Dynamic Selection

An access type can refer to a subprogram; an access-to-subprogram value can be created by the `Access` attribute. A subprogram can be called using this pointer. This allows you to include a pointer to a routine inside of a record, or as a parameter. This is known as a **callback**.

```ada
type Trig_Function is access function (F : Float) return Float;

T : Trig_Function;
X, Theta : Float;

T := Sin’Access;
X := T(Theta); -- implicit dereferencing. SHOULD NOT BE USED!

X := T.all (Theta); -- explicit dereferencing. THIS IS PREFERRED.
```

**T can point to functions (such as Sin, Cos and Tan) that have a matching parameter list. Functions must have matching return types.**
“If” Statement
Format

<if_statement> ::= if <condition> then <sequence_of_statements>
{elsif <condition> then <sequence_of_statements>}
|else <sequence_of_statements>| end if;
Conditional Expressions

Expression where results are True or False.

Qualifiers

\[ = \ /\ = \ > \ < \ >= \ <= \ in \ not \ in \]

Boolean Operators

\[ \text{and} \ \ \text{or} \ \ \text{xor} \ \ \text{not} \]
Examples

type DAY is (Monday, Tuesday, …, Sunday);

   TODAY, TOMORROW : DAY;

if TODAY = Sunday then -- simple if-then
   TOMORROW := Monday;
end if;

if TODAY = Sunday then -- if-then-else
   TOMORROW := Monday;
else
   TOMORROW := DAY ' SUCC(TODAY);
end if;
Nested IF Example

type DIRECTION is (Left, Right, Back);

ORDER : DIRECTION;

if ORDER = Left then
  TURN_LEFT;
else
  if ORDER = Right then
    TURN_RIGHT;
  else
    if ORDER = Back then
      TURN_BACK;
    end if;
  end if;
end if;
ELSIF Construct

Reduces level of nesting in a program

Example:
If ORDER = Left then
    TURN_LEFT;
elsif ORDER = Right then
    TURN_RIGHT;
elsif ORDER = Back then
    TURN_BACK;
end if;
More IF Examples

if TODAY = FRIDAY then
   Num_Weeks := Num_Weeks + 1;
   Num_Left := Num_Left - 1;
   Ada.Text_IO.Put_Line("TGIF!!!");
end if;

if SCORE >= 90 then LETTER GRADE := 'A';
elsif SCORE >= 80 then LETTER GRADE := 'B';
elsif SCORE >= 70 then LETTER GRADE := 'C';
elsif SCORE >= 60 then LETTER GRADE := 'D';
else LETTER GRADE := 'F';
end if;
Short Circuit Control Forms

if COUNTER < TABLE'LAST then
    if RESULT(COUNTER) /= 0 then
        -- some statements
    end if;
end if;

if COUNTER < TABLE'LAST and then
    RESULT(COUNTER) /= 0 then
    -- some statements
end if;

if MY_PTR = null or else MY_PTR.ALL.ITEM = 0 then
    -- some statements
end if;

Sometimes, we run into the case of trying to use an if statement with an array, but the index might not be valid.

Use the short circuit control forms to guarantee safe checks: (and then, or else)
Why Avoid “not”

✓ “Not” can add unnecessary complexity
✓ you should use statements that there are no questions what is meant by a false out come

**BAD**
if not x = y or z = q
then
    something;
end if;

**GOOD (or BETTER)**
if  x = y or z = q
then
    null;
else
    something;
end if;
Case Statement
Format

case <discrete_expression> is
  when <choice> { | <choice> } =>
    <sequence_of_statements>
  {when <choice> { | <choice> } =>
    <sequence_of_statements>}</
end case;
Expression

Expression

when Choice 1
when Choice 2
when Choice 3
when others

<discrete_expression> | <discrete_range> | others
“others”

✓ The rest (or all) receive this action or value

✓ Used in case statements, setting values, and exception handling

✓ Must be last alternative
Rules for ‘case’

- Expression must be discrete
- Choices must be mutually exclusive and exhaustive
- When "others" is used, it is only allowed as the last alternative
- There is no “fall through” in Ada. Each case, when complete, goes to the end of the case statement, NOT the next option
Alternatives must be:

• **discrete**
• **exhaustive**, and
• **mutually exclusive**.

