OBJECT-BASED PROGRAMMING LANGUAGES

-- Ada --

Ada Program Units and Subprograms

Main Programs

Ada Program Unit Lib

Character Sets

Lexical Units

Package STANDARD

Operators

Statements

Reserved Words

Blocks

Packages

Types

Generic Units

Tasks

Exceptions

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**

Program Units: SUBPROGRAM

A *subprogram* is an expression of sequential action.

Two kinds of subprograms exist:

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

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

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**

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**

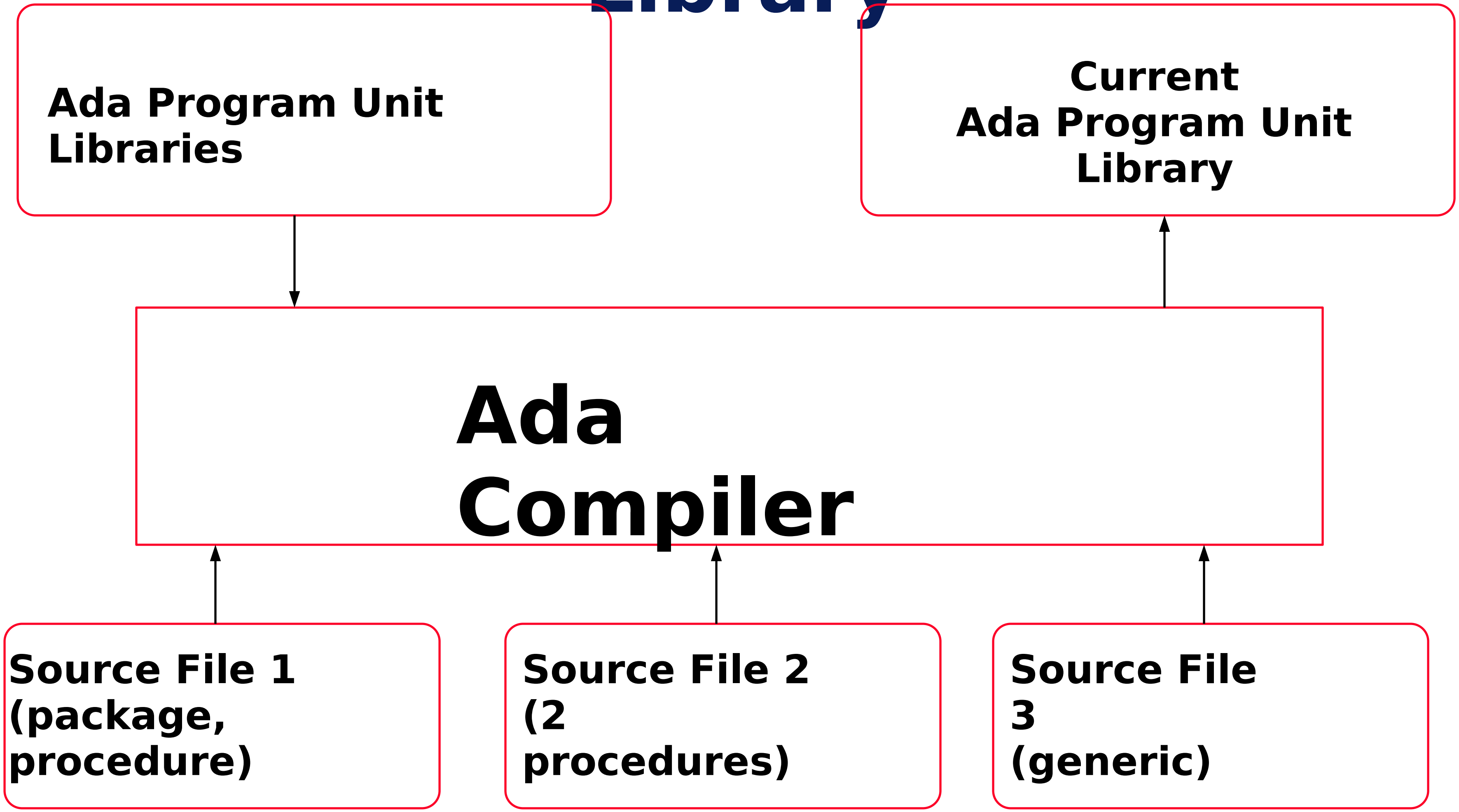
Procedures as Main Programs

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

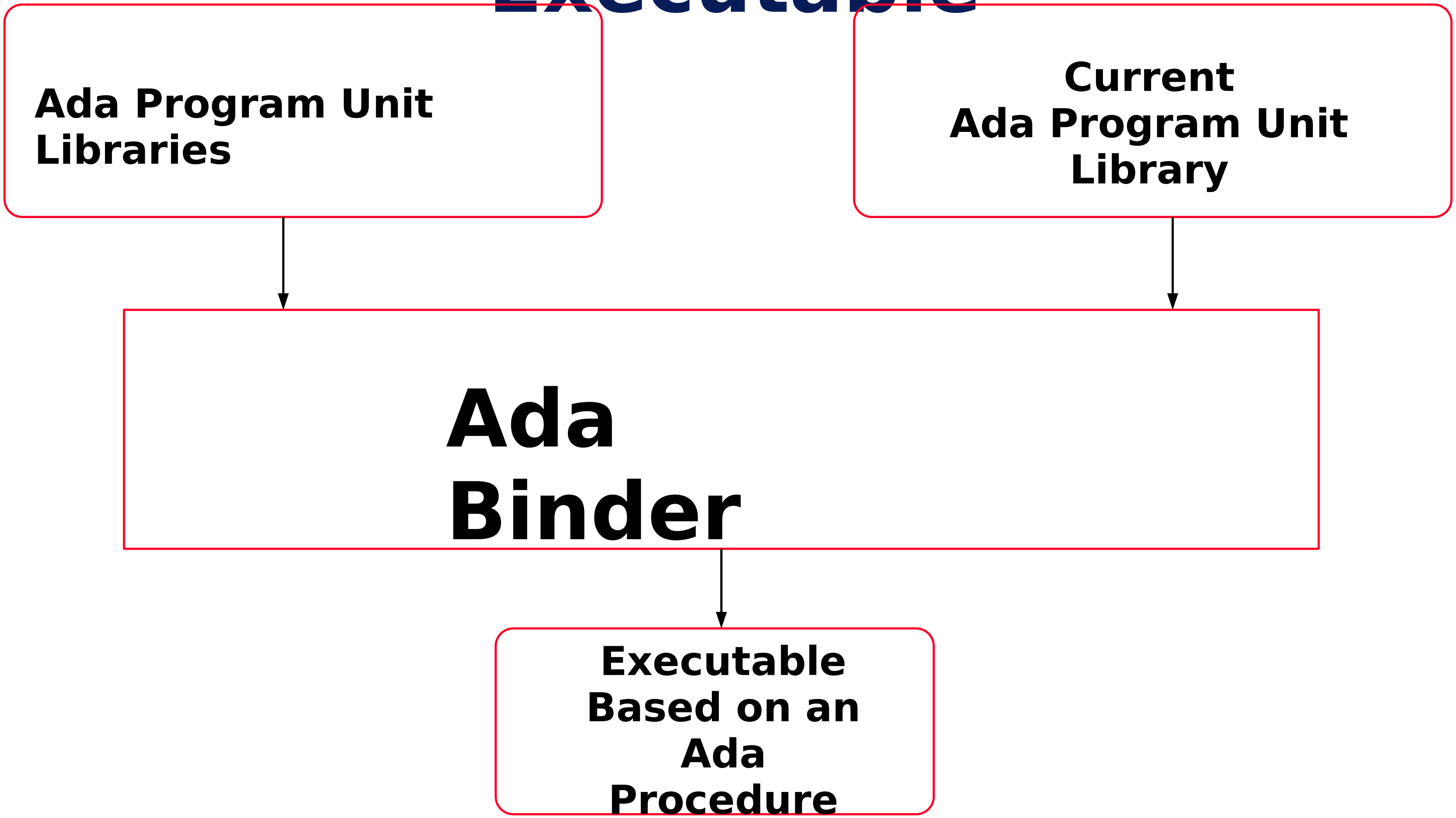
Ada Compiler

Source File 1 (package, procedure)

