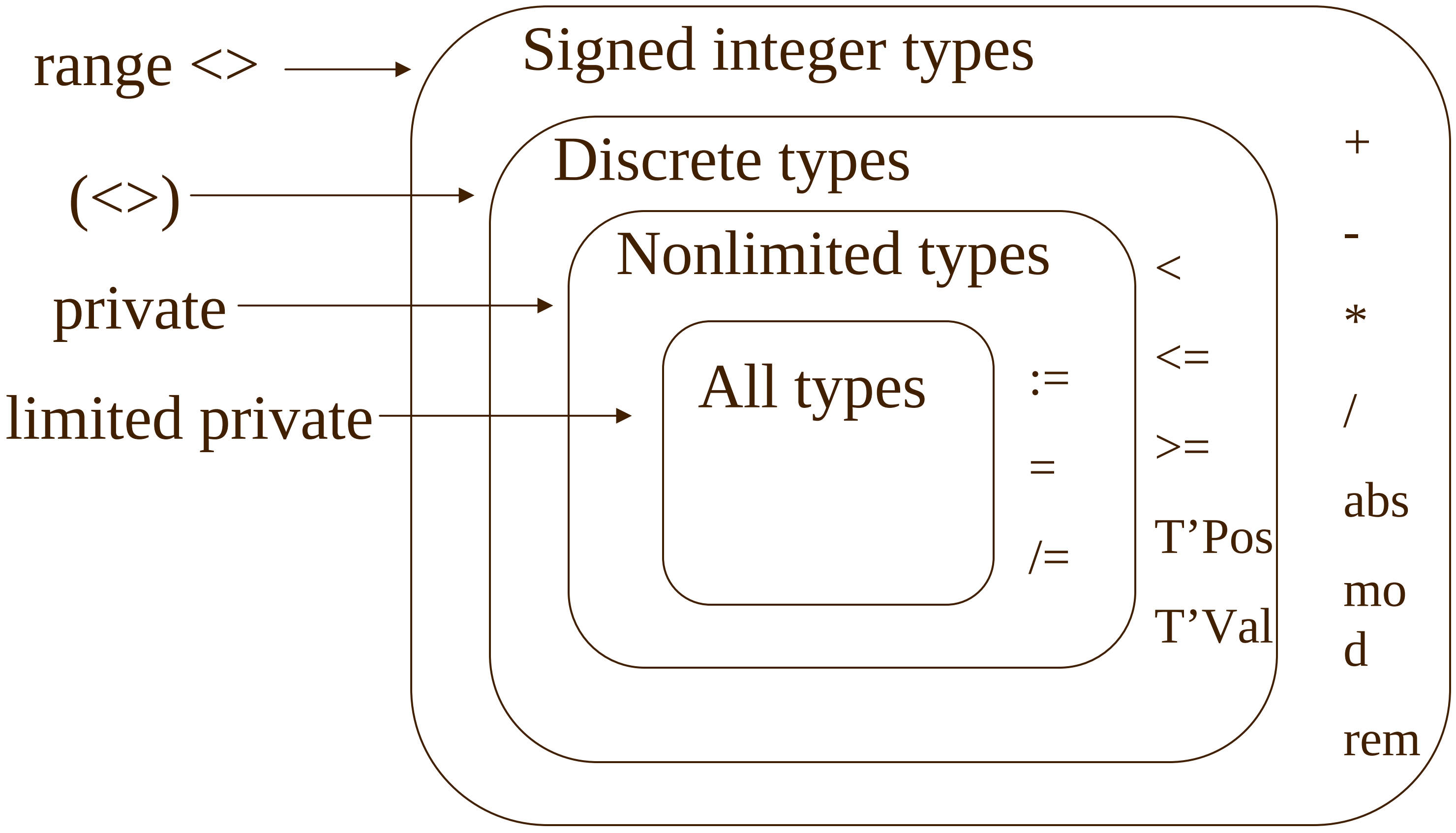
- Template of a subprogram or package
- Used to create actual program units -- “instantiation”
- Promote reusable software
- Improve productivity, maintainability

Generics

✓ Provides static polymorphism as opposed to the dynamic polymorphism

✓ Static is intrinsically more reliable but usually less flexible.