CASE Statement Examples

type Grade_Type is ('A','B','C','D','F','I');

case LETTER_GRADE is
   when 'A' => -- a sequence of statements
   when 'B' => -- a sequence of statements
   when 'C' => -- a sequence of statements
   when 'D' => -- a sequence of statements
   when 'F' => -- a sequence of statements
   when 'I' => -- a sequence of statements
end case;

case LETTER_GRADE is
   when 'A' | 'B' | 'C' => -- a sequence of statements
   when 'D' => -- a sequence of statements
   when 'F' => -- a sequence of statements
   when others => -- a sequence of statements
end case;

case LETTER_GRADE is
   when 'A' .. 'C' => -- a sequence of statements
   when 'D' => -- a sequence of statements
   when 'F' => -- a sequence of statements
   when others => -- a sequence of statements
end case;
procedure SWITCH (HEADING :in out DIRECTION) is
begin
  case HEADING is
    when NORTH => HEADING := SOUTH;
    when EAST  => HEADING := WEST;
    when SOUTH => HEADING := NORTH;
    when WEST  => HEADING := EAST; --what happens if I add SW?
    when others => HEADING := EAST; --what happens if I add SW?
  end case;
end SWITCH;

case NUMBER is  --assume NUMBER is an integer
  when 2 => DO_SOMETHING;
  when 3 | 7 | 8 => DO_SOMETHING_ELSE;
  when 9 .. 20 => DO_SOMETHING_RADICAL;
  when others => DO_OTHER_THINGS;
    Ada.Text_IO.PUT_LINE ("Others");
end case;
Better than “if” ???

```ada
procedure SWITCH (HEADING :in out DIRECTION) is
begin
    case HEADING is
        when NORTH => HEADING := SOUTH;
        when EAST  => HEADING := WEST;
        when SOUTH => HEADING := NORTH;
        when WEST  => HEADING := EAST;
    end case;
end case;
end SWITCH;
```

```ada
procedure SWITCH (HEADING :in out DIRECTION) is
begin
    if Heading = North then  Heading := South;
    elsif Heading = East then Heading := West;
    elsif Heading = South then Heading := North;
    else Heading := East;
    end if;
end SWITCH;
```
Summary of Case Statements

✔ Expression must be discrete

✔ Choices must be mutually exclusive and exhaustive

✔ When "others" is used, it is only allowed as the last alternative
ITERATIVE STATEMENTS
Iterative Statements (Loops)

✓ Basic Loop -- used to execute a sequence of statements up to an infinite number of times.

✓ For .. loop -- used to execute a sequence of statements a finite number of times.

✓ While .. loop -- used to execute a sequence of statements until a certain condition is met.
Iterative Statement Examples

loop
  <sequence of statements>
end loop;

for I in 1 .. 10 loop
  <sequence_of_statements>
end loop;

while I < 10 loop
  <sequence_of_statements>
end loop;
Basic Loop

Basic loop is infinite because in embedded systems sometimes we want to loop continuously.

```ada
loop
    <sequence of statements>
end loop;
```
Basic Loop

EXAMPLES:

loop
  GET (SAMPLE);
  PROCESS (SAMPLE);
end loop;

loop
  GET (ALTITUDE);
  if ALTITUDE < 1000 and not GEAR_DOWN then
    LIGHT_GEAR_DOWN_LAMP;
    SOUND_ALARM;
  end if;
end loop;
Exit Statement

Two Forms:

*Unconditional exit*

exit;

*Conditional exit*

exit when <boolean expression>;}
Basic Loop
‘exit’ Statement

loop
    ...
    if x = 20 then
        exit;
    end if;
end loop;

--or--
loop
    ...
    exit when x = 20;
end loop;
Basic Loop

EXAMPLE:
INDEX : integer := 1;
E : Float := 0.0;
TERM : Float := 1.0;
loop
  exit when INDEX := MAX_TERM;
  INDEX := INDEX + 1;
  TERM := TERM / FLOAT(INDEX);
  E := E + TERM;
end loop;
loop
  if INDEX = MAX_TERM then
    exit;
  else
    INDEX := INDEX + 1;
    TERM := TERM / FLOAT(INDEX);
    E := E + TERM;
  end if;
end loop;
Named Loops

OUTER:
  loop
    INNER:
      loop
        exit OUTER;
      end loop INNER;
    exit;
  exit;
end loop OUTER;
Exit Statement

exit; -- exits the innermost enclosing loop
exit Outer_loop; -- exits a named loop

```
OUTER_LOOP:

  MIDDLE_LOOP:

    INNER_LOOP:

      exit;

    exit OUTER_LOOP:

  exit when INDEX = MAX_TERM;
```
Basic Loop

**EXAMPLE:**

\[
X := \ldots
\]

**OUTER:**

loop

**INNER:**

loop

if \(X = 20\) then

exit OUTER;

end if;

exit INNER when \(X = 21\);

\(X := X + 2;\)

end loop INNER;

end loop OUTER;
Exit Statement

exit when INDEX = MAX_TERM;
-- exits innermost enclosing
-- loop when condition is true
exit OUTER_LOOP when INDEX = MAX_TERM
exits the named loop
when condition is true

OUTER_LOOP:

MIDDLE_LOOP:

INNER_LOOP:
exit;

exit WHEN INDEX = MAX_TERM;

