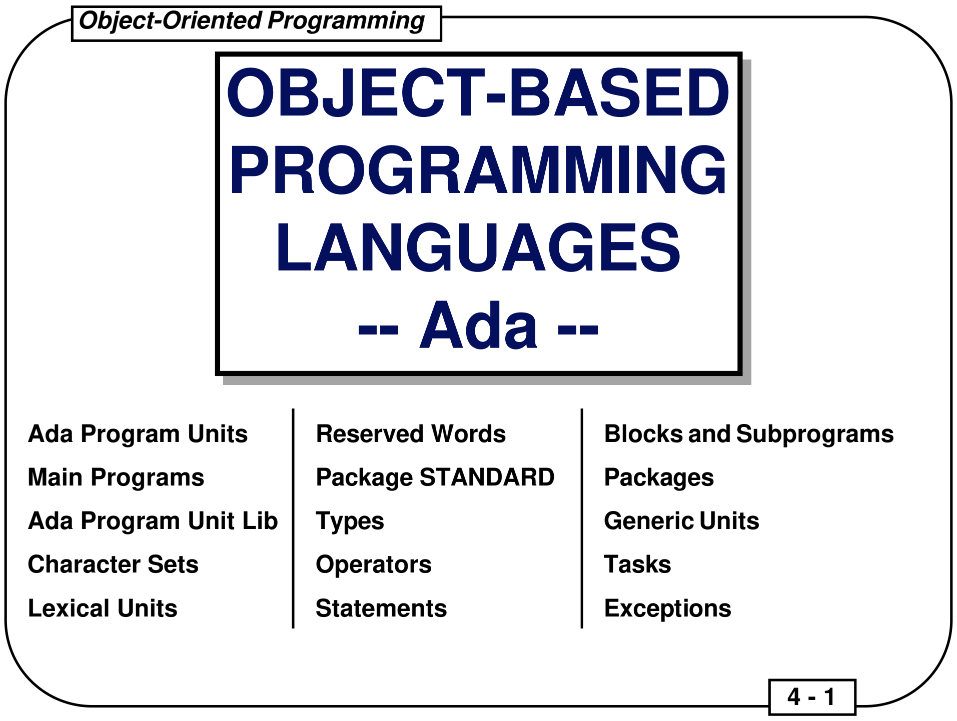


Object-Oriented Programming

OBJECT-BASED PROGRAMMING LANGUAGES -- Ada --

Ada Program Units
Main Programs
Ada Program Unit Lib
Character Sets
Lexical Units

Reserved Words
Package STANDARD
Types
Operators
Statements

Blocks and Subprograms
Packages
Generic Units
Tasks
Exceptions

Objectives of Module 4

- Present and discuss the syntax of the Ada Programming Language
- Present and discuss the features of Ada which support object-oriented programming

Suggested Reading

David J. Naiditch, **Rendezvous with Ada: A Programmer's Introduction**, John Wiley & Sons, 1989, ISBN 0-471-61654-0

Grady Booch, **Software Engineering with Ada, 2nd Edition**, Benjamin/Cummings Publishing Company, 1987, ISBN 0-8053-0604-8

Grady Booch, **Software Components with Ada**, Benjamin/Cummings Publishing Company, 1987, ISBN 0-8053-0610-2

Michael B. Feldman and Elliot B. Koffman, **Ada Problem Solving and Program Design**, Addison-Wesley Publishing Company, 1992, ISBN 0-201-50006-X

Karl A. Nyberg (editor), **The Annotated Ada Reference Manual, 2nd Edition**, Grebyn Corporation, 1991, ISBN unknown

Ada Program Units

Ada source file -- contains one or more Ada program units

An Ada Program or System is composed of one or more program units, where a program unit is:

- **a subprogram**
- **a package**
- **a task**
- **a generic unit**

Each program unit is divided into two parts:

- **a specification, which defines its interface to the outside world**
- **a body, which contains the code of the program unit**

When an Ada compiler runs, it compiles an Ada source file. This Ada source file contains one or more Ada program units. The compiler places these units into an Ada program unit library if they compile successfully.

Later, when an Ada linker (also commonly called a binder) is run, an executable is produced from program units within an Ada program unit library. The mainline is specified to the linker, and the linker sets this unit up to begin execution. An Ada mainline is always an Ada procedure (a subprogram unit).

Program units can be nested -- they can contain zero or more other program units.

Program Units: SUBPROGRAM

A *subprogram* is an expression of sequential action.

Two kinds of subprograms exist:

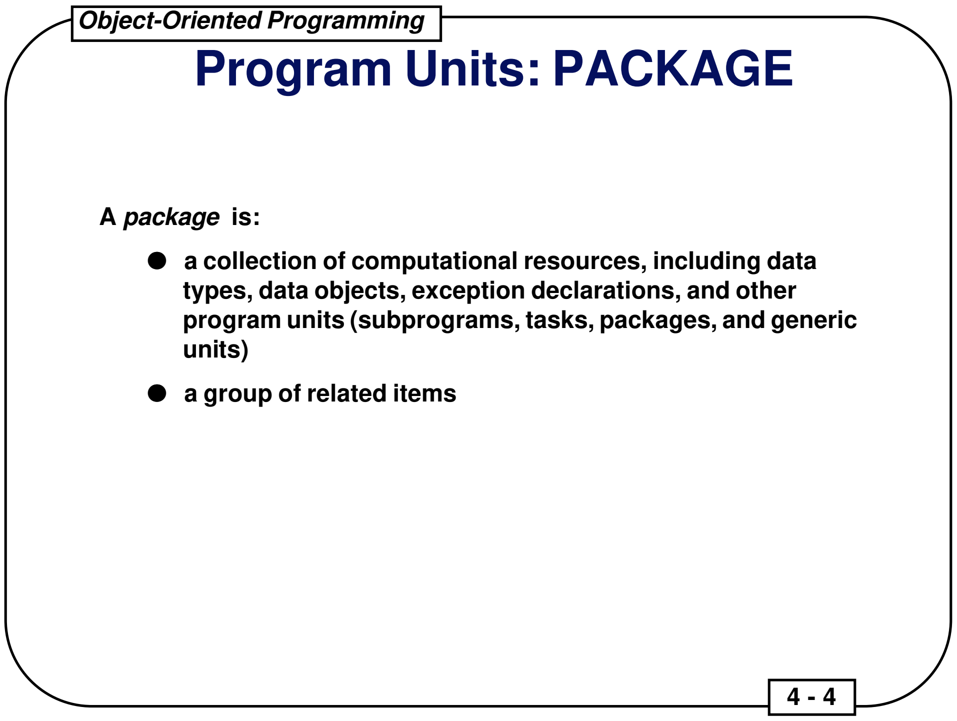
- **procedure**
- **function**

An example of a procedure subprogram:

```
type REAL_ARRAY is array (1..20) of FLOAT;
procedure SORT (ITEMS : in out REAL_ARRAY);  -- spec
procedure SORT (ITEMS : in out REAL_ARRAY) is -- body
     -- definitions of types, objects, exceptions, and
     -- other program units (subprograms, packages, tasks,
     -- and generic units) local to SORT
begin
     -- body of code which implements SORT
     null;  -- no code for now
end SORT;
```

An example of a function subprogram:

```
function IS_ZERO (ITEM : in FLOAT) return BOOLEAN: -- spec
function IS_ZERO (ITEM : in FLOAT) return BOOLEAN is -- body
     -- definitions of types, objects, exceptions, and
     -- other program units (subprograms, packages, tasks,
     -- and generic units) local to IS_ZERO
begin
     null;   -- no code for now
end IS_ZERO;
```



Program Units: PACKAGE

A *package* is:

- a collection of computational resources, including data types, data objects, exception declarations, and other program units (subprograms, tasks, packages, and generic units)
- a group of related items

In object-oriented programming, a package contains the definition of a particular object or a class of objects. This includes the member data and all methods associated with the object or class.

Packages are fundamental to Ada. For instance, Ada by itself has no Input/Output capabilities, so the package TEXT_IO is provided with all Ada compilers to provide input/output capabilities for characters, strings, floating point numbers, fixed point numbers, integers, and user-defined types.