Generic Types *(Formal parameters)*



Generics

Generic procedure for selection sort

```
-- Generic procedure for selection sort.
generic
  type Index is (<>);
  type Item is private;
  type Vector is array(Index range <>) of
Item;
  with function "<"(Left, Right: Item) return
Boolean is <>;
procedure SelectionSort(A: in out Vector);
```

Generics

Generic procedure for selection sort

```
procedure SelectionSort(A: in out Vector) is
  Min: Index;
  Temp: Item;
begin
  for I in A'First .. Index'Pred(A'Last) loop
      Min := I;
      for J in I .. A'Last loop
        if A(J) < A(Min) then Min := J; end if;
      end loop;
      Temp := A(I); A(I) := A(Min); A(Min) := Temp;
  end loop;
end SelectionSort;
```

Generics *generic procedure for selection sort*

```
with SelectionSort;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Sort is
  type Point is
    record
      X, Y: Float;
    end record;
  type  Point_Vector is array(Character range <>) of Point;
  function "<"(Left, Right: Point) return Boolean is
  begin
    return (Left.X < Right.X) or else
           ((Left.X = Right.X) and then (Left.Y < Right.Y));
  end "<";
  procedure Point_Sort is new SelectionSort(
    Character, Point, Point_Vector);
  A: Point_Vector := ((10.0,1.0), (4.0,2.0), (5.0,3.4), (10.0,0.5));
begin
  Point_Sort(A);
  for I in A'Range loop
    Put(A(I).X,5,2,0);
    Put(A(I).Y,5,2,0);
    New_Line;
  end loop;
end Sort;
```

Generics

Generic formal package . . .

```
with Ada.Strings.Bounded;
generic
    with  package Bounded_Length_Instance is
    new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
type String_List_Type is
    array (positive range <>) of
    bounded_Length_Instance.Bounded_String;

procedure Pack_Lines (Lines : in out String_List_Type);
pragma Preelaborate (Pack_Line);
```

Generics

Generic formal package . . .

```
with Ada.Strings.Bounded;
package Bounded_80 is
   new Ada.Strings.Bounded.Generic_Bounded_Length
   (Max => 80);
with Bounded_80;
package Bounded_80_Lists  is
   type Bounded_80_List_Type  is
   array (Positive range <>) of
   Bounded_80.Bounded_String;
end Bounded_80_Lists;
```

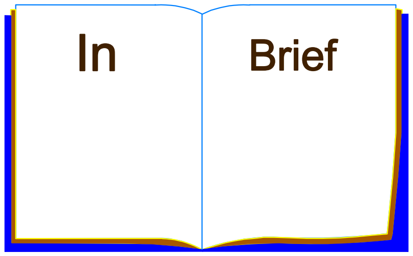
Generics

Generic formal package . . .

```
with Bounded_80, Bounded_80_Lists, Pack_Lines;
procedure Pack_Lines_80 is
   new Pack_Lines
   (Bounded_Length_Instance => Bounded_80,
    String_List_Type =>
   Bounded_80_Lists.Bounded_80_List_Type);
```

Pack_Line_80 is an ordinary procedure with formal parameter of type bounded_80_Lists.Bounded_80_List_Type.

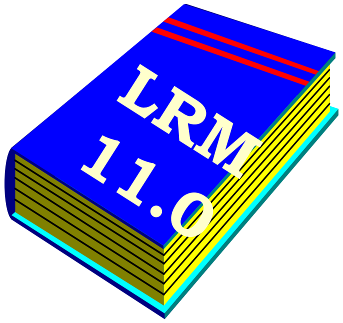
Summary



- A generic is a form of an Ada program unit that promotes reuse.
- Designing and creating generics is considered an advanced topic of the Ada language.

EXCEPTIONS

Hadnling all posible Erors!



Note : There are four errors in this slide.

Definition

* Ada was developed for real-time systems--systems which should never crash.

EXCEPTIONS ARE

Events that are infrequent, but not necessarily errors.

-- Perform some repair action and continue normal execution.