To go from the
INNER_LOOP to the
OUTER_LOOP we could
use exit MIDDLE_LOOP
For Loop

for <loop index> in <start value> .. <stop value> loop
    <sequence of statements>
end loop;

<loop index> is an identifier. It is declared implicitly— it is NOT declared by the programmer. Its type depends on the type of <start value> and <stop value>. It is treated as a constant within the loop. It “disappears” after the loop

<start value> and <stop value> must be discrete expressions.
Control Variables

- ARE IMPLICITLY DECLARED
- MUST BE DISCRETE
- TAKE THEIR TYPE FROM THE DISCRETE RANGE
- ARE IN EXISTENCE ONLY UNTIL end loop
- CANNOT BE MODIFIED (LOCAL CONSTANT)
- ONLY SINGLE STEP INCREMENT (DECREMENT)
- CAN 'HIDE' A VARIABLE WITH SAME NAME
Example of Control Variables

for DAY in DAYS loop
    ...
end loop;

for COUNTER in reverse 1..10 loop
    ...
end loop;
For Loop

EXAMPLES:

for INDEX in 1..15 loop
  -- Index is type Universal INTEGER
  <SOS>
end loop;

for INDEX in reverse 1..15 loop
  -- Backwards
  <SOS>
end loop;
For Loop

EXAMPLE:

procedure MAIN is
  SUM : INTEGER := 0;
  I   : INTEGER := 10;
begin
  for I in 1..5 loop
    SUM := I + MAIN.I + SUM;
  end loop;
end MAIN;
procedure MAIN is
    type DAYS is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    subtype WEEKDAYS is DAYS range Mon .. Fri;
begin
    for TODAY in Mon .. Sun loop
        <sequence of statements>
    end loop;
    for TODAY in DAYS loop
        <sequence of statements>
    end loop;
    for TODAY in DAYS range Mon .. Fri loop
        <sequence of statements>
    end loop;
    for TODAY in WEEKDAYS loop
        <sequence of statements>
    end loop;
end;
For Loop

**EXAMPLE:**

```ada
with Ada.Text_IO;
procedure PRINT_ALL_VALUES is
  type COLORS is (RED, WHITE, BLUE);
  package COLOR_IO is new Ada.Text_IO.ENUMERATION_IO (COLORS);
  begin
    for INDEX in 1..5 loop
      null;
    end loop;
    for A_COLOR in COLORS loop
      COLOR_IO.PUT(A_COLOR);
      Ada.Text_IO.NEW_LINE;
    end loop;
  end PRINT_ALL_VALUES;
```

by the ASEET Team

Introduction to Ada
For Loop

with Ada.Text_IO;
procedure MAIN_PROGRAM is
    type INDEX_TYPE is range 1 .. 100;
    type CTR_TYPE is INTEGER range -10 .. 10;
    package INDEX_IO is new
        Ada.Text_IO.INTEGER_IO(INDEX_TYPE);
    package COUNTER_IO is new
        Ada.Text_IO.INTEGER_IO(CTR_TYPE);
    begin
        for INDEX in INDEX_TYPE'FIRST.. INDEX_TYPE'LAST loop
            Ada.Text_IO.PUT("The index number is ");
            INDEX_IO.PUT(INDEX);
            Ada.Text_IO.NEW_LINE;
        end loop;

        for COUNTER in CTR_TYPE'RANGE loop
            Ada.Text_IO.PUT("The counter number is ");
            COUNTER_IO.PUT(CTR_TYPE);
            Ada.Text_IO.NEW_LINE;
        end loop;
    end MAIN_PROGRAM;
For Loop Example

procedure MAIN is
    type INDEX is range 0 .. 200;
begin
    for I in reverse INDEX loop
        <sequence of statements>
    end loop;

    for I in reverse INDEX range 1 .. 50 loop
        <sequence of statements>
    end loop;
end MAIN;
Iterative Statement (While Loop)

while <boolean expression> loop
  <sequence of statements>
end loop;

while not Ada.Text_IO.END_OF_FILE(INPUT_FILE)
  loop
    ARGU_IO.GET(INPUT_FILE, ARGUMENT);
    Ada.Text_IO.PUT(ARGUMENT);
  end loop;

while INPUT < 0 or INPUT > 100 loop
  Ada.Text_IO.PUT_LINE("Bad Input - Try again");
end loop;
While Loop

EXAMPLES:

while NOT_DARK loop
    PLAY_TENNIS;
end loop;