An example of an Ada package specification --

```
package Console_Terminal_Screen is

     -- definitions of types, objects, exceptions, and other program

     -- units provided to external program units by this package

     subtype ROW is INTEGER range 1..24;

     subtype COLUMN is INTEGER range 1..80;

     procedure Clear_Screen;

     procedure Position_Cursor (At_Row : in ROW;

       At_Column : in COLUMN);

     procedure Write (Value : in STRING);

end Console_Terminal_Screen;
```

Program Units: TASK

A *task* is:

- **an action implemented in parallel with other tasks**
- **a code item which may be implemented on one processor, a multiprocessor (more than one CPU), or a network of processors**
- **composed of a specification and a body**

An example of a task specification --

```
task Terminal_Driver is
     -- entry points to the task specify how other
     -- tasks communicate with this task
     entry Get (Char : out CHARACTER);
     entry Put (Char : in  CHARACTER);
end Terminal_Driver;
```

Program Unit: GENERIC UNIT

A *generic unit* is:

- **a reusable software component**
- **a special implementation of a subprogram or package which defines a commonly-used algorithm in data-independent terms**

An example of a generic subprogram is --

```
generic -- specification
     type ELEMENT is private;  -- thing manipulated
procedure Exchange (Item1 : in out ELEMENT;
  Item2 : in out ELEMENT);

procedure Exchange (Item1 : in out ELEMENT;
  Item2 : in out ELEMENT) is -- body
     Temp : ELEMENT;
begin
     Temp := Item1;
     Item1 := Item2;
     Item2 := Temp;
end Exchange;
```

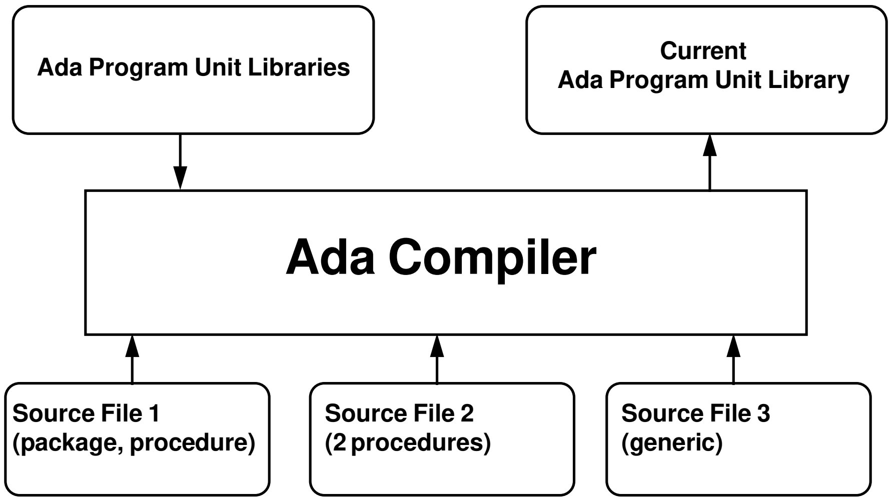
Procedures as Main Programs

Ada does not have a separate construct for a main program.

Instead, Ada program units (subprograms, packages, tasks, and generic units) are compiled into an Ada library and then, at some later time, one of the procedures is selected to be the mainline procedure at which execution of the program is to start.

A main procedure has no parameters.

Ada and the Ada Program Unit Library


Ada Program Unit Libraries
Current Ada Program Unit Library
Ada Compiler
Source File 1 (package, procedure)
Source File 2 (2 procedures)
Source File 3 (generic)

Note that the Ada compiler outputs into the current Ada program unit library as its only target. The compiler does not create .o files necessarily.

Reuse is accomplished by accessing existing Ada program unit libraries, thereby gaining use of the program units contained in them.

All Ada compilers provide a common program unit library that contains the following packages:

package STANDARD; -- contains integers, floats, and operations

package TEXT_IO; -- support for I/O

package SYSTEM; -- ability to address memory

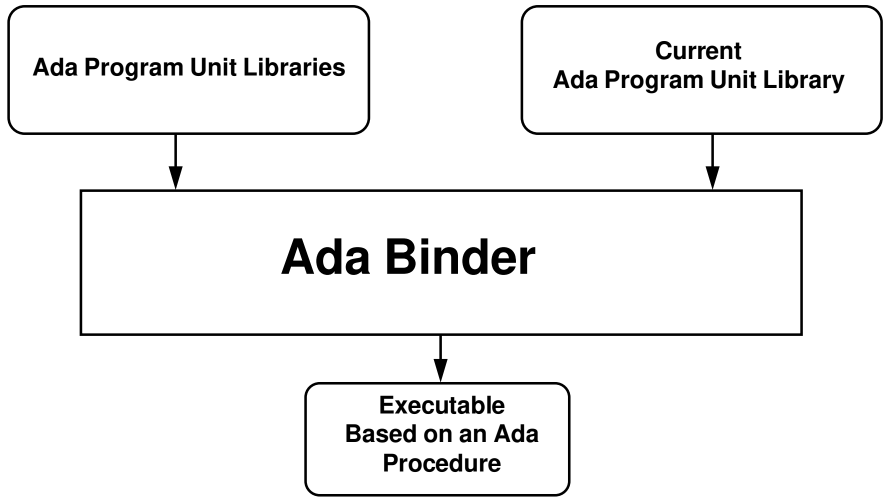
package SEQUENTIAL_IO; -- support for sequential I/O only

package DIRECT_IO; -- support for random I/O only

package IO_EXCEPTIONS; -- errors which may come up in I/O

package LOW_LEVEL_IO; -- special platform-specific I/O

Creating an Executable


Ada Program Unit Libraries
Current Ada Program Unit Library
Ada Binder
Executable Based on an Ada Procedure

The Ada Binder builds an executable from a procedure that is located anywhere in the Ada libraries. A chain of program units is assembled to create this executable:

- the mainline procedure comes first, and this procedure requires certain program units for support (it *depends* upon these program units and they are named in its **with** statements)
- the program units **withed** by the mainline procedure are incorporated into the executable, and these program units may **with** other program units
- the next layer of program units is included, and so on as they **with** yet other program units
- the Ada runtime system, which supports initialization, exception handling, tasking, and other features of the language may either be included in the executable or tied into by the executable

The **with** statement in Ada causes one program unit to gain access to another. An example:

```
with Text_IO;

procedure IO_Demo is  -- call Put_Line function in Text_IO

begin

  Text_IO.Put_Line("This line is written to the console");

end IO_Demo;
```

Basic and Extended Character Sets

The Basic Character Set (BCS) is one of two character sets used by Ada programs:

- The BCS was designed to facilitate transportability between computer systems.
- The BCS consists of:
  - uppercase letters only: `A-Z`
  - digits: `0-9`
  - special characters: `" # ' ( ) * + - / , . : ; < = > _ | &`
  - the space character

The Extended Character Set (ECS) maps to the 95-character ASCII (American Standard Code for Information Interchange) set:

- The ECS consists of:
  - all characters in the BCS
  - more special characters: `! ~ $ ? @ [ ] \ ` { } ^ %`
  - lower-case letters: `a-z`

Package ASCII

Package ASCII **within the supplied package STANDARD provides:**

- **names for the non-printing ASCII characters**
- **names for the characters in the ECS which are not a part of the BCS**

Examples:

```
c1  : character := ASCII.NUL;
c2a : character := '#';
c2b : character := ASCII.SHARP; -- same as c2a
c3a : character := 'a';
c3b : character := ASCII.LC_A;  -- same as c3a
```

The predefined package STANDARD is the only Ada package which is automatically withed by every Ada program unit without the programmer having to explicitly do so. Consequently, package ASCII is always available.

A sample program:

```
with Text_IO;  -- for output
procedure ASCII_Demo is
begin
  Text_IO.Put("Ring the Bell: ");  -- Put for a string
  for I in 1 .. 20 loop
    Text_IO.Put(ASCII.BEL);  -- Put for a character
  end loop;
  Text_IO.New_Line;
end ASCII_Demo;
```

Lexical Unit

A Lexical Unit is a basic token of the Ada language which is built from the character sets:

- comments

```
-- this is a comment, starting at the -- and

-- going to the end of the line
```

- identifiers (a letter followed by zero or more letters, digits, and underscores, and case is not significant)

```
A   THIS_IS_A_TEST   FACTOR_44

hello_world  usart_status_flag

package -- this is a reserved word
```

- numeric literals (real/floats and integers in bases 2 to 16)

```
45   2.7   9.9e-56  1_000_000   16#F.2C#

2#0010#   7#16#  8#1_377#  16#0c2b#  16#CC_48#

3.14159_26535_89793_23846_26434
```

Lexical Unit, Continued

- character literals

  `'A'    '*'`

  `'''` -- the character '

- strings

  `"hello, world"`

  `""` -- the empty string

  `""""` -- a string whose content is "

- delimiters (single and compound)

  ```
  '  (  )  *  +  ,  -  .  /  :  ;  <  =  >  |  &
  =>  ..  **  :=  /=  >=  <=  <<  >>  <  >
  ```

Notes:

- Any number of spaces (and lines) may separate lexical units
- A lexical unit must fit on one line

Reserved Words

Reserved words are identifiers which may be used in only certain contexts:

- They may *NOT* be used as variables, enumeration literals, procedure names, etc.
- They may be a part of strings ("my package is in").
- They may be a part of other lexical units (e.g., PACKAGE_52 is O.K.).