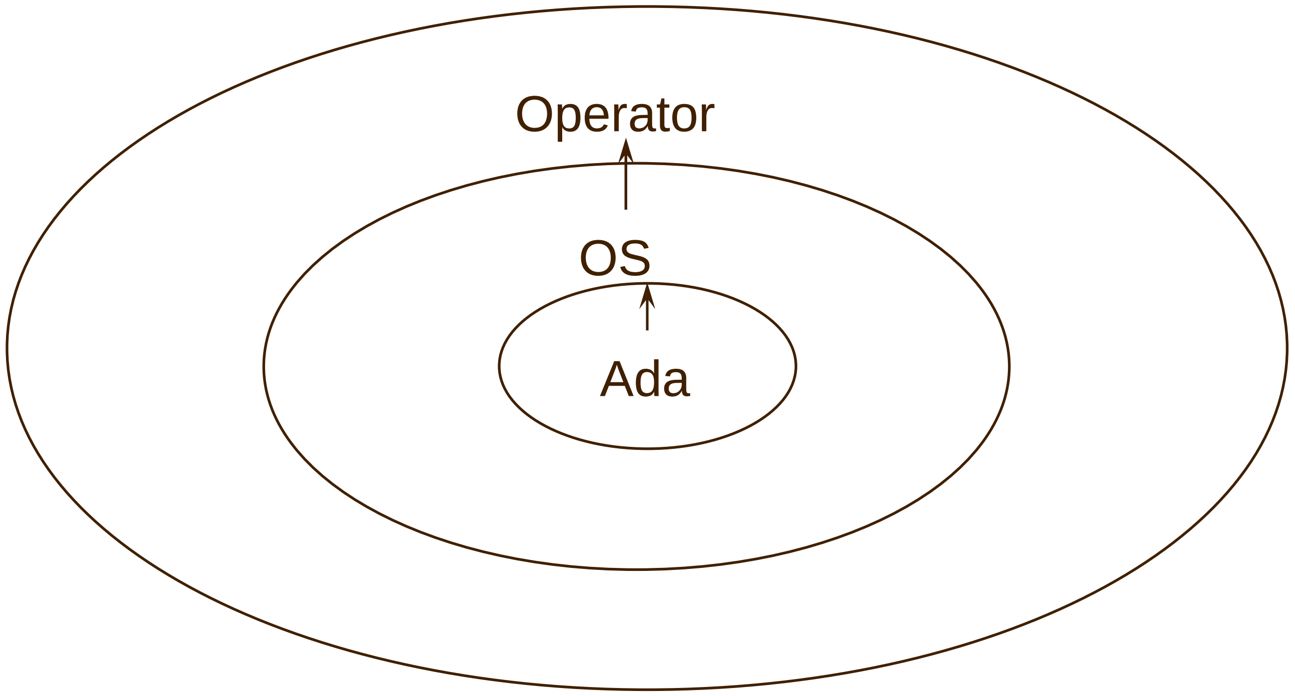
Events that are errors or terminating conditions

-- Execution passes to exception handler but does not return to point where exception was raised.

-- May restart sequence of actions under better conditions

Error Handling Levels

Traditional Approach:



Goals

* To deal with software errors without operator intervention

* To deal with unusual events that are not errors

Exception Handling Levels

This is the way
we try to handle
it in Ada

Example
(no exceptions)

```
package STACK_PACKAGE is
        type STACK_TYPE is limited private;
        procedure PUSH (STACK : in out STACK_TYPE;
                            ELEMENT : in ELEMENT_TYPE;
                            OVERFLOW : out BOOLEAN);
end STACK_PACKAGE;

with Ada.Text_IO, STACK_PACKAGE;
use Ada.Text_IO, STACK_PACKAGE;
procedure FLAG_WAVING is
     STACK : STACK_TYPE;
     ELEMENT : ELEMENT_TYPE;
     FLAG  : BOOLEAN;
begin
        PUSH(STACK, ELEMENT, FLAG);
        if FLAG then
              PUT ("Stack overflow");
        end if;
end FLAG_WAVING;
```

Using Exceptions

```
package STACK_PACKAGE is
   STACK_TYPE is limited private;
   STACK_OVERFLOW : exception;
   STACK_UNDERFLOW : exception;
   procedure PUSH (STACK : in out STACK_TYPE;
                   ELEMENT  : ELEMENT_TYPE);
         -- may raise Stack_Overflow
end STACK_PACKAGE;

with Ada.Text_IO, STACK_PACKAGE;
procedure MORE_NATURAL is
   STACK     : STACK_TYPE;
   ELEMENT : ELEMENT_TYPE;
begin
    STACK_PACKAGE.PUSH(STACK, ELEMENT);
exception
   when STACK_OVERFLOW =>
       Ada.Text_IO.PUT ("Stack overflow");
end MORE_NATURAL;
```

Defining Some Terms

* EXCEPTION

  The name attached to a situation that prevents completion of an action. (e.g. CONSTRAINT_ERROR is the name attached to the violation of a constraint).