TURN_ON_LIGHTS;
while COUNT <= 100 loop
    COUNT := COUNT+ 1;
    if EMERGENCY then
        exit;
    end if;
end loop;
While Loop Example

package INT_IO is new
    Ada.Text_IO.INTEGER_IO(INPUT_TYPE);
package ENUM_IO is new
    Ada.Text_IO.ENumeration_IO(DONE_TYPE);

while INPUT not in 1 .. 99 loop
    Ada.Text_IO.PUT_LINE("Bad Input - Try again.");
    INT_IO.GET(INPUT);
end loop;

while not DONE loop
    INT_IO.GET(INPUT);
    Ada.Text_IO.PUT("More input (true/false)?");
    ENUM_IO.GET(Done);
end loop;
 Nested Example

with Ada.Text_IO;
procedure COUNT_SMALL_LETTERS is
  CURRENT_LINE : STRING(1 .. 100);
  LENGTH : NATURAL;
  NO_SMALL_LETTERS : NATURAL := 0;
begin
  for LINE_NUMBER in 1 .. 10 loop
    Ada.Text_IO.GET_LINE(CURRENT_LINE, LENGTH);
    for CHAR_NUMBER in 1 .. LENGTH loop
      if CURRENT_LINE(CHAR_NUMBER) in 'a' .. 'z' then
        NO_SMALL_LETTERS :=
        NO_SMALL_LETTERS + 1;
      end if;
    end loop;
    Ada.Text_IO.PUT("There are ");
    Ada.Text_IO.PUT(NATURAL'IMAGE(NO_SMALL_LETTERS));
    Ada.Text_IO.PUT_LINE(" small letters");
  end loop;
end;
Summary

✓ LOOPING CONSTRUCTS

– BASIC LOOP
  • execute an infinite number of times

– FOR LOOP
  • execute a finite number of times

– WHILE LOOP
  • execute until a certain condition is met
Ada’s Private Parts

*What is essential is invisible to the eye.*

*Antoine de Saint-Exupery, The Little Prince*
Ada’s Private Parts

Information hiding -- Avoid revealing implementation details. The more that is hidden the more freedom you will enjoy in maintenance.

Private -- outside view, predefined “=” and “/=”

Limited Private -- outside view, only user defined operations (equality not defined!)
package B_R is
  type NUMBERS is range 0..99;
  procedure TAKE (A_NUMBER : out NUMBERS);
  function NOW_SERVING return NUMBERS;
  procedure SERVE (NUMBER : in NUMBERS);
end B_R;

package body B_R is
  SERV_A_MATIC : NUMBERS := 1;

  procedure TAKE (A_NUMBER : out NUMBERS) is
  begin
    A_NUMBER := SERV_A_MATIC;
    SERV_A_MATIC := SERV_A_MATIC + 1;
  end TAKE;

  function NOW_SERVING return NUMBERS
  is separate;
  procedure SERVE (NUMBERS : in NUMBERS) is separate;
end B_R;

with B_R; -- logical link to package specification
use B_R; -- direct visibility for infix notation
procedure ICE_CREAM is
  YOUR_NUMBER : NUMBERS;
begin
  TAKE (YOUR_NUMBER);
  loop
    if NOW_SERVING = YOUR_NUMBER
      then
        SERVE (YOUR_NUMBER);
        -- correct use of the abstraction
        -- wait for your number to be called
        exit;
    end if;
  end loop;
end ICE_CREAM;
Abusing the abstraction

with B_R; use B_R;
procedure ICE_CREAM is
  YOUR_NUMBER : NUMBERS;
begin
  TAKE (YOUR_NUMBER);
  loop
    if NOW_SERVING = YOUR_NUMBER then
      SERVE (YOUR_NUMBER);
      exit;
    else
      YOUR_NUMBER := YOUR_NUMBER - 1;
    end if;
  end loop;
end ICE_CREAM;

--BUT, nothing prevents me from modifying my number.
--This is an abuse of the “abstraction”
-- Solution? Use a private type to prevent changing
-- the value of objects of type NUMBERS

package B_R is

  type NUMBERS is private;

-- Only the type name is exportable; the data
-- structure has not been defined
-- Only operations below are defined

  procedure TAKE (A_NUMBER : out NUMBERS);
  function NOW_SERVING return NUMBERS;
  procedure SERVE (NUMBER : in NUMBERS);

private
  type NUMBERS is range 0..99;
end B_R;

-- PROBLEM - “assignment” is still defined, so I
-- can still abuse the abstraction

with B_R; use B_R;
procedure ICE_CREAM is
  YOUR_NUMBER : NUMBERS;
begin
  TAKE (YOUR_NUMBER);
  loop
    if NOW_SERVING = YOUR_NUMBER then
      SERVE (YOUR_NUMBER);
      exit;
    else
      YOUR_NUMBER := NOW_SERVING;
      -- just go to the front of the line!
    end if;
  end loop;
end ICE_CREAM;
Introduction to Ada