Complete List of Ada Reserved Words

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array	else		pragma	then
at	elsif	limited	private	type
	end	loop	procedure	
begin	entry			use
body	exception	mod	raise	
	exit		range	when
case		new	record	while
constant	for	not	rem	with
	function	null	renames	
			return	xor
			reverse	

Package STANDARD

Package STANDARD **is automatically** ***withed*** **and** ***used*** **by all Ada program units.**

Package STANDARD **contains:**

- **type BOOLEAN and the associated operations**
- **type INTEGER and the associated operations**
- **type FLOAT and the associated operations**
- **the types universal real, universal integer, and universal fixed along with their associated operations**
- **type CHARACTER and the associated operations**
- **package ASCII (provides alternate character representations)**
- **subtype NATURAL and subtype POSITIVE**
- **type STRING and the associated operations**
- **type DURATION (a fixed point type used to represent time)**
- **several predefined exceptions**

- type BOOLEAN and associated operations:

  `=    /=    <    <=    >    >=`

  `and    or    xor    not`

- type INTEGER and associated operations:

  `=    /=    <    <=    >    >=`

  `+    -    abs` (unary operations)

  `+    -    *    /    rem    mod` (binary operations)

  `**`

- type FLOAT and associated operations:

  `=    /=    <    <=    >    >=`

  `+    -    abs` (unary operations)

  `+    -    *    /` (binary operations)

  `**` (INTEGER exponent)

- types CHARACTER and STRING:

  `=    /=    <    <=    >    >=`

  `&` (for both char and string)

- subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;

- subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

- predefined exceptions:

  ```
  CONSTRAINT_ERROR
  PROGRAM_ERROR
  TASKING_ERROR
  NUMERIC_ERROR
  STORAGE_ERROR
  ```

- the types universal real, universal integer, and universal fixed and operations:

```
universal real := universal integer * universal real
universal real := universal real * universal integer
universal real := universal real / universal integer
universal fixed := user-defined fixed * user-defined fixed
universal fixed := user-defined fixed / user-defined fixed
```

Type Definitions and Object Declarations

A *type* is a class of objects which characterizes:

- a set of values which objects of that type may take on
- a set of attributes (e.g., INTEGER'LAST is the last integer)
- a set of operations which may be performed on objects of that type

Several classes of types are available in Ada:

- scalar data types
  - integer
  - real (floating point and fixed point)
  - enumeration
- composite data types
  - array
  - record
- access data types
- private data types
- subtypes
- derived types

Scalar Data Types

- integer:

```
INTEGER   -- a predefined type
NATURAL  -- a predefined type, >= 0
POSITIVE  -- a predefined type, >= 1
type INDEX is range 1..50;  -- user-defined
```

- real (floating point and fixed point):

```
FLOAT  -- a predefined type
type MASS is digits 10;  -- 10 sig digit user-defined float
type VOLTAGE is delta 0.01  -- a user-defined fixed point
  range 0.0 .. 50.0;
```

- enumeration:

  BOOLEAN -- a predefined type (FALSE, TRUE)

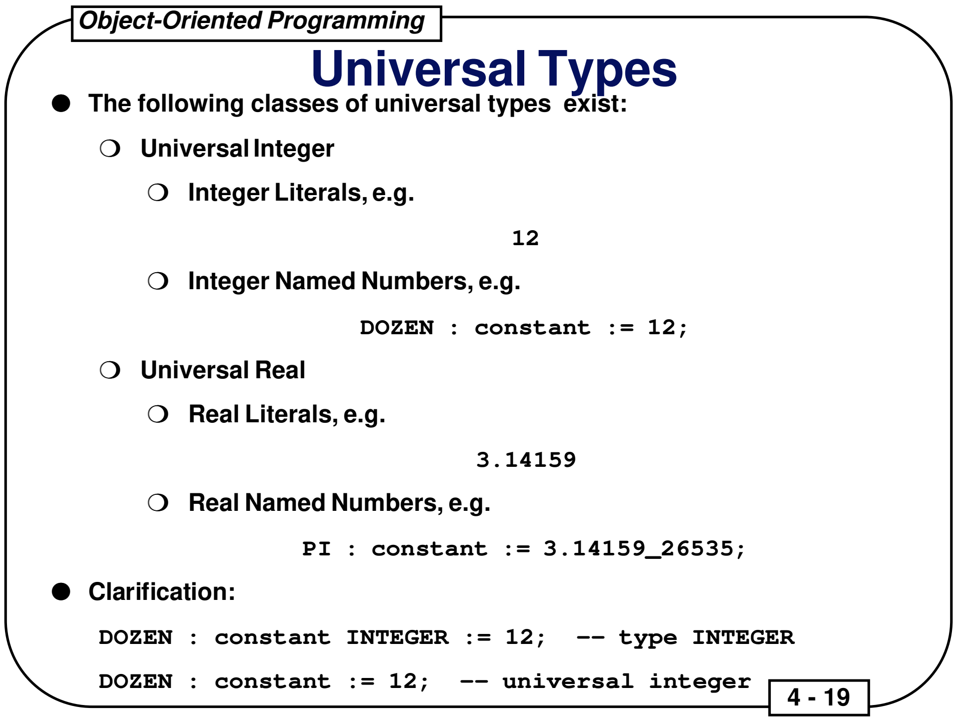
  CHARACTER -- a predefined type

  type COLOR is (RED, GREEN, BLUE); -- user-defined

Numeric and Discrete Types

	Integer	Real	Enumeration
Numeric	x	x	
Discrete	x		x

It is important to be able to distinguish between numeric and discrete types since only discrete types may be used for loop variables.



Universal Types

- The following classes of universal types exist:
  - Universal Integer
    - Integer Literals, e.g.

```
12
```

    - Integer Named Numbers, e.g.

```
DOZEN : constant := 12;
```

  - Universal Real
    - Real Literals, e.g.

```
3.14159
```

    - Real Named Numbers, e.g.

```
PI : constant := 3.14159_26535;
```

- Clarification:

```
DOZEN : constant INTEGER := 12;  -- type INTEGER
DOZEN : constant := 12;  -- universal integer
```

Advantages of Universal Types:

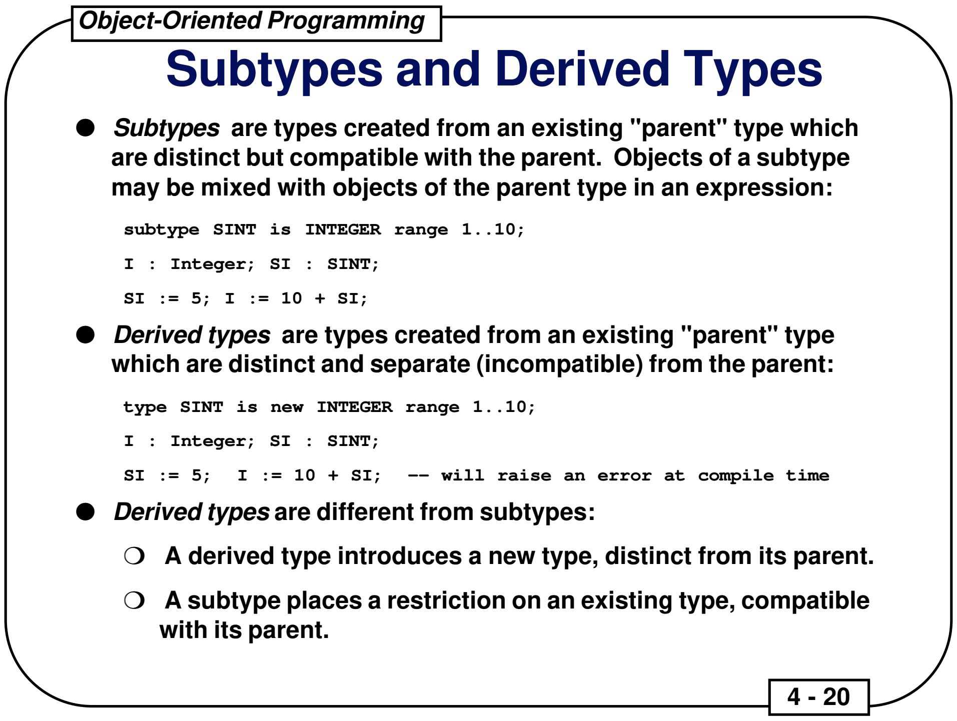
- Universal Types do not have any practical size constraints

```
SPEED_OF_LIGHT : constant := 186_282;
  -- valid on even 16-bit machines (where INTEGER'LAST = 32_767)
```

- Code may execute faster: when named numbers are combined with other named numbers or numeric literals, the resulting expression may be evaluated at compilation time rather than run time

Expressions consisting of only named numbers or numeric literals are called literal expressions. Named numbers may be initialized to literal expressions but not to non-literal expressions:

```
DOZEN : constant := 12;
BAKERS_DOZEN : constant := DOZEN + 1;
  -- OK because "DOZEN + 1" is a literal expression
DOZEN : constant INTEGER := 12;
BAKERS_DOZEN : constant := DOZEN + 1;
  -- not OK because "DOZEN + 1" is an expression based on an INTEGER variable
```



Subtypes and Derived Types

- ***Subtypes*** **are types created from an existing "parent" type which are distinct but compatible with the parent. Objects of a subtype may be mixed with objects of the parent type in an expression:**

```
subtype SINT is INTEGER range 1..10;
I : Integer; SI : SINT;
SI := 5; I := 10 + SI;
```

- ***Derived types*** **are types created from an existing "parent" type which are distinct and separate (incompatible) from the parent:**

```
type SINT is new INTEGER range 1..10;
I : Integer; SI : SINT;
SI := 5;  I := 10 + SI;  -- will raise an error at compile time
```

- ***Derived types*** **are different from subtypes:**
  - ❍ **A derived type introduces a new type, distinct from its parent.**
  - ❍ **A subtype places a restriction on an existing type, compatible with its parent.**

Derived types make a lot of sense, providing a check when mapping to the real world. For instance, in the real world, you would never try to add something of type SPEED (say, in miles per hour) to something of type TIME (say, in hours). It simply does not make sense. Derived types in Ada prevent this kind of thing from happening:

```
type SPEED is new FLOAT range 0.0 .. 1_000_000.0;  -- MPH
type TIME is new FLOAT range 0.0 .. 24.0;  -- HOURS
S : SPEED := 25.0; T : TIME := 12.00;
S := S + T;  -- illegal, flagged at compile time
```

Arrays

An *array* is an object that consists of multiple homogenous components (i.e., each component is of the same type).