* RAISING AN EXCEPTION

  Telling the invoker of an action that the corresponding error situation has occurred.

* HANDLING AN EXCEPTION

  Executing some actions in response to this occurrence.

Exceptions

* An exception has a name
  - may be predefined
  - may be declared
* The exception is raised
  - may be raised implicitly by run time system
  - may be raised explicitly by the *raise* statement
* The exception is handled
  - exception handler may be placed in any frame
  - exception propagates until handler is found
  - if no handler anywhere, control is returned to the operating system (like in other systems)

NOTE: A frame is the executable part surrounded by begin - end

Predefined Exceptions

* CONSTRAINT_ERROR

  Violation of range, index, or discriminant constraint...

* NUMERIC_ERROR (obsolete - now CONSTRAINT_ERROR)

  Execution of a predefined numeric operation cannot deliver a correct result

* PROGRAM_ERROR

  Attempt to access a program unit which has not yet been elaborated

* STORAGE_ERROR

  storage allocation is exceeded...

* TASKING_ERROR

  exception arising during inter-task communication

Predefined Exceptions

CONSTRAINT_ERROR:

1) Range constraint violation

   Example:

```
subtype SHORT_INTEGER is   INTEGER range - 128..127;
COUNTER : SHORT_INTEGER := 128;
```

2) Index constraint violation

   Example:

```
type N_ARRAY is array (1..100) of INTEGER;
An_ARRAY : N_ARRAY;
begin
     AN_ARRAY(0) :=1;
end;
```

3) Discriminant constraint violation

   Example:

```
type P_STRING (LENGTH : POSITIVE := 256) is
record
     DATA : STRING(1 .. LENGTH);
end record;
FILENAME : P_STRING(0);
```

Predefined Exceptions

CONSTRAINT_ERROR:

4) Attempt to use a variant record's component that does not exist

EXAMPLE:

```
type OUTPUT_FORMATS is  (CHAR_DATA, INTEGER_DATA);
type OUTPUT_DATA (KIND : OUTPUT_FORMATS) is
     record
        case KIND is
          when CHAR_DATA       => CHAR_VALUE : CHARACTER;
          when INTEGER_DATA => INTEGER_VALUE : INTEGER;
        end case;
   end record;
OUTPUT : OUTPUT_DATA (CHAR_DATA);
begin
              INT_IO.PUT(OUTPUT.INTEGER_VALUE);
end;
```

5) Attempt to dereference a null pointer

Predefined Exceptions

Numeric_Error (now Constraint_Error):

1) Division by zero

Example:

```
        QUOTIENT, DIVIDEND, DIVISOR : FLOAT;
begin
         DIVIDEND := 1000.0;
         DIVISOR := 0.0;
        QUOTIENT := DIVIDEND / DIVISOR;
end;
```

2) A predefined operation cannot deliver a correct result

Example:

```
        RESULT : INTEGER;
                  -- Assume a 16-bit integer.
         RESULT := 2 ** 32;
```

Predefined Exceptions

3) If a unit is not elaborated before it is referenced.

EXAMPLE:

```
declare
    function CALCULATE (X : INTEGER) return INTEGER;

    X : INTEGER := CALCULATE (2);

    function CALCULATE (X : INTEGER) return INTEGER is
    begin
        return 2**X;
    end CALCULATE;
begin
    null;
end;
```

Predefined Exceptions

Storage_Error:

1) If a collection's storage allocation is exhausted

```
EXAMPLE:
declare
        type PERSONNEL_DATA;
        type DATA_POINTER is
          access PERSONNEL_DATA;
        type PERSONNEL_DATA is
            record
                NAME : STRING(1 .. 30);
                NEXT : DATA_POINTER;
       end record;
FIRST, LAST : DATA_POINTER;
```

Storage_Error:

```
begin
      FIRST := new PERSONEL_DATA;
      LAST := FIRST;
      loop
        LAST.ALL.NEXT := new
                   PERSONNEL_DATA
        LAST := LAST.ALL.NEXT;
      end loop;
end;
```

Predefined Exceptions

2) If storage is not sufficient for elaboration of a declaration or call of a subprogram.