--CORRECT SOLUTION – use limited private type
--to prevent improper access

package B_R is

  type NUMBERS is limited private;

procedure TAKE (A_NUMBER : out NUMBERS);
function NOW_SERVING return NUMBERS;
procedure SERVE (NUMBER : in NUMBERS);

function "=" (LEFT, RIGHT : in NUMBERS) return BOOLEAN;
-- this operation must be added to allow the user
-- to compare two objects of a limited private type

private
  type NUMBERS is range 0..99;
end B_R;

with B_R; use B_R;

procedure ICE_CREAM is
  YOUR_NUMBER : NUMBERS;
  procedure GO_TO_DQ is separate;

begin
  TAKE (YOUR_NUMBER);
  loop
    if NOW_SERVING = YOUR_NUMBER then
      SERVE (YOUR_NUMBER);
      exit;
    else
      GO_TO_DQ;  --only other option ;=}
    end if;
  end loop;
end ICE_CREAM;

--To sum it up, “private” and “limited private”
--restrict the visibility and access to types and
--objects. This allows you to “Enforce your
--abstractions”
Exercise

✔ Modify your previous program.

✔ Change Age to Private. Pass Age between subprograms.

✔ Modify Age objects.

✔ Try adding and multiplying Ages;

✔ Change Age to Limited Private and *, / operations.
Object-Oriented Programming

- Hierarchical Library Units
- Tagged types
- Class Wide programming
- Dispatching
- Control
Benefits of OOD for Complex Systems

- An OOD approach facilitates
- Identifying and creating software modules with strong internal cohesion and weak external coupling with other modules.
- Identifying and creating Ada packages.
- Providing a direct correspondence between the physical world and the computer software.
Disadvantages of OOD for Complex Systems

✓ OOD may have negative space and timing impact.

✓ OOD requires replacing the traditional functional decomposition mindset.
What is an Object?

- A self-contained thing which is characterized by:
  - it's states
  - the actions it performs
  - the actions that can be performed upon it
  - an unique designation

- An Object is an instance of a Class (set, type, group)

- Views of an Object:
  - External perspective
  - Internal perspective

- An Object may be composed of one or more Objects
Ada Object

● In Ada, an Object is typically implemented as an:

● Abstract State Machine (ASM) package or generic package, with no exportable type. There is only one object, the state machine itself.

● Abstract Data Type (ADT) package or generic package, with exportable type. You can make as many instances as you need.
What is an ASM?

- An ASM package or generic package encapsulates:
  - A single, possibly complex Ada object of a single type, both typically hidden in the body
    Note: It may also encapsulate hidden auxiliary Ada objects and types.
  - Exported operations (e.g. subprograms or task entry points) that operate on this object.
    Note: It may also encapsulate hidden auxiliary operations.
  - Exceptions (error conditions) that are raised (and must be properly handled) when the operations cannot produce a valid result.
  - An ASM is represented by a singular noun (e.g., Account_Payable).
Buhr Sample 1
Abstract State Machine

Package
- Type
- Data
- Subprogram
  - Subprogram
    - Socket
Example of an Abstract State Machine

with Aircrew_Characteristics;
package Aircrew is
  -- Abstract State machine
  procedure Select_Crew_Member (Crew_Member : in
   Aircrew_Characteristics.Crew_Members);
  procedure Swap_Crew_Member (Old_Crew_Member : in
   Aircrew_Characteristics.Crew_Members;
   New_Crew_Member : in
   Aircrew_Characteristics.Crew_Members);
  procedure Select_Crew_For (Aircraft : in
   Aircrew_Characteristics.Aircraft_Class);
  procedure Build_Crew;
  function Numbers_Of_Crew_Members return Natural;
  Can_Not_Build_Crew : exception;
  Crew_Member_Does_Not_Exist : exception;
end Aircrew;

with Aircraft_Classes;
package Aircrew_Characteristics is
  type Crew_Member is limited private;
  -- private part ...
end Aircrew_Characteristics;
What is an ADT?

- An ADT package or generic package encapsulates:
- Multiple export Ada objects (data variables) of a single type. Note: It may also encapsulate hidden auxiliary Ada objects and types.
- Exported operations (e.g. subprograms or task entry points) that operate on these objects. Note: It may also encapsulate hidden auxiliary operations.
- Exceptions (error conditions) that are raised (and must be properly handled) when the operations cannot produce a valid result.
- An ADT is represented by a plural noun (e.g. Windows)
Example of an Abstract Data Type

with Coordinates, Velocity;
package Targets_Id is
  type TARGETS is private;
  type TARGET_CLASSIFICATION is (FRIENDLY, ENEMY, UNKNOWN);
  type TARGET_FORM is (AIRCRAFT, MISSILE, UNKNOWN);

procedure Classify (Target : in out TARGETS);
procedure Create (Target : out TARGETS);
procedure Delete (Target : in TARGETS);
function Classification_Of (Target : in TARGETS)
  return TARGET_CLASSIFICATION;
function Coordinates_Of (Target : in TARGETS)
  return Coordinates.COORDINATES_TYPE;
function Exists (Target : in TARGETS) return BOOLEAN;
function Form_Of (Target : in TARGETS) return TARGET_FORM;
function Velocity_Of (Target : in TARGETS) return Velocity.VELOCITY_TYPE;
CAN_NOT_CREATE : exception; -- Create_Next
DOES_NOT_EXIST : exception; -- Classify, Delete, Form_Of
private -- Hidden definition of target identification
  -- private part. We don't care!
end Targets_Id;
Kinds of Objects