Source File 2 (2 procedures)

Source File 3 (generic)

Creating an Executable



Ada Program Unit Libraries

Current Ada Program Unit Library

Ada Binder

Executable Based on an Ada Procedure

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
```

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.).**

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 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**
- **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

Enumeration		Integer	Real
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
```

Subtypes and Derived Types

l *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;
```

l *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
```

l *Derived types* are different from subtypes:

m A derived type introduces a new type, distinct from its parent.

m A subtype places a restriction on an existing type, compatible with its parent.

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**

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', ' ');
```

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:**

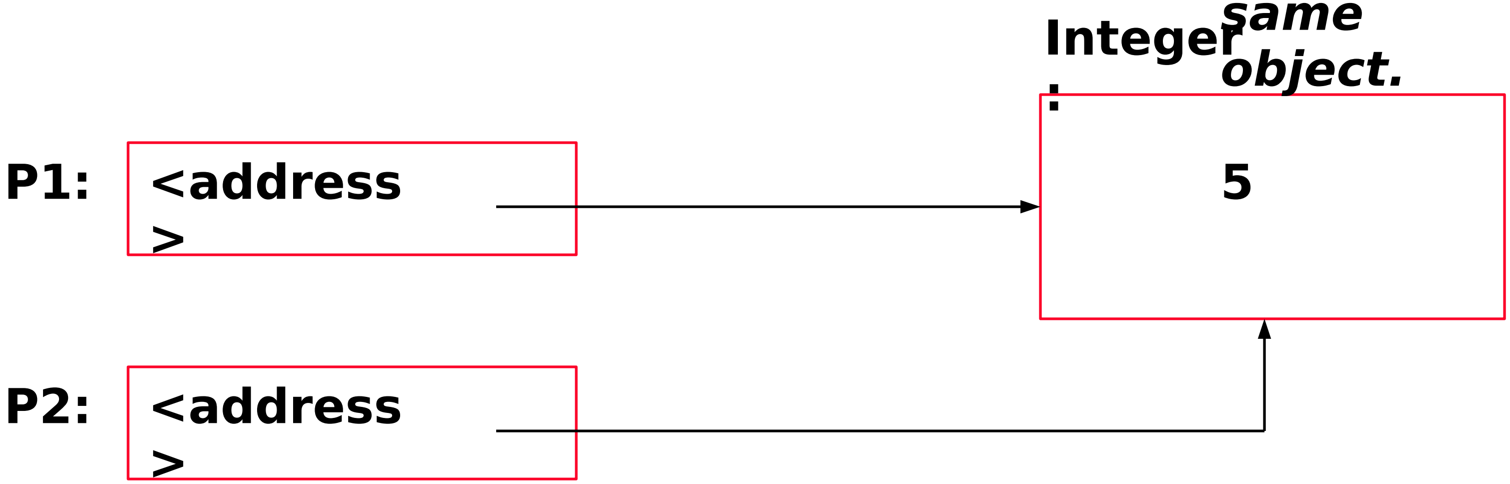
```
type RECORDING_MEDIUM is (PHONOGRAPH, CASSETTE, CD);
type MUSIC_TYPE is (CLASSICAL, JAZZ, NEW_AGE, FOLK, POP);
type RECORDING (Device : RECORDING_MEDIUM := CD) is record
 Music : MUSIC_TYPE;
 case Device is
   when PHONOGRAPH =>
     Speed : RPM;
   when CASSETTE =>
     Length : NATURAL;
   when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