EXAMPLE:

```
procedure Hog is
      type PERSONNEL_DATA
          (LENGTH : POSITIVE := 30) is  --now 30, but can grow
      record
            NAME : STRING(1 .. LENGTH);  --have to reserve room for WORST case
      end record;
   ME : PERSONNEL_DATA;
   YOU : PERSONNEL_DATA;
begin
      null;
end;
```

3) If a task's storage allocation is exceeded

Predefined Exceptions

TASKING_ERROR :

1) Attempt to rendezvous with a dead task, or

2) Called task is aborted during rendezvous

These predefined exceptions may also be raised by the raise statement.

Raising Exceptions

Three parts of explicitly raising exception:

$\longrightarrow$ * Declaring the exception

* Defining the exception handler

* Raising and handling the exception

Declaring Exceptions

exception_declaration ::= identifier_list : exception;

Exceptions may be declared anywhere an object declaration is appropriate

* However, an exception is not an object
  - may not be used as subprogram parameter, record or array component
  - has same scope as an object, but its effect may extend beyond its scope

Example:

```
procedure Calculation is
     SINGULAR : exception;
     OVERFLOW : exception;
     UNDERFLOW : exception;
begin
-- sequence of statements
end CALCULATION;
```

Example

```
function SUBSTRING(ORIGINAL : STRING;
                   FROM : POSITIVE;
                   SIZE : POSITIVE) return STRING is
   ORIGINAL_LENGTH : POSITIVE;
   INDEX_ERROR : exception;
begin
.
.
.
end SUBSTRING;
```

Exceptions

Three parts to explicitly raised exception:

*   Declaring the exception

$\longrightarrow$ *   Defining the exception handler

*   Raising and handling the exception

Defining an Exception Handler

* An exceptional condition is "caught" and "handled" by an exception handler
* Exception handler(s) may appear at the end of any frame (block, subprogram, package or task body)

```
begin
...
exception
... exception handler(s)
end;
```

* Form similar to case statement

```
exception_handler ::=
when exception_choice { |exception_choice} =>sequence_of_statements

exception_choice ::= exception_name | others
```

Exceptions

The following "frames" may contain handlers.

block stmt	subprogram	package body	task body
declare	**procedure** id **is**	**package body** id **is**	**task body** id **is**
declarations	declarations	declarations	declarations
begin	**begin**	**begin**	**begin**
statements	statements	statements	statements
exception	**exception**	**exception**	**exception**
exception	exception	exception	exception
handlers	handlers	handlers	handlers
end; end;	**end;**	**end;**	

If an exception is raised by the sequence of statements, the frame is searched for a handler for the exception.

Execution will ALWAYS leave the frame, whether or not a handler exists!!

Exceptions

If a handler is found for the exception, then the sequence of statements it contains is executed, and the frame is abandoned.

block statement	$\longrightarrow$	is finished
subprogram body	$\longrightarrow$	returns
package body	$\longrightarrow$	is elaborated
task body	$\longrightarrow$	is completed

Example

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
    AGE : POSITIVE
begin
    Ada.Text_IO.PUT("Enter age? ");
      INT_IO.GET(AGE);
      -- What happens if characters entered?
end MAIN;

with INT_IO, Ada.Text_IO;
procedure MAIN is
      AGE : POSITIVE;
begin
      Ada.Text_IO.PUT("Enter age? ");
      INT_IO.GET(AGE);
exception
      when Ada.Text_IO.DATA_ERROR =>
        Ada.Text_IO.PUT_LINE("Illegal age.Rerun program.");
end MAIN;
```

Example

```
with INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
use INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
procedure RANK_ORDER is
    FIRST, SECOND : TEST_SCORES;
begin
    PUT("Enter first test score");
    GET(FIRST);
    PUT("Enter second test score");
    GET(SECOND);  -- Assume that we input a letter
    if SECOND_IS_GREATER(FIRST, SECOND) then
            SWAP(FIRST, SECOND);
    end if;
end RANK_ORDER;
```

The letter will not get into SECOND. What was in SECOND previously will be in SECOND when the exception is propagated.

Example