- Active
  - Acts Autonomously
  - Acts upon other objects
  - May be acted upon
- Passive
  - Never acts autonomously
  - May act upon other objects
  - Is acted upon
Classifications of Operations in OOD

- Constructors: Operations that change the state of an object (e.g. Push (Element_On,My_Stack);).

- Selectors: Operations that retrieve the state of an object (e.g. Is_Empty --Return true if stack is empty).

- Iterators: For an object that is a collection of other objects, operations that visit the collection's components. (e.g. Get_Next(The_Iterator : in out Iterator);
  - Advance the iterator to the next item in the queue).

- Desired characteristics of object operations

  - Sufficient to permit all common uses of the object
  - Complete to permit all common uses of the object
  - Primitive whose efficient implementation can only be achieved with access to the underlying data representation.
Classes

- A class has the following characteristics:
  - represents real world object
  - is a template to create objects (variables)
  - encapsulates protected data with the operation on the data
  - access to protected data is through calls (messages) that invoke operations
  - In Ada a class may be a package that encapsulates a user defined data type and operations. There are many ways to create classes in Ada.
What a language needs for OOP

<table>
<thead>
<tr>
<th>Essentials</th>
<th>Nice to have</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>Typing</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Concurrency</td>
</tr>
<tr>
<td>Modularity</td>
<td>Persistence</td>
</tr>
<tr>
<td>Hierarchy</td>
<td></td>
</tr>
</tbody>
</table>
Ada 95 and OOSE

Ada 95

Is a superior language for supporting software engineering principles.

Is a language which provides support for OOP for those who wish to use it.

Provides a somewhat different model for OOP than other languages.
Object-Oriented Methodologies

- **OO methodologies** attempt to build complex systems using “classes” and “objects” as the building blocks
  - Attempts to model the real world via abstraction
  - Evolutionary, not revolutionary

- Experience has shown that this is the superior way to manage and decompose an inherently complex system. **Functional methodologies** require you to understand the entire system before you can modularize it!
How to implement the OO Paradigm

Macro View

- Identify the core requirements for the software (conceptualization)
- Develop a model of the system’s desired behavior (analysis)
- Create an architecture for the implementation (design)
- Evolve the implementation through successive refinements (evolution)
- Manage post delivery evolution (maintenance)
How to implement the OO Paradigm

Micro View

- Identify the classes at a given level of abstraction (OOD issue)
- Identify the objects at a given level of abstraction (OOD issue)
- Identify the relationships among these classes and objects (OOD issue)
- Specify the interface and then code the implementation of these classes and objects (OOP issue).
Benefits of the OO Paradigm

Reuse
Well-designed classes can be extended via inheritance. Previous code can be reused without modification to users of the original code.
   Allowing lots of subclasses allows special cases to be truly special without modifications to normal classes.

Integration
OO supports incremental development.
   Additions to the system are easier.
   It is easy to develop the parent class first, and then concentrate on subordinate classes.
Benefits of the OO Paradigm (cont.)

Testing
Easier to test, as specific behaviors and classes can be tested individually.
Specific test cases can be designed to test a class, and then this behavior does not need to be retested for subordinate classes.

Maintenance
Additions to the system can be accomplished without modifying existing code.
Instead of modifications to existing classes, new classes can be constructed.
Ada 95

✔ Ada 95 core language =

Ada 83

+ OOP (inheritance, Polymorphism, Dynamic Binding)
+ Hierarchical library units
+ data-oriented synchronization
+ flexible scheduling
+ … Specialize needs annexes
Ada 95 and OOSE

✔ Ada 95 has attempted to build on Ada 83 rather than changing the syntax drastically

✔ Mechanism for inheritance has been added
  – Permits type extension
  – Uses tagged types

● Child library units provide different views to different clients

● Ada 95 provides full support for OOP (Object-Oriented Programming)
Extensible vs. non-extensible types

```
type Sensor is record -- Not Extensible
  Current_State : State := On;
end Record;

procedure XXX ( Some_Parameter : Sensor);
function YYY return Sensor;
function ZZZ (Some_Parameter : in Sensor) return WWW;
```

This is an Abstract Data Type (ADT).