An entire array is referenced by a single identifier:

```
type FLOAT_ARRAY is array (1..10) of FLOAT;
  -- type declaration
My_Float_Array : FLOAT_ARRAY;
  -- array reference and definition
```

Each component of an array is referenced by the identifier which references the array being followed by an index in parentheses:

```
My_Float_Array(5) := 12.2;  -- assign one element
for i in My_Float_Array'First .. My_Float_Array'Last loop
  My_Float_Array(i) := 0.0;  -- initialize all elements
end loop;
```

Array Type Statement

The general syntax is:

type array_type_name is array (index_specification) of element_type;

- ***array_type_name*** is the name given to this type, not the name of a specific array; specific arrays are declared later as array objects
- ***index_specification*** is the type and value range limits, if any, of the index
- ***element_type*** is the type of the array elements

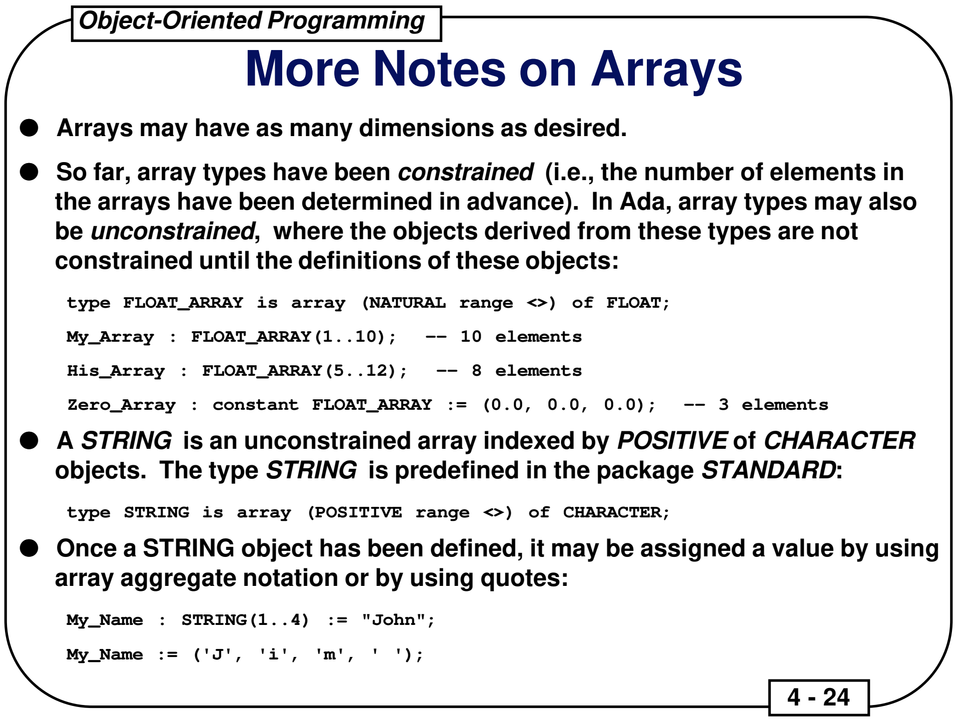
Examples of Arrays

```
type Color is (RED, GREEN, BLUE);  -- an enumeration type (used later)


type VALUES is array (1..8) of FLOAT;  -- a vector of 8 real numbers
My_Floats : VALUES;  -- object definition
His_Floats : VALUES := (1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8);
  -- object definition with initialization
Zero_Floats : VALUES := (others => 0.0);  -- initialized to 0.0


type CVAL is array (COLOR) of FLOAT;
  -- a vector of 3 real numbers indexed by RED, GREEN, and BLUE
My_Color_Values : CVAL;  -- object definition
Reference_Color_Values : constant CVAL := (1.1, 2.2, 3.3);
  -- constant object declaration


type SCREEN_DOTS is array (1..1024, 1..1024) of COLOR;
My_CRT_Screen : SCREEN_DOTS := (others=> (others=> RED));  -- all RED
```

Array Aggregates

An entire array may be initialized by assigning it to an array aggregate.

```
type MENU_SELECTION is (SPAM, MEAT_LOAF, HOT_DOG, BURGER);
type DAY is (MON, TUE, WED, THU, FRI);
type SPECIAL_LIST is array (DAY) of MENU_SELECTION;
Specials:SPECIAL_LIST;

Specials := SPECIAL_LIST'(SPAM, HOT_DOG, BURGER, MEAT_LOAF, SPAM);
Specials := (SPAM, HOT_DOG, BURGER, MEAT_LOAF, SPAM);
Specials := (MON => SPAM,
  TUE => HOT_DOG,
  WED => BURGER,
  THU => MEAT_LOAF,
  FRI => SPAM);
Specials := (MON | FRI => SPAM,
  TUE | WED | THU => BURGER);
Specials := (MON .. WED => BURGER, others => MEAT_LOAF);
```



More Notes on Arrays

- **Arrays may have as many dimensions as desired.**
- **So far, array types have been *constrained* (i.e., the number of elements in the arrays have been determined in advance). In Ada, array types may also be *unconstrained*, where the objects derived from these types are not constrained until the definitions of these objects:**

```
type FLOAT_ARRAY is array (NATURAL range <>) of FLOAT;
My_Array : FLOAT_ARRAY(1..10);   -- 10 elements
His_Array : FLOAT_ARRAY(5..12);   -- 8 elements
Zero_Array : constant FLOAT_ARRAY := (0.0, 0.0, 0.0);   -- 3 elements
```

- **A *STRING* is an unconstrained array indexed by *POSITIVE* of *CHARACTER* objects. The type *STRING* is predefined in the package *STANDARD*:**

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

- **Once a STRING object has been defined, it may be assigned a value by using array aggregate notation or by using quotes:**

```
My_Name : STRING(1..4) := "John";
My_Name := ('J', 'i', 'm', ' ');
```

More Notes on STRING Objects

- A double quote may be placed into a string by using two double quotes in a row:

```
Message : STRING (1..4) := """Hi""";
```

- Strings are fixed in their length. Ada does not support any predefined variable-length strings, but such strings can be implemented (see CS Parts):

```
Pet_Name : STRING(1..5);
Pet_Name := "Pete";  -- illegal because Pet_Name has 5 chars and "Pete" has 4
Pet_Name := "Repeat";  -- illegal for similar reasons
Pet_Name := "Pete ";  -- OK (note trailing space)
Pet_Name(1..4) := "Jake";  -- OK due to slice
```

- Slices can be used to assign parts of a string:

```
Message(1..2) := "Hi";
```

- Although the size of a STRING variable is set by its constraint, the size of a STRING constant may be inferred from the size of the aggregate assigned to it:

```
My_Pet : constant STRING := "Jake the Snake";
```

- Package TEXT_IO supports a STRING input procedure called GET_LINE which may be used to input values into STRINGs. GET_LINE requires the name of the STRING object and a variable of type NATURAL as parameters:

```
Input_Line : STRING(1..80);  Input_Last : NATURAL;
Text_IO.Get_Line(Input_Line, Input_Last);
  -- Assume that the user types "Hello<CR>", where <CR> is the RETURN key
  -- Input_Line(1..5) = "Hello" and Input_Last = 5 is the result
```

Boolean Vectors

A *boolean vector* is a user-defined type which is a vector of BOOLEANs:

```
type BOOLEAN_VECTOR is array (POSITIVE range <>) of BOOLEAN;
```

A Boolean vector is the only type of array that can be operated on by the logical operators *and, or*, *xor*, and *not*.

```
declare
  T : constant BOOLEAN := TRUE;  F : constant BOOLEAN := FALSE;
  A : BOOLEAN_VECTOR (1..4) := (T, F, T, F);
  B : BOOLEAN_VECTOR (1..4) := (T, F, F, T);
  C : BOOLEAN_VECTOR (1..4);
begin
  C := not A;     -- yields (F, T, F, T);
  C := A and B;   -- yields (T, F, F, F);
  C := A or B;    -- yields (T, F, T, T);
  C := A xor B;   -- yields (F, F, T, T);
end;
```

Array Attributes and Operations

Some interesting array attributes are:

FIRST -- first index value	LAST -- last index value
RANGE -- array'FIRST .. array'LAST	LENGTH -- number of elements

These attributes apply to array objects (which are, of course, constrained) and constrained array types. Operations on arrays are:

Operation	*Restrictions*
Attributes (FIRST, etc)	None
Logical (not, and, or, xor)	Must be BOOLEAN vectors of same length and type
Concatenation (&)	Must be vectors
Assignment (:=)	Must be of the same size and type
Type Conversions	Same size and component and index types
Relational (<,>,<=,>=)	Must be discrete vectors of same type
Equality (=, /=)	Must be of the same type

Record Types without Discriminants

The most basic kind of record is that declared without discriminants. The general syntax of a record type declaration is:

```
type record_type_name is record
  record_components;
end record;
```

Example:

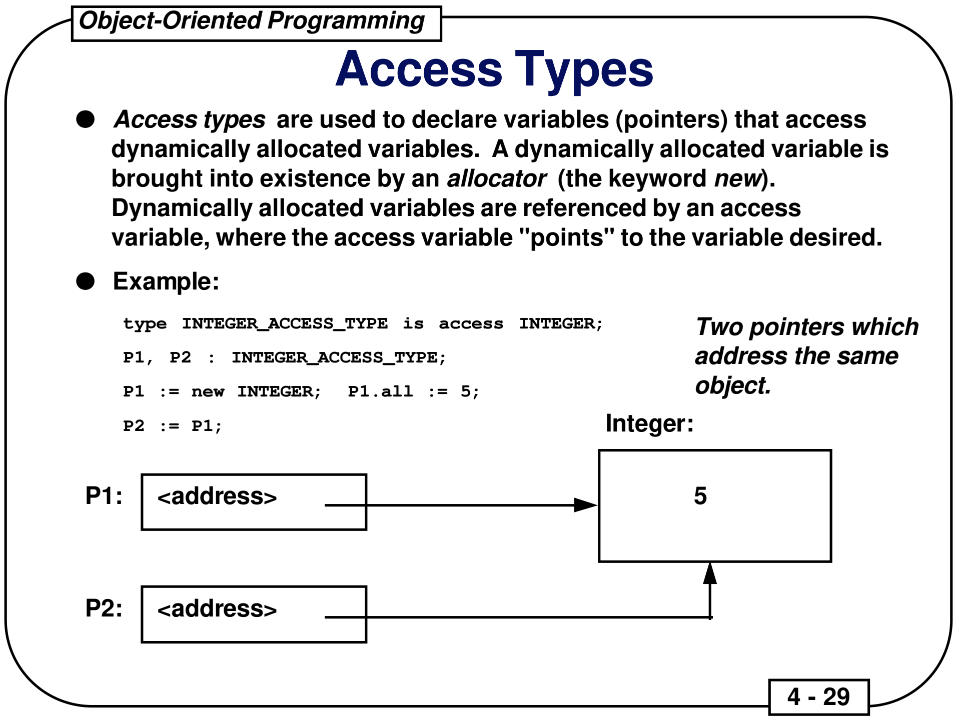
```
type MY_RECORD is record
  I : Integer;
  F : Float;
end record;
```

Record Types with Discriminants

Record types with discriminants may be used to define records to be of the same type even though the kind, number, and size of the components differ between individual records.

Variant records **are those that differ from one another in the kind and number of components. Example:**

```
type RECORDING_MEDIUM is (PHONOGRAPH, CASSETTE, CD);
type MUSIC_TYPE is (CLASSICAL, JAZZ, NEW_AGE, FOLK, POP);
type RECORDING (Device : RECORDING_MEDIUM := CD) is record
  Music : MUSIC_TYPE;
  case Device is
    when PHONOGRAPH =>
      Speed : RPM;
    when CASSETTE =>
      Length : NATURAL;
    when CD => null;
  end case;
end record;
```

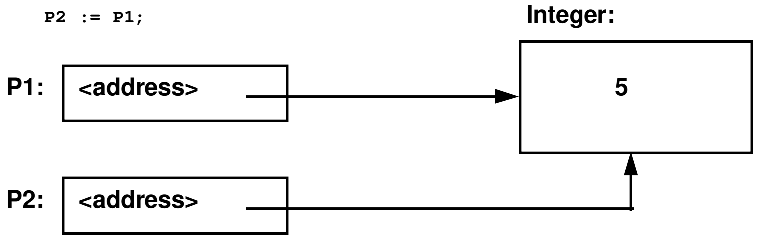


Access Types

- ***Access types*** **are used to declare variables (pointers) that access dynamically allocated variables. A dynamically allocated variable is brought into existence by an *allocator* (the keyword *new*). Dynamically allocated variables are referenced by an access variable, where the access variable "points" to the variable desired.**
- **Example:**

```
type INTEGER_ACCESS_TYPE is access INTEGER;
P1, P2 : INTEGER_ACCESS_TYPE;
P1 := new INTEGER;  P1.all := 5;
P2 := P1;
```

Two pointers which address the same object.



In Ada, every pointer (access variable) has an initial value of **null**. *This is the only kind of data object in Ada that is initialized implicitly when it is created without the user having to define the initialization.*

Example:

```
type CHARACTER_ACCESS_TYPE is access CHARACTER;

CH1_Ptr : CHARACTER_ACCESS_TYPE;  -- value is null by default

CH2_Ptr : CHARACTER_ACCESS_TYPE := null;  -- made null explicitly

CH3_Ptr : CHARACTER_ACCESS_TYPE := new CHARACTER;

  -- character is created along with the access variable

CH4_Ptr : CHARACTER_ACCESS_TYPE := new CHARACTER'('A');

  -- character is created and initialized

CH : CHARACTER;

CH := CH4_Ptr.all;  -- CH takes on the value 'A'
```

Representation Attributes

The following are attributes which may be applied to various entities in order to determine some of their specifics:

- ADDRESS -- reports the memory location of an object, program unit, label, or task entry point
- SIZE -- reports the size, in bits, of an object, type, or subtype
- STORAGE_SIZE -- reports the amount of available storage for access types and tasks; if P is an access type, P'STORAGE_SIZE gives the amount of space required for an object accessed by P; if P is a task, P'STORAGE_SIZE gives the number of storage units reserved for task activation
- POSITION (records only) -- reports the offset, in storage units, of a record component from the beginning of a record
- FIRST_BIT (records only) -- reports the number of bits that the first bit of a record component is offset from the beginning of the storage unit in which it is contained
- LAST_BIT (records only) -- reports the number of bits that the last bit of a record component is offset from the beginning of the storage unit that contains the first bit of the record component

The 4 Representation Clauses

- **Length clauses -- establish amount of storage space used for objects**

```
type DIRECTION is (UP, DOWN, RIGHT, LEFT);
for DIRECTION'SIZE use 2;   -- 2 bits
```

- **Enumeration clauses -- specify the internal representation of enumeration literals**

```
type BIT is (OFF, ON);
for BIT'SIZE use 1;
for BIT use (OFF => 0, ON => 1);
```

- **Record Representation clauses -- associate record components with specific locations in bit fields**
- **Address clauses -- specify the addresses of objects**

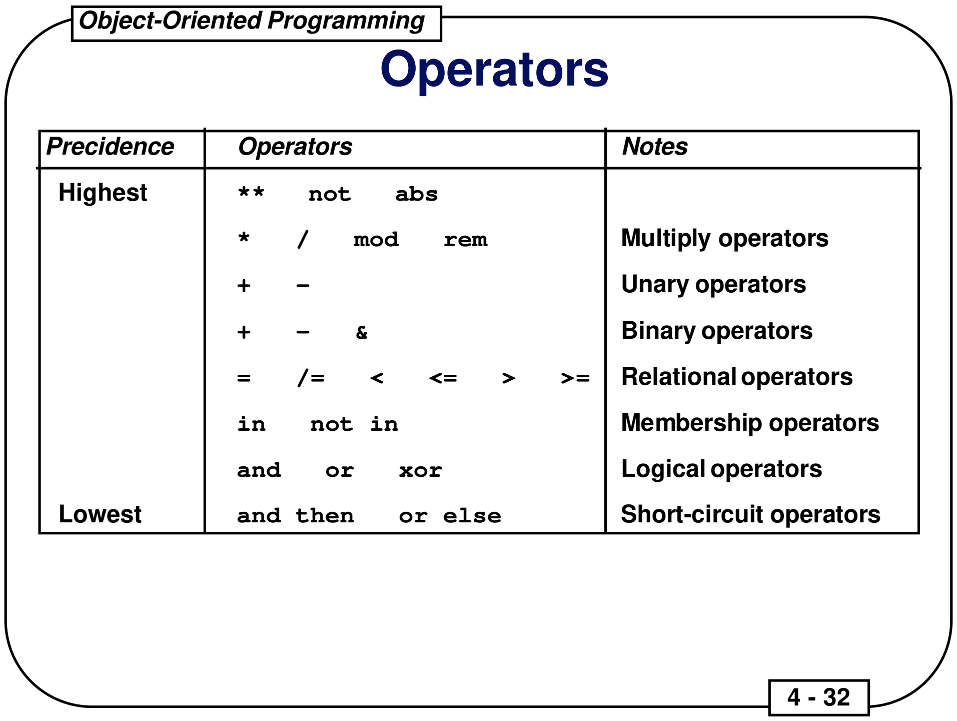
```
CPU_STATUS : Integer;   -- define object
for CPU_STATUS use at 16#080#;   -- define address
```

An example for records:

```
type BIT is (OFF, ON);
for BIT'SIZE use 1;  for BIT use (0, 1);
type STATUS_WORD is record
  Carry_Bit : BIT;
  Overflow_Bit : BIT;
  Zero_Bit : BIT;
end record;
for STATUS_WORD'SIZE use 8;
for STATUS_WORD use record
  Carry_Bit      at 0 range 0..0;  -- bit 0
  Overflow_Bit at 0 range 2..2;  -- bit 2
  Zero_Bit        at 0 range 7..7;  -- bit 7
end record;
CPU_STATUS : STATUS_WORD;  -- define object
for CPU_STATUS use at 16#100#;  -- define address of object
```

An example for tasks:

```
task RUNNING_SCORE is
  entry HIT;  for HIT use at 16#020#;
  entry MISS; for MISS use at 16#040#;
end RUNNING_SCORE;
```