```
with INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
use INT_IO, Ada.Text_IO, UTILITY_PROGRAMS;
procedure RANK_ORDER is
     FIRST, SECOND : TEST_SCORES;
begin
     PUT("Enter first test score");
     GET(FIRST);
     PUT("Enter second test score");
     GET(SECOND);        --  Assume that we input a letter
     if SECOND_IS_GREATER(FIRST, SECOND) then
          SWAP(FIRST, SECOND);
     end if;
exception
     when DATA_ERROR =>
          PUT ("Invalid response--using zero as a default");
          FIRST := 0;
          SECOND := 0;
end RANK_ORDER;
```

NOTE: DATA_ERROR is a predefined exception. (LRM para 14.4)

Restrictions

* Exception handlers must be at the end of a frame
* Nothing but exception handlers may lie between exception and end of frame
* A handler may name any visible exception declared or predefined
* A handler includes a sequence of statements
  - response to exception condition
* A handler for others may be used
  - must be the last handler in the frame
  - handles all exceptions not listed in previous handlers of the frame (including those not in scope of visibility)
  - can be the only handler in the frame

Example

```
with FIX_IT, REPORT_IT, PUNT;
procedure WHATEVER is
    PROBLEM_CONDITION : exception;
begin
    <sequence of statements>
exception
    when PROBLEM_CONDITION =>
          FIX_IT;
    when CONSTRAINT_ERROR =>
          REPORT_IT;
    when others =>
          PUNT;
end WHATEVER;
```

Exceptions

Three parts to explicitly raised exception:

* Declaring the exception

* Defining the exception handler

$\longrightarrow$ * Raising and handling the exception

How Exceptions Are Raised

* Implicitly by run time system

  - predefined exceptions

Explicitly by raise statement

raise_statement ::=raise [exception_name];

-the name of the exception must be visible at the point of the raise statement

-a raise statement without an exception name is allowed only within an exception handler

Implicitly Raised Exceptions

```
function TOMORROW (DAY : DAYS_OF_THE WEEK)
      return DAYS_OF_THE_WEEK is
begin
      return DAYS_OF_THE_WEEK'SUCC(DAY);
exception
      when CONSTRAINT_ERROR =>
                return DAYS_OF_THE_WEEK'FIRST;
end TOMORROW;
```

frame

Explicitly Raised Exceptions

```
function TOMORROW (DAY:DAYS) is
    LAST_DAY : exception;
begin
    if DAY = DAYS'LAST then
            raise LAST_DAY;
    else
            return DAYS'SUCC(DAY);
    end if;
exception
    when LAST_DAY =>
            return DAYS'FIRST;
end TOMORROW;
```

Example

```
procedure WHATEVER is
    PROBLEM_CONDITION : exception;
    REAL_BAD_CONDITION : exception;
begin
    if PROBLEM_ARISES then
         raise PROBLEM_CONDITION;
    end if;
    if SERIOUS_CONDITION then
         raise REAL_BAD_CONDITION;
    end if;

exception
    when PROBLEM_CONDITION =>
         FIX_IT;
    when CONSTRAINT_ERROR =>
         REPORT_IT;
    when others =>
         PUNT;
end WHATEVER;
```

Effects of Raising an Exception

(1) Control transfers to exception handler at end of frame being executed (if handler exists)

(2) Exception is lowered

(3) Sequence of statements in exception handler is executed

(4) Control passes to end of frame

If frame does not contain an appropriate exception handler, the exception is propagated - effectively skipping steps 1 thru 3 and going straight to step 4

Exceptions

how can the same exception be raised again?

* Within a handler, the exception that caused transfer to the handler may be raised again by a normal raise statement (mentioning its name) or by a raise statement of the form

```
raise;
```

* Abandons the execution of the frame and propagates the corresponding exception.

Example

```
with Ada.Text_IO, MATRIX_OPS;
procedure EXCEPTIONAL is
   SINGULAR : exception
begin
   if MATRIX_OPS.MATRIX_IS_SINGULAR then
         raise SINGULAR
   end if;
   --    sequence of statements
exception
   when SINGULAR =>
         Ada.Text_IO.PUT("Matrix is singular"):
   when others =>
         Ada.Text_IO.PUT("Fatal error");
          raise;
end EXCEPTIONAL;
```

Example - No Exception handling

Example with no exception handling:

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
   AGE : POSITIVE;
begin -- main
   Ada.Text_IO.PUT("Enter Age:");
   INT_IO.GET(AGE);
end;
```