You can extend the functionality by adding operations. The data structure is static.
Extensible vs. non-extensible types

```ada
type Sensor is tagged record -- Extensible
  Current_State : State := On;
end Record;

procedure XXX (Some_Parameter : Sensor);
function YYY return Sensor;
function ZZZ (Some_Parameter : in Sensor) return WWW;
```

This is a tagged type that can be used as the parent of an inheritance chain.

The data structure may be extended, and operations added.
Encapsulation

- Declare a class
- Attributes
- Methods (Operations)
- An object’s state based on attributes
- An object’s behavior based on methods
- Packaged public & private components

✓ Encapsulation
✓ Inheritance
✓ Polymorphism
Encapsulation...

- Is sometimes used as a synonym for “information hiding” [Meyer97]
- Encompasses a classes attributes and operations [Pressman01]
- Is the combination of hiding and message interface abstraction; of hiding and packaging [Eliens00]
- Extends the concept of abstraction from data to both data and functions [Daconta99]
- The basic idea behind a package or a module [Ledgard96]
Encapsulation

- Declare a class
  - Attributes
  - Methods (Operations)
  - An object’s state based on attributes
  - An object’s behavior based on methods
  - Packaged public & private components

- Encapsulation
- Inheritance
- Polymorphism
Declaring a Class: Ada 95

package Rx is
  type Rx_type is private;
  procedure del_Rx (Rx_num);
  procedure refill_Rx (Rx_num);
  procedure wr_Rx (Rx_num, patient, physician, ...
   ...
  private:
    type Rx_type is record
      Rx_num: int;
      patient: string;
      physician: string;
      med_name: string;
      num_refills: integer;
    end record;
  end Rx;
Private Parts: Pkg Body

package body Rx is

procedure refill_Rx (the_num : in int) is
  Rx : Rx_type;
  begin
    --traverse Rx store, find Rx record
    if Rx.Rx_num = the_num then
      if Rx.num_refills > 0 then
        Rx.num_refills := Rx.num_refills - 1;
      else
        raise no_refills_left;
      endif;
    endif;
  end refill_Rx;
Inheritance

- Declare a parent class
- May be abstract
- Can have children
- Children inherit ALL attributes & ops
- Children may also have children

- Encapsulation
- Inheritance
- Polymorphism
Inheritance Example

Sortie
- unit
- sortie_num
- alert?
- home_base
- vehicle
- set_alert_status

TankerSortie
- fuel_amount
- refuel_ac

AttackSortie
- num_warheads
- target_list
- launch_weapon

BomberSortie
- landing_base
- scramble
- take_fuel
- return_to_base

refuels
Inheritance: Ada 95

type attack_sortie is new sortie with record
   num_warheads: integer;
   target_list: target_list_type;
end record;

type bomber_sortie is new attack_sortie with record
   landing_base: string;
end record;
type attack_sortie is new sortie with
record
num_warheads: integer;
target_list: target_list_type;
end record;

type bomber_sortie is new attack_sortie with
null record;
Tagged Types

type Rectangle is tagged
record
  Length : Float := 0.0;
  Width : Float := 0.0;
end record;
-- Operations for inheritance now defined
-- Example: Rectangles have a defined perimeter, and
-- children derived from Rectangle will have Perimeter

function Perimeter (R : in Rectangle ) return Float is
begin
  return 2.0 * (R.Length +R.Width);
end Perimeter;
Tagged Types - Inheritance

Any operation declared in the scope of a tagged type (or in the scope of a type derived from a tagged type) is a “primitive” operation. It is automatically inherited.

This is an example of “extending” the type without adding any new components. This is similar to the way Ada 83 used derived types.

type Square is new Rectangle with
  null record; -- inherit from parent, add no new fields

-- Square will inherit the primitive operation Perimeter

function Area (S : in Square ) return Float is
  begin
    return (S.Length * S.Width);
  end Area;
Tagged Types - Inheritance

type Cuboid is new Rectangle with
record
    Height : Float := 0.0;
end record;

function Perimeter (C : in Cuboid ) return Float is
begin
    return Perimeter (Rectangle(C)) * 2 + ( 4 * C.Height);
end Perimeter;

Cuboid inherits Perimeter from Rectangle. The function will have to be
updated for the new type (Perimeter is defined differently for cubes!).