Operators

Precidence	Operators	Notes
Highest	`**` `not` `abs`	
	`*` `/` `mod` `rem`	Multiply operators
	`+` `-`	Unary operators
	`+` `-` `&`	Binary operators
	`=` `/=` `<` `<=` `>` `>=`	Relational operators
	`in` `not in`	Membership operators
	`and` `or` `xor`	Logical operators
Lowest	`and then` `or else`	Short-circuit operators

Sample expressions:

```
PI                          -- a simple expression
(B**2) - (4.0 * A * C)
B**2 - 4.0*A*C              -- same meaning as the above
CH in 'a' .. 'z'            -- a boolean expression
24.2**3                     -- a static expression
(not SUNNY) or WARM         -- a boolean expression
not SUNNY or WARM           -- same meaning as the above
not (SUNNY or WARM)         -- different from above
"Hello" & "  " & "Joe"      -- string concatenation
b > 0 and then a/b < 5      -- short circuit boolean expression
```

Statements

A *statement* is a sequence of characters terminated by a semicolon (;).

```
Value := Value + 1;  -- an assignment statement


Value

:=

2

;    -- another assignment statement


Value := 2;  -- same as the last statement
```

Statements, Continued

- sequential control
  - assignment
  - block
  - null
  - return
  - procedure call
- conditional control
  - case
  - if

These are all the kinds of statements recognized by Ada compilers.

- iterative control
  - exit
  - loop
- other statements
  - abort
  - accept
  - code
  - delay
  - entry call
  - goto
  - raise
  - select

Statements: Sequential Control

- *assignment*

```
Value := 1;
Value :=
  SQRT(B**2 + A**2);
```

- *block*

```
declare  -- vars local to block
  local_1 : integer;
begin  -- code of the block
  local_1 := 2;
  value := value / local_1;
end;  -- end of the block
```

- *null*

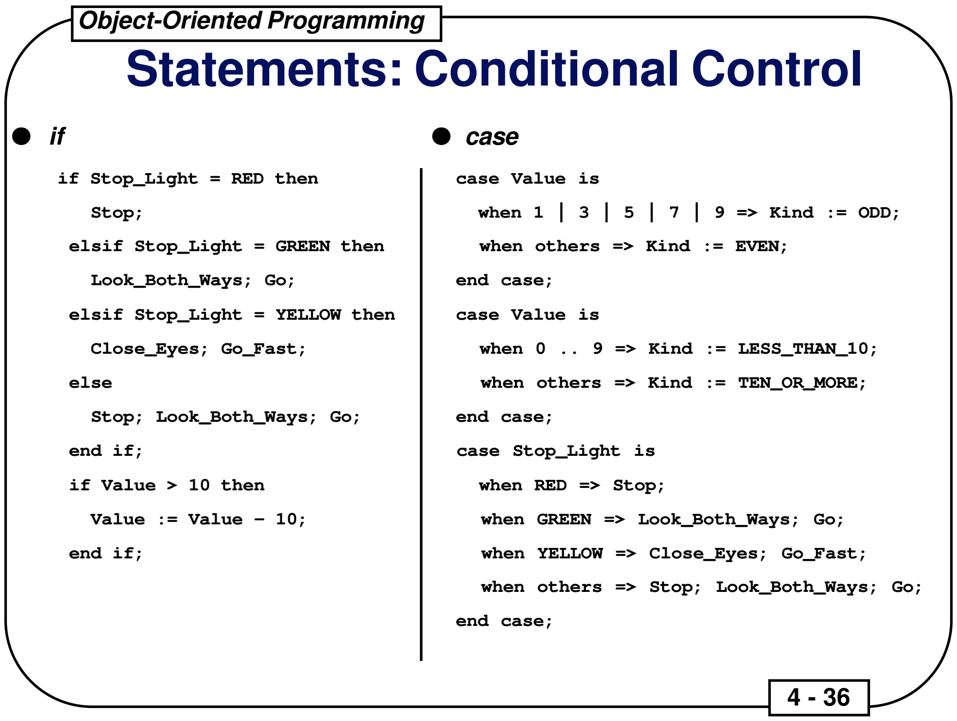
```
null;
```

- *return*

```
return;
return PI*2.0;
```

- *procedure call*

```
Text_IO.Put_Line("Hello");
Put ("Enter text: ");
Stacks.Push(100.0, My_Stack);
```



Statements: Conditional Control

- *if*

```
if Stop_Light = RED then
   Stop;
 elsif Stop_Light = GREEN then
   Look_Both_Ways; Go;
 elsif Stop_Light = YELLOW then
   Close_Eyes; Go_Fast;
 else
   Stop; Look_Both_Ways; Go;
 end if;
 if Value > 10 then
   Value := Value - 10;
 end if;
```

- *case*

```
case Value is
  when 1 | 3 | 5 | 7 | 9 => Kind := ODD;
  when others => Kind := EVEN;
end case;
case Value is
  when 0 .. 9 => Kind := LESS_THAN_10;
  when others => Kind := TEN_OR_MORE;
end case;
case Stop_Light is
  when RED => Stop;
  when GREEN => Look_Both_Ways; Go;
  when YELLOW => Close_Eyes; Go_Fast;
  when others => Stop; Look_Both_Ways; Go;
end case;
```

Notes on Case Statements:

- The case expression must be of a discrete type (Integer or Enumeration).
- Every possible value of the case expression must be covered in one and only one **when** clause.
- If the **when others =>** clause is used, it must appear as a single choice at the end of the case statement.
- Choices in a **when** clause must be static (able to be resolved at compile time).

Statements: Iterative Control

- *two kinds of exit statements*

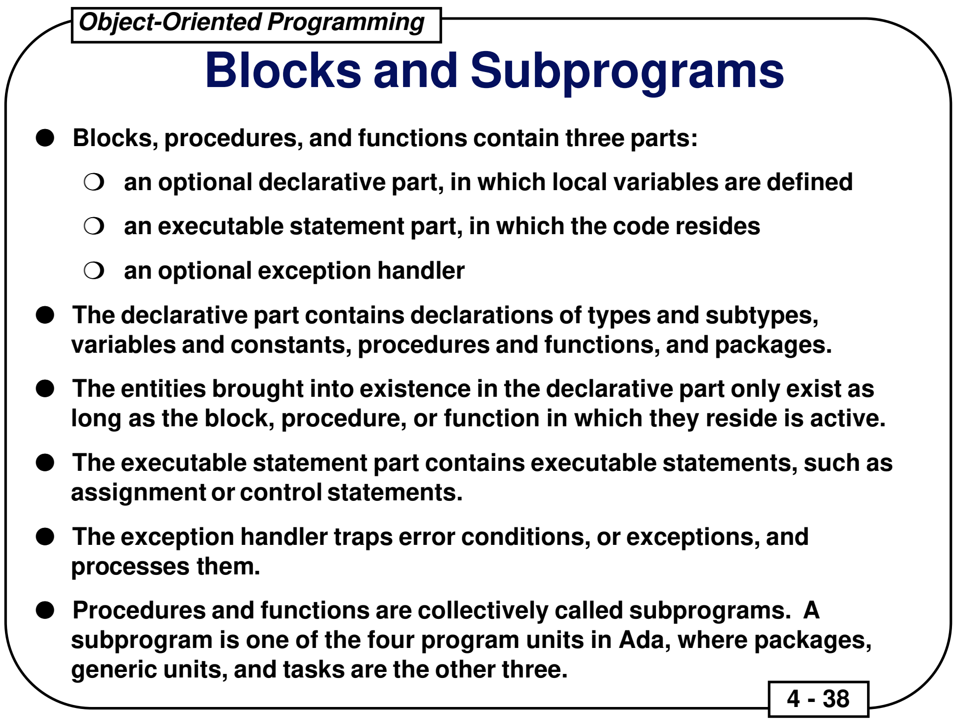
```
exit;  -- unconditional

exit when A = 0;  -- conditional
```

- *three kinds of loops*

```
loop  -- simple loop
  Bit := Status_Bit;
  exit when Bit = ON;
end loop;
```

```
while Status_Bit = OFF loop
  null;  -- while loop
end loop;
```

```
i := 42;
for i in 1 .. 20 loop  -- for loop, outer I  is hidden
  sum := sum + i;
end loop;
sum := sum + i;  -- outer I  is visible again
```



Blocks and Subprograms

- **Blocks, procedures, and functions contain three parts:**
  - **an optional declarative part, in which local variables are defined**
  - **an executable statement part, in which the code resides**
  - **an optional exception handler**
- **The declarative part contains declarations of types and subtypes, variables and constants, procedures and functions, and packages.**
- **The entities brought into existence in the declarative part only exist as long as the block, procedure, or function in which they reside is active.**
- **The executable statement part contains executable statements, such as assignment or control statements.**
- **The exception handler traps error conditions, or exceptions, and processes them.**
- **Procedures and functions are collectively called subprograms. A subprogram is one of the four program units in Ada, where packages, generic units, and tasks are the other three.**

The Basic Differences Between Subprograms and Blocks

- Subprograms can be compiled separately, while blocks are embedded in some larger unit which must be compiled as a whole.
- Embedded subprograms can only be placed in the declarative part of a unit, while blocks can only be placed in the statement part.
- Subprograms can be invoked by a call, while blocks are invoked as part of the flow of execution only.