Example with no recovery

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
   AGE : POSITIVE;
begin
   Ada.Text_IO.PUT("Enter_Age:");
   INT_IO.GET(AGE);
exception
   when Ada.Text_IO.DATA_ERROR =>
         Ada.Text_IO.PUT("Illegal Age");
end MAIN;
```

Example with recovery

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
   AGE : POSITIVE;
begin
   Ada.Text_IO.PUT("Enter Age:");
   loop
         begin  -- A new block
            INT_IO.GET(AGE);
            exit; -- exit the loop if Age is OK
         exception
            when Ada.Text_IO.DATA_ERROR =>
                  Ada.Text_IO.PUT_LINE(
                        "Illegal age. Reenter:");
         end;
   end loop;
end MAIN;
```

Example
restricted number of retries

```
with INT_IO, Ada.Text_IO;
procedure MAIN is
      AGE : POSITIVE;
      NUMBER_OF_RETRIES : NATURAL := 0;
      TOO_MANY_MISTAKES : exception;
begin
      Ada.Text_IO.PUT("Enter Age:");
      loop
        begin
        INT_IO.GET(AGE);
        exit; -- exit the loop if Age is OK
exception
      when Ada.Text_IO.DATA_ERROR =>
         if  NUMBER_OF_RETRIES = 10 then
        raise TOO_MANY_MISTAKES;
        end if;
          NUMBER_OF_RETRIES := NUMBER_OF_RETRIES + 1;
        Ada.Text_IO.PUT_LINE ("Illegal age.Reenter:");
        end;
      end loop;
exception
      when TOO_MANY_MISTAKES =>
              Ada.Text_IO.PUT_LINE ("Too many mistakes made.);
end MAIN;
```

Exceptions

Exceptions may be raised by the elaboration of declarations. Then, the corresponding frame is NOT searched for a handler.

```
with Ada.Text_IO;
procedure BAD_STRING is
    NAME : STRING (0 .. 20);  -- Needs to begin with 1
begin
    null;
exception
    when others =>
            Ada.Text_IO.PUT_LINE("Exiting main program.");
end BAD_STRING;
```

NOTE- the above program will terminate, but the exception handler will not be invoked.

Catching declaration exceptions

```
with Ada.Text_IO;
procedure BAD_POSITIVE is
begin
        declare
                VALUE : POSITIVE := -1;
        begin
                null;
        end;
exception
        when others =>
                Ada.Text_IO.PUT_LINE("Exiting main program.");
end BAD_POSITIVE;
```

No recovery if error

```
with SEQUENTIAL_IO;
use SEQUENTIAL_IO;
procedure OPERATE (NAME : STRING) is
   FILE : FILE_TYPE;
begin
   OPEN(FILE, INOUT_FILE, NAME);

   -- Process the file (This could "bomb" and leave the  file open.)

   CLOSE(FILE);
end OPERATE;
```

Giving code its' Last Wishes

```
with SEQUENTIAL_IO;
use SEQUENTIAL_IO;
procedure SAFE_OPERATE(NAME : in STRING) is
    FILE : FILE_TYPE;
begin
    OPEN(FILE, INOUT_FILE, NAME);

    -- Operate on the file (may raise an exception)

    CLOSE(FILE);
exception
    when others =>
      CLOSE(FILE);
        raise;   --propagate the exception
end SAFE_OPERATE;
```

Exceptions

* When an exception is raised within the sequence of statements of a frame, the execution of this sequence of statements is always abandoned.

What happens next depends on the presence or <u>absence </u>of appropriate exception handlers.

-- Subprogram body =>

Same exception is raised implicitly at the point of call of the subprogram (except for a main program).

Absence

-- Block statement => between "begin" and "end"

Same exception is raised within the frame containing the block statement.