To do this, you need to *override* the operation. One way to do this is to
write a new Perimeter. A better way it to base the new Perimeter on the
parent class operation. Note the *view conversion* in Perimeter - preventing
recursion, and basing the perimeter of a cube on the perimeter of the base
class.
with Calendar;

package New_Alert_System is

  type Device is (Teletype, Console, Big_Screen);

  --define the “base class”

  type Alert is tagged record
    Time_of_Arrival : Calendar.Time;
    Message : Text;
  end record;

  procedure Display ( ... );
  procedure Handle (A : in out Alert);
  procedure Set_Alarm ( ... );
Ada 95 Inheritance Programming

--now, use the base class and extend it via inheritance

type Low_Alert is new Alert;

type Medium_Alert is new Alert with record
   Action_Officer : Person;
   end record;

--and override one of the inherited operations

procedure Handle (MA : in out Medium_Alert);

type High_Alert is new Medium_Alert with record
   Ring_Alarm_At : Calendar.Time;
   end record;

procedure Handle (HA: in out High_Alert);

end New_Alert_System;
Completing the subprograms...

package body New_Alert_System is

procedure Handle(A: in out Alert) is begin
    A.Time_of_Arrival:= Calendar.Clock;
    Log(A);
    Display(A, Teletype);
end Handle;

procedure Handle(MA: in out Medium_Alert) is begin
    Handle(Alert(MA) );  -- conversion(no dispatch)
    MA.Action_Officer:= Assign_Volunteer;
    Display(MA, Console);
end Handle;

procedure Handle(HA: in out High_Alert) is begin
    Handle(Medium_Alert(HA) );  -- conversion(no dispatch)
    Display(HA, Big_Screen);
    Set_Alarm(HA);
end Handle;
...
end New_Alert_System;
Another user needs to extend again ...

with New_Alert_System;
package Emergency_Alert_system is
  type Emergency_Alert is
      new New_Alert_System.Alert with private;
  procedure Handle (EA : in out Emergency_Alert);
  procedure Display (EA : in Emergency_Alert;
                    On : in New_Alert_System.Devices);

  procedure Log (EA : in Emergency_Alert);

private
  ....
end Emergency_Alert_System;
Abstract Types & Subprograms

-- Baseline package used to serve as root of inheritance tree
package Vehicle_Package is
    type Vehicle is abstract tagged null record;
    procedure Start (Item : in out Vehicle) is abstract;
end Vehicle_Package;

• Purpose of an abstract type is to provide a common foundation upon which useful types can be built by derivation.

• An abstract subprogram is a place holder for an operation to be provided (it may not have a body).

• An abstract subprogram MUST be overridden for EACH subclass before any object of the subclass can be declared.
Abstract Types and Subprograms

```
type Train is new Vehicle with
   record
      passengers : Integer;
   end Train;

My_Train : Train; -- ILLEGAL
```

We can’t yet declare an object of *Train*. Why? Because we haven't filled in the *abstract parts* declared in its parent. We have completed the *abstract record*, but still need to define procedure *Start* for the *Train*.

```
type Train is new Vehicle with
   record
      passengers : Integer;
   end Train;

procedure Start (Item : in out Train) is ....
My_Train : Train;
```
Abstract types

type Planes is abstract new Vehicle with
record
   Wingspan : Some_Type;
end Planes;

function Runway_Needed_To_Land
   (Item : Planes) return Feet is abstract;

Type Planes is an abstract type based on Vehicle, which is also an abstract type. Therefore, the procedure Start must be overridden (to satisfy the requirements for Vehicle) and the function Runway_Needed_To_Land must be overridden (to satisfy the requirements for Planes) for any type derived from Planes.
With each tagged type T there is an associated type T’Class. This type comprises the union of all the types in the tree of derived types rooted at T. Legal values of T'Class are the values of T and all its derived types. A value of any type derived from T can be implicitly converted to the type T'Class.
Polymorphic Data

- Class-wide variables are said to be polymorphic because their values are of different “shapes” (from the Greek poly, many, and morphe, form).

- When calling a subroutine with a polymorphic parameter, the compiler is unable to determine which subroutine to call, since the actual type of a polymorphic variable will not be known until run time!

- Thus, the binding to some subroutines must be delayed until execution - hence the term “dynamic binding”.

- A call to a routine that must be dynamically bound is called a “dispatching” call
Class Wide Programming (Dispatching)

```ada
-- class-wide value as parameter
Procedure Move_All ( Item : in out Vehicle’Class) is

begin

  ... Start (Item); -- dispatch according to tag

  ...

end Move_All;
```

The procedure Move_All is a **class-wide** operation, since any variable in the Vehicle hierarchy can be passed to it.

Start, however, is defined for each type within the Vehicle hierarchy. Depending on the type of Item, a different Start will be called. During runtime, the specific type of Item is known, but it is not known at compile time. The **runtime system** must **dispatch** to the correct procedure call.
Static Binding

-- class-wide value as parameter

Procedure Move_All ( Item : in out Vehicle'Class) is
...
begin
...
  Start (Item); -- dispatch according to tag (Dynamic Dispatching)
    --vs.
  if Item in Jet then
    Start (Jet(Item)); -- static call to the Start for Jet.
      -- This is a membership test and View Conversion.
  ...
end Move_All;