Integer:

5

P1: **<address>**

P2: **<address>**

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
```

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**

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

l sequential control l
iterative control

m assignment m
exit

m block m loop

m null

m return l other
statements

m procedure call
m abort

m accept

l conditional control
m code

m case m delay

m if m entry call

m goto

m raise

m select

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

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***
- ***case***

```
if Stop_Light = RED then        case Value is
  Stop;                  when 1 | 3 | 5 | 7 | 9 =>
Kind := ODD;
 elsif Stop_Light = GREEN then              when
others => Kind := EVEN;
  Look_Both_Ways; Go;           end case;
 elsif Stop_Light = YELLOW then     case Value is
  Close_Eyes; Go_Fast;             when 0 .. 9 =>
Kind := LESS_THAN_10;
 else              when others => Kind :=
TEN_OR_MORE;
  Stop; Look_Both_Ways; Go;         end case;
 end if;             case Stop_Light is
 if Value > 10 then              when RED => Stop;
  Value := Value - 10;             when GREEN =>
Look_Both_Ways; Go;
 end if;                when YELLOW => Close_Eyes;
Go_Fast;
      when others => Stop; Look_Both_Ways; Go;
    end case;
```

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

l Blocks, procedures, and functions contain three parts:

m an optional declarative part, in which local variables are defined

m an executable statement part, in which the code resides

m an optional exception handler

l The declarative part contains declarations of types and subtypes, variables and constants, procedures and functions, and packages.

l 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.

l The executable statement part contains executable statements, such as assignment or control statements.

l The exception handler traps error conditions, or exceptions, and processes them.

l 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.

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;
```

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;
```

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*).

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;
```

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 (<>);	= /= := > >= < <= PRED SUCC FIRST LAST	Discrete
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:**

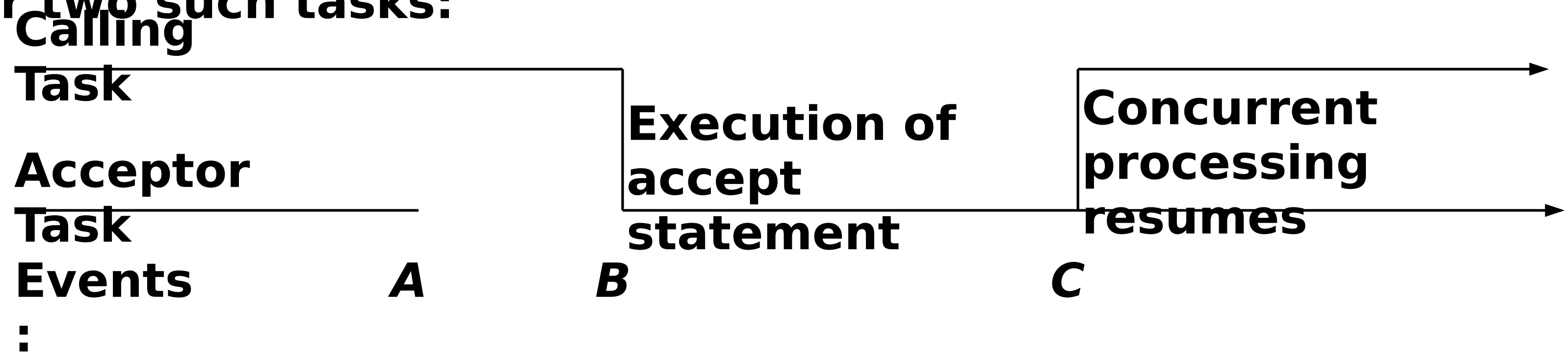
```
task task_name;  -- specification
task body task_name is -- body
 -- local variable declarations
begin
 -- code
end task_name;
```

- **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;
```

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:



Calling Task

Acceptor Task

Execution of accept statement

Concurrent processing resumes

Events: ***A*** ***B*** ***C***

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;
```

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
```