* In either case we say that the exception is "propagated"

How Exceptions Can Be Useful

* Normal processing could continue if

- the cause of exception condition can be "repaired"
- an alternative approach can be used
- the operation can be retired

* Degraded processing could be better than termination

- for example, safety-critical systems

* If termination is necessary, "clean-up" can be done first

Example

```
-- With no exception handlers, control passes to
-- surrounding frame.
with Ada.Text_IO;
procedure GET_NUMBERS is
    type NUMBERS is range 1 .. 100;
    package NUM_IO is new
        Ada.Text_IO.INTEGER_IO (NUMBERS);
    A_NUMBER : NUMBERS;
begin
    loop
        NUM_IO.GET (A_NUMBER);
        Ada.Text_IO.NEW_LINE;
        Ada.Text_IO.PUT("The number is");
        NUM_IO.PUT(A_NUMBER);
        Ada.Text_IO.NEW_LINE;
    end loop;
end GET_NUMBERS;
```

Example

```
-- With an exception handler, you retain control.
with Ada.Text_IO;
procedure GET_NUMBERS is
    type NUMBERS is range 1 .. 100;
    package NUM_IO is new
        Ada.Text_IO.INTEGER_IO (NUMBERS);
    A_NUMBER : NUMBERS;
begin
    loop
            NUM_IO.GET (A_NUMBER);
            Ada.Text_IO.NEW_LINE;
            Ada.Text_IO.PUT("The number is");
            NUM_IO.PUT(A_NUMBER);
            Ada.Text_IO.NEW_LINE;
    end loop;
exception
    when Ada.Text_IO.DATA_ERROR =>
        Ada.Text_IO.SKIP_LINE;
        Ada.Text_IO.PUT_LINE ("That was a bad number");
end GET_NUMBERS;
```

Example

```
-- Unrestricted number of retries
begin
    loop
        begin
            NUM_IO.GET (A_NUMBER);
            EXIT;
        exception
            when Ada.Text_IO.DATA_ERROR =>
              Ada.Text_IO.SKIP_LINE;
             Ada.Text_IO.PUT_LINE (
                 "Bad number, try again");
        end;
    end loop;
    Ada.Text_IO.NEW_LINE;
    Ada.Text_IO.PUT ("The number is");
    NUM_IO.PUT (A_NUMBER);
    Ada.Text_IO.NEW_LINE;
end GET_NUMBERS;
```

Package Ada.Exception

```
package Ada.Exceptions is
        type Exception_Id is private;
        Null_Id : constant Exception_Id;
        function Exception_Name(Id : Exception_Id) return String;
        type Exception_Occurrence is limited private;
        type Exception_Occurrence_Access is access all Exception_Occurrence;
        Null_Occurrence : constant Exception_Occurrence;
        procedure Raise_Exception(E : in Exception_Id; Message : in String := "");
        function Exception_Message(X : Exception_Occurrence) return String;
        procedure Reraise_Occurrence(X : in Exception_Occurrence);
        function Exception_Identity(X : Exception_Occurrence) return Exception_Id;
        function Exception_Name(X : Exception_Occurrence) return String;
                -- Same as Exception_Name(Exception_Identity(X)).
        function Exception_Information(X : Exception_Occurrence) return String;
        procedure Save_Occurrence(Target : out Exception_Occurrence;
                          Source : in Exception_Occurrence);
        function Save_Occurrence(Source : Exception_Occurrence)
                          return Exception_Occurrence_Access;
private
        ... -- not specified by the language
```

Example Package Ada.Exception

```
With Ada.exceptions;
Q : aliased Queue;
Ex1, Ex2, Ex3, Ex4 : exception;
procedure P1 is
    begin
            Ada.Exceptions.Raise.Exception (Ex1’identity,”P1” &
                    integer’image(13));
    exception
            when E: others => Put((13,Save.Occurrence(E), Q);
end P1;
…….
End Test;
```

Example Package Ada.Exception

Retrieved exception occurrences

```
…....
begin
    P1;
    while not Empty(Q) loop
    begin
            Ada.Exceptions.ReRaise.Occurrence(Get(Q’Access).
                      Occurrence.all);
    exception
            when E: others => Put_Line(Exception_Information(E);
    end;
    end loop;
end Test_Retrieve;
```

Summary

* Exceptions are events considered as errors or terminating conditions
* Execution of the current frame is always abandoned and control passes to an exception handler
* The handler may restart sequence of statements under better conditions by nesting frames in a loop

Tasking and Task Types

- ✓ Covered in “Real-Time” tutorial tomorrow