Blocks

The general form of a block:

```
declare  -- optional
  -- variable definitions
begin
  -- statements
  null;
exception
  -- exception handler
end;
```

Subprograms

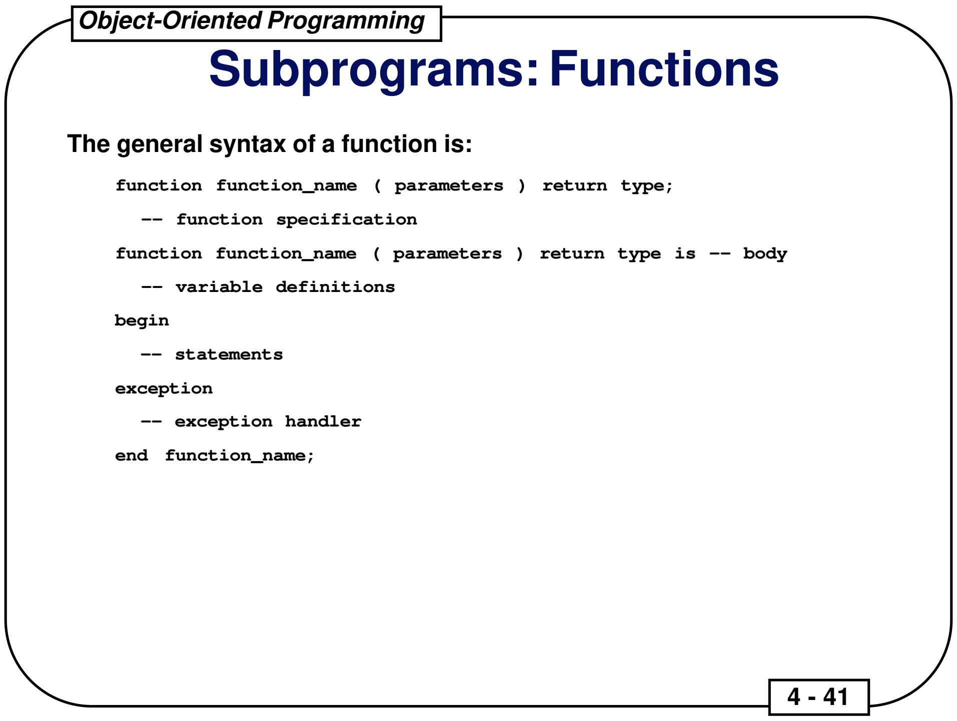
Subprograms are the basic units of sequential execution in an Ada system.

There are two classes of subprograms:

- *procedures* -- accept and return values in parameter lists
- *functions* -- accept values in parameter lists and only return one value

Parameter lists contain three classes of formal parameters:

- *in* -- parameter values are passed into subprograms
- *out* -- parameter values are passed out of subprograms (procedures only)
- *in out* -- parameter values are passed both ways (procedures only)



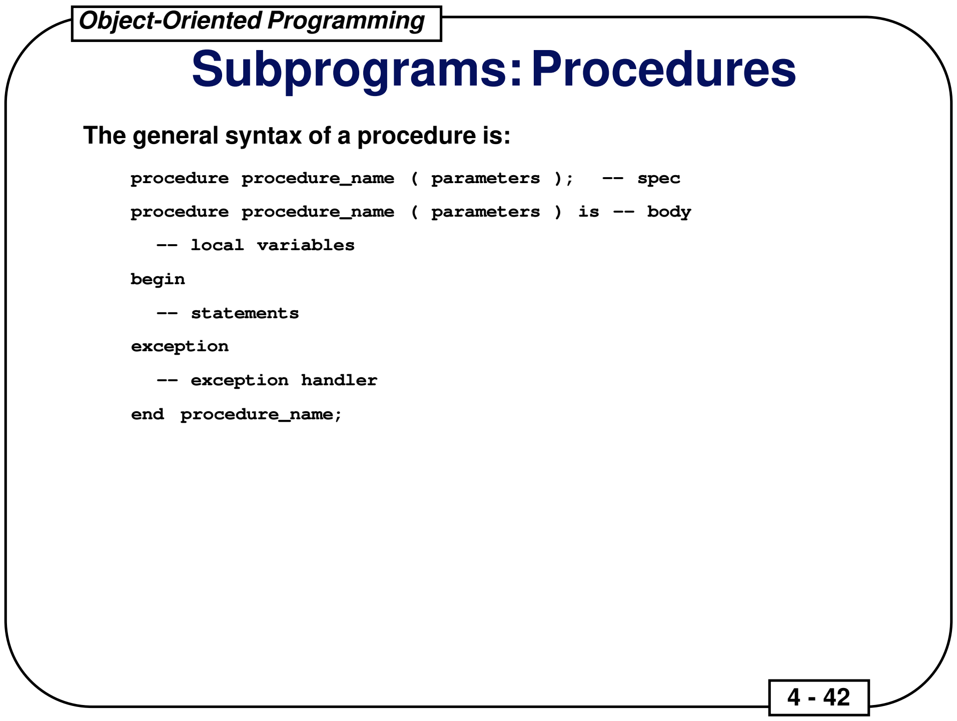
Subprograms: Functions

The general syntax of a function is:

```
function function_name ( parameters ) return type;
  -- function specification
function function_name ( parameters ) return type is -- body
  -- variable definitions
begin
  -- statements
exception
  -- exception handler
end  function_name;
```

Examples of Functions and Function Calls

```
function Sin (Angle : in FLOAT) return FLOAT;  -- spec
function Sin (Angle : in FLOAT) return FLOAT is -- body
begin
  null;  -- detail omitted
  return 1.0;  -- dummy return value
end Sin;
function Cos (Angle : FLOAT) return FLOAT; -- mode is 'in' by default
function "*" (Left, Right : in COMPLEX_NUMBER) return COMPLEX_NUMBER;

-- Examples of calls
X := Sin (2.2);
X := Cos(Angle => 45.2);
Y := Trig_Lib.Cos(X);
C3 := Complex."*" (C1, C2);
C3 := C1 * C2;
```



Subprograms: Procedures

The general syntax of a procedure is:

```
procedure procedure_name ( parameters );  -- spec
procedure procedure_name ( parameters ) is -- body
  -- local variables
begin
  -- statements
exception
  -- exception handler
end  procedure_name;
```

Examples of Procedures and Procedure Calls

```
procedure Get_Status (Result : out STATUS); -- spec
procedure Get_Status (Result : out STATUS) is -- body
begin
  Result := OK;
end Get_Status;

procedure Create (File : in out FILE_TYPE;  -- spec
  Name : in STRING := "DUMMY.TXT";
  Mode : in FILE_MODE := IN_FILE);

-- Procedure calls
Get_Status (Value);
Get_Status (Result => Value);
Create(FD);
Create (FD, Mode => OUT_FILE);
Create (FD, "T.T", INOUT_FILE);
Create (File => FD, Name => "Myfile.txt", Mode => IN_FILE);
```

Notes on Subprograms

- ***Overloading*: Subprogram names may be overloaded (i.e., two or more subprograms may have the same names but different types or numbers of parameters), and Ada can resolve these from context.**
- ***Recursion*: A subprogram may call itself, or recurse.**

Packages

A *package* is an encapsulation mechanism in Ada, allowing the programmer to collect groups of entities together. As a rule, these entities should be logically related. A *package* usually consists of two parts: a specification and a body.

Packages directly support object-oriented programming, providing a means to describe a class or object (an *abstract data type*).

A package may implement either an object as an abstract state machine (wherein the state information is stored as hidden data in the package body) or a class using private data types (wherein the state information is stored as private data associated with each object).

```
package Console is -- abstract state machine
  type LOCATION is record
    row : INTEGER;
    col : INTEGER;
  end record;
  procedure Clear_Screen;
  procedure Position_Cursor (Where : in LOCATION);
  procedure Write (Item : in CHARACTER);
  procedure Write (Item : in STRING);
end Console;
package Complex is -- class definition
  type OBJECT is private;
  function Set (Real : in FLOAT; Imag : in FLOAT) return OBJECT;
  function Real_Part (Item : in OBJECT);  function Imag_Part (Item : in OBJECT);
  function "+" (Left, Right : in OBJECT) return OBJECT;
  function "-" (Left, Right : in OBJECT) return OBJECT;
private
  type OBJECT_TYPE; -- deferred to body
  type OBJECT is access OBJECT_TYPE;
end Complex;
```

Package Specifications and Bodies

The general form of a package specification is:

```
package package_name is
  -- visible declarations
private
  -- private type declarations
end package_name;
```

The general form of a package body is:

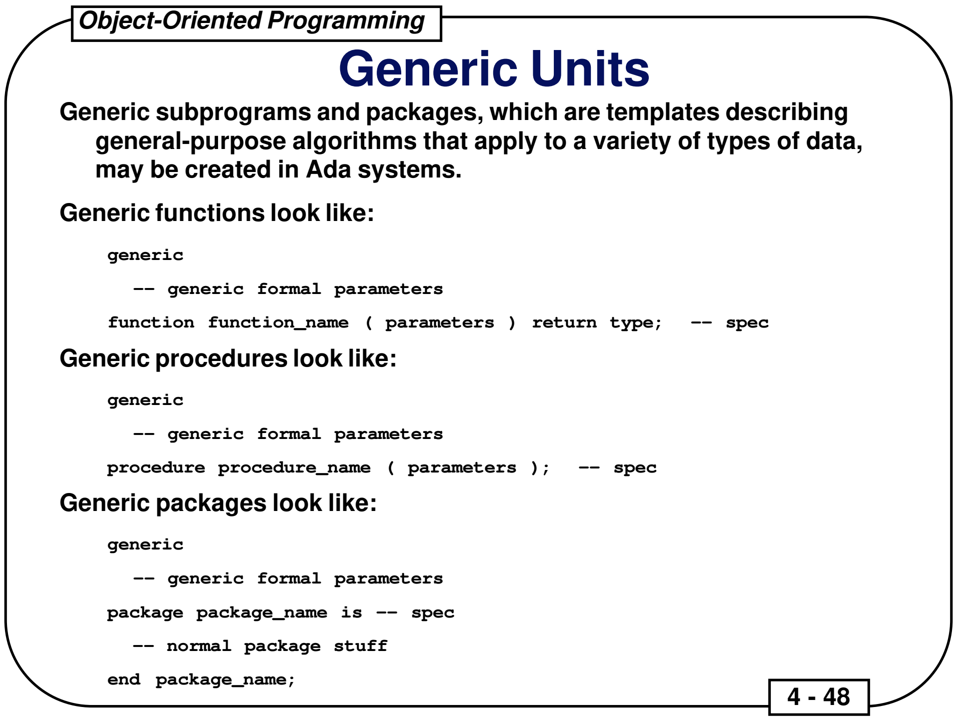
```
package body package_name is
  -- implementations of code and hidden data
begin
  -- initialization statements
end package_name;
```

Uses of Packages

- Collections of constants and type declarations
- Collections of related functions
- Abstract State Machines
- Abstract Data Types

Notes on Packages

- Package bodies may contain an optional initialization part. If this is present, the code of the initialization part of a package is executed before the first line of code in the mainline procedure.
- Packages may be embedded in: blocks, subprograms, other packages, and any program unit in general.
- A *private type* is a type definition which is visible in the specification of a package, but its underlying implementation is hidden from the code *withing* the package and is of no concern to the outside world.
- *Private types* are the means of implementing *abstract data types* in Ada. In a package containing a private type, the only operations which may be performed on objects of that type are assignment, tests for equality and inequality, and the procedures and functions explicitly exported by the package.
- In a package containing a *limited private type*, the only operations which may be performed on objects of that type are the procedures and functions explicitly exported by the package.



Generic Units

Generic subprograms and packages, which are templates describing general-purpose algorithms that apply to a variety of types of data, may be created in Ada systems.

Generic functions look like:

```
generic
  -- generic formal parameters
function function_name ( parameters ) return type;  -- spec
```

Generic procedures look like:

```
generic
  -- generic formal parameters
procedure procedure_name ( parameters );  -- spec
```

Generic packages look like:

```
generic
  -- generic formal parameters
package package_name is -- spec
  -- normal package stuff
end package_name;
```

The bodies of generic functions and procedures are as for normal subprograms except that the general types used in the specifications are employed rather than conventional types.

An example:

```
generic
  type ELEMENT is private;  -- anything that can be assigned may be used
procedure Exchange (Item1, Item2 : in out ELEMENT);  -- spec

procedure Exchange (Item1, Item2 : in out ELEMENT) is -- body
  Temp : ELEMENT;  -- temporary is of the same type as the parameters
begin
  Temp := Item1;  -- this works for any thing that may be assigned
  Item1 := Item2;
  Item2 := Temp;
end Exchange;
```

Generic Formal Parameters

- There are three kinds of generic formal parameters: types, objects, and subprograms.
- Types as generic formal parameters:

Type Parameter	*Operations Allowed*	*Data Types*
type T is private;	= /= :=	All assignable
type T is limited private;	--none--	All
type D is (<>);	= /= := > >= < <=	Discrete
	PRED SUCC FIRST LAST	
type I is range <>;	integer operations	Integer
type F is digits <>;	real operations	Float
type FIXED is delta <>;	fixed point operations	Fixed

- Object declarations may appear as formal parameters.
- Subprograms may appear as formal parameters.

Tasks

In Ada, one can write programs that perform more than one activity concurrently. This concurrent processing is called *tasking*, and the units of code that run concurrently are called *tasks*.

- **A simple format for task specifications and bodies:**

```
task task_name;   -- specification
task body task_name is -- body
  -- local variable declarations
begin
  -- code
end task_name;
```

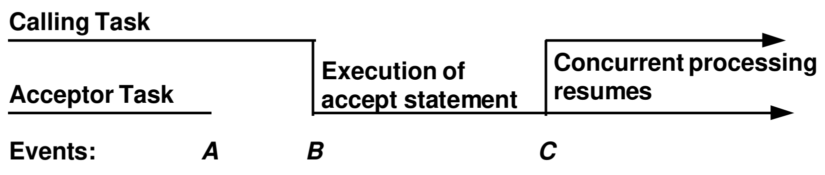
- **A more complex format:**

```
task task_name is -- spec
  entry entry_name ( parameters );
end task_name;
task body task_name is -- body
begin
  accept entry_name ( parameters ) do -- code follows
  end entry_name;
end task_name;
```

The **entry** statement in the task specification identifies the entry points to the task. The **accept** statement in the task body identifies the code to be executed at that entry point. The entry points to a task are called like subprogram calls from other program units.

Tasks That Rendezvous

The interfacing of two tasks in order to pass data is called a *rendezvous* in Ada. The following is a representative timeline for two such tasks:



Key to Events --

A Acceptor task reached an accept statement and is waiting for a call to its entry point.

B Calling task calls the Acceptor task at its entry point, and the Acceptor task executes code in the accept statement.

C The accept statement is completed, data is transferred back to the Calling task if necessary, and both tasks resume concurrent operation.

Exceptions

- **Two kinds of errors are commonly encountered in programming: compilation errors and runtime errors.**
- **In Ada, runtime errors are called *exceptions*. Exceptions may be predefined or user-defined. To define an exception:**

```
Exception_Name  :  exception;
```

To raise an exception:

```
raise  Exception_Name;
```

- ***Exception handlers* are Ada constructs that handle *exceptions*. An *exception handler* is placed at the end of a block, subprogram, package, or task, and is denoted by the keyword exception followed by the text of the handler code. Example (for a block):**

```
begin  -- note that I is defined external to the block
  I := I / 0;  -- division by zero
exception
  when NUMERIC_ERROR =>
    I := 10;
end;
```

Predefined Exceptions -- Package STANDARD

- CONSTRAINT_ERROR -- raised whenever a value goes out of bounds, such as assigning a value of 11 to a variable whose type is in the range from 0 to 10
- NUMERIC_ERROR -- raised when illegal or unmanageable mathematical operations are performed, such as dividing by zero
- STORAGE_ERROR -- raised when the computer runs out of available memory
- TASKING_ERROR -- raised when there is a problem with the multitasking environment, such as calling a task which is no longer active
- PROGRAM_ERROR -- all other exceptions not covered by the above or other exceptions defined by the user or some other package, such as reaching the end of a function without hitting a return statement

Predefined Exceptions -- Package TEXT_IO

- DATA_ERROR -- encountered an input that is not of the required type
- DEVICE_ERROR -- malfunction in the underlying system, such as disk space being full
- END_ERROR -- an attempt was made to read past the end of a file
- LAYOUT_ERROR -- raised by COL, LINE, or PAGE if the value returned exceeds COUNT'LAST
- MODE_ERROR -- an attempt was made to read from or test the end of a file whose current mode is OUT_FILE, or an attempt was made to write to a file whose current mode is IN_FILE
- NAME_ERROR -- the string given as a file name to CREATE or OPEN does not allow the identification of an external file
- STATUS_ERROR -- an attempt was made to operate on a file that is not open or open a file that is already open
- USE_ERROR -- an operation is attempted that is not possible for reasons that depend on the characteristics of the external file

Exception Propagation

- If the program unit that raises an exception does not contain an exception handler that handles the exception, the exception is propagated to the next level beyond the unit. This level varies, depending on the unit raising the exception:
  - ❍ If the unit is a mainline procedure, the Ada runtime environment handles the exception by aborting the program.
  - ❍ If the unit is a block, the exception is passed to the program unit (or block) containing the block that raised the exception.
  - ❍ If the unit is a subprogram, the exception is passed to the program unit or block that called the subprogram.
- The propagation path of an exception is determined at runtime.
- To reraise the current exception in an exception handler, the statement

```
raise;
```

may be used.

Suppressing Exceptions

Ada performs many checks at runtime to ensure that array indices are not exceeded, variables stay within range, etc. If these checks fail, *exceptions* are raised.

This results in larger code and slower execution speed.

In certain real-time applications, where space and time constraints are critical, runtime error checking may be too expensive. A solution is to use *exception suppression*.

Exception suppression turns off runtime error checking. It is implemented by a *pragma* (a compiler directive) called SUPPRESS:

```
pragma SUPPRESS (RANGE_CHECK);
  -- turns off range checking on array indices and variable values
pragma SUPPRESS (RANGE_CHECK, INTEGER);
  -- turns off range checking on integers only
pragma SUPPRESS (RANGE_CHECK, X);
  -- turns off range checking for a particular object
```