Example (direct visibility)

```
Package MEASURES is
    type AREA is private;
    type LENGTH is private;
    function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
    function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
    type LENGTH is range 0..100;
    type AREA is range 0..10000;
end MEASURES;
---------------------------------------------------------------------------------
with MEASURES;  use MEASURES;  --direct visibility
procedure MEASUREMENT is
    SIDE1,SIDE2  :  LENGTH;
    FIELD :  AREA;
begin
    …...
    FIELD := SIDE1 * SIDE2;  --NOTE:  Infix notation of user-defined operation
end MEASUREMENT;
```

Example (indirect visibility)

```
Package MEASURES is
    type AREA is private;
    type LENGTH is private;
    function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
    function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
    type LENGTH is range 0..100;
    type AREA is range 0..10000;
end MEASURES;
-------------------------------------------------------------------------------
with MEASURES; --NOTE - no “use” clause
procedure MEASUREMENT is
    SIDE1,SIDE2  :  MEASURES.LENGTH;
    FIELD :  MEASURES.AREA;
begin
    …...
    FIELD := MEASURES.”*”(SIDE1, SIDE2);
end MEASUREMENT;
```

Example (Ada95 compromise)

```
Package MEASURES is
     type AREA is private;
     type LENGTH is private;
     function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
     function "*" (LEFT, RIGHT : LENGTH) return AREA;
private
     type LENGTH is range 0..100;
     type AREA is range 0..10000;
end MEASURES;
--------------------------------------------------------------------------------
with MEASURES;  -- NOTE:  no “use” clause
procedure MEASUREMENT is
use MEASURES.Length;       --direct visibility of type
use MEASURES.Area;         --direct visibility of type
     SIDE1,SIDE2  :  LENGTH;
     FIELD :  AREA;
begin
     …...
     FIELD := SIDE1 * SIDE2;  --probably still bad form, but compiles OK
end MEASUREMENT;
```

Example (The best way)

```
Package MEASURES is
      type AREA is private;
      type LENGTH is private;
      function Add_Length (LEFT, RIGHT : LENGTH) return LENGTH;
      function Calc_Area (LEFT, RIGHT : LENGTH) return AREA;
private
      type LENGTH is range 0..100;
      type AREA is range 0..10000;
end MEASURES;
-------------------------------------------------------------------------------
with MEASURES;  -- NOTE:  no "use" clause
procedure MEASUREMENT is

      SIDE1,SIDE2  :  MEASURES.LENGTH;
      FIELD :  MEASURES.AREA;
begin
      …...
      FIELD := Measures.Calc_Area (Side1, Side2);
end MEASUREMENT;
```

Router metrics of Typing

- Predefined vs. User-Define
- Attributes
- Type Conversion
- Scalar, Access and Private Types
- Use of Appropriate Types

Strong typing

Ada enforces strong typing using "name equivalence." (not structural equivalence)

Applies to both predefined and user-defined types

```
procedure TESTER is
    type APPLES is range 0 .. 100;     -- an integer type
    type ORANGES is range 0 .. 100; --ditto
    A : APPLES := 100;                 -- initialize during declaration
    O : ORANGES;
begin
    O := A;                            --  ERROR, incompatible types
end;
```

--> A and O are not compatible in Ada.

TYPING ENFORCES:

- Abstraction, hiding of implementation details (simulates real world events)
  - Properties of objects and operations are separated from underlying and internal implementation - dependent properties
    - Object_A : Fruit; (Don't really care about the details of fruit)

TYPING ENFORCES:

- FACTORIZATION OF PROPERTIES, MAINTAINABILITY
  - Common properties of objects are described and collected in one place
  - A name is associated with that description
  - Can change properties of objects by changing only the type declaration

Typing

- ✓ Typing is the enforcement of the class of an object.
- ✓ It prevents inadvertent conversion of one type to another.
- ✓ Very strong typing prevents the conversion of one type to another.
- ✓ Strong typing requires explicit action on the part of the implementor to “coerce” one type into another.
- ✓ Ada supports strong typing. It provides both automatic and used-defined coercion.

More on Typing

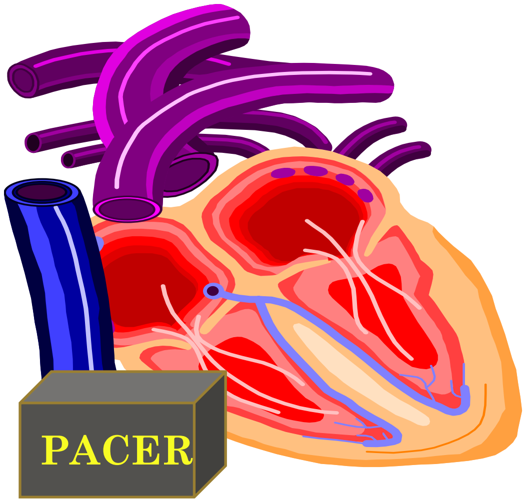
- ✓ Without type checking, a program can crash at run-time for mysterious reasons
- ✓ Typing allows early error detection
- ✓ Type declarations are part of design. Helps with documentation
- ✓ More efficient code can be generated.

FORCES DESIGN DECISIONS TO BE MADE EARLY!!

Strong Typing Example

type Systolic is range 90 .. 140; ***--Systolic -- heart pumping out 90 - 140***

type Diastolic is range 50 .. 90; ***--Diastolic -- heart filling up 50 - 90***




```
Systolic_Init_Value :     constant Systolic := 110;
Diastolic_Init_Value :    constant Diastolic := 60;

Systolic_Reading   :      Systolic := Systolic_Init_Value ;
Diastolic_Reading :       Diastolic := Diastolic_Init_Value;

Systolic_Set_Value  : Systolic := 200; -- Warning of run-time error at compile
Diastolic_Set_Value : Diastolic;

Systolic_Set_Value := Systolic_Reading + Diastolic_Reading;  --Error at compile
```

Problems of Weak Typing

- Theriac 25 X-Ray machine
- Mars Lander (2000)

Types

- ✓ Characteristics
  - –set of values
  - –set of primitive operations

Type Definition

type Some_Type is (definition of what the type is);

Subtypes

- used to further limit values of abstraction
- has all primitive operations of base-type
  - subtype Sub_Type is Some_Type
    range lower..upper;

Objects

A run-time entity that contains a value of the object's type.

Object Definition

Object : Type_Name
[:=initial value];

Classes of Ada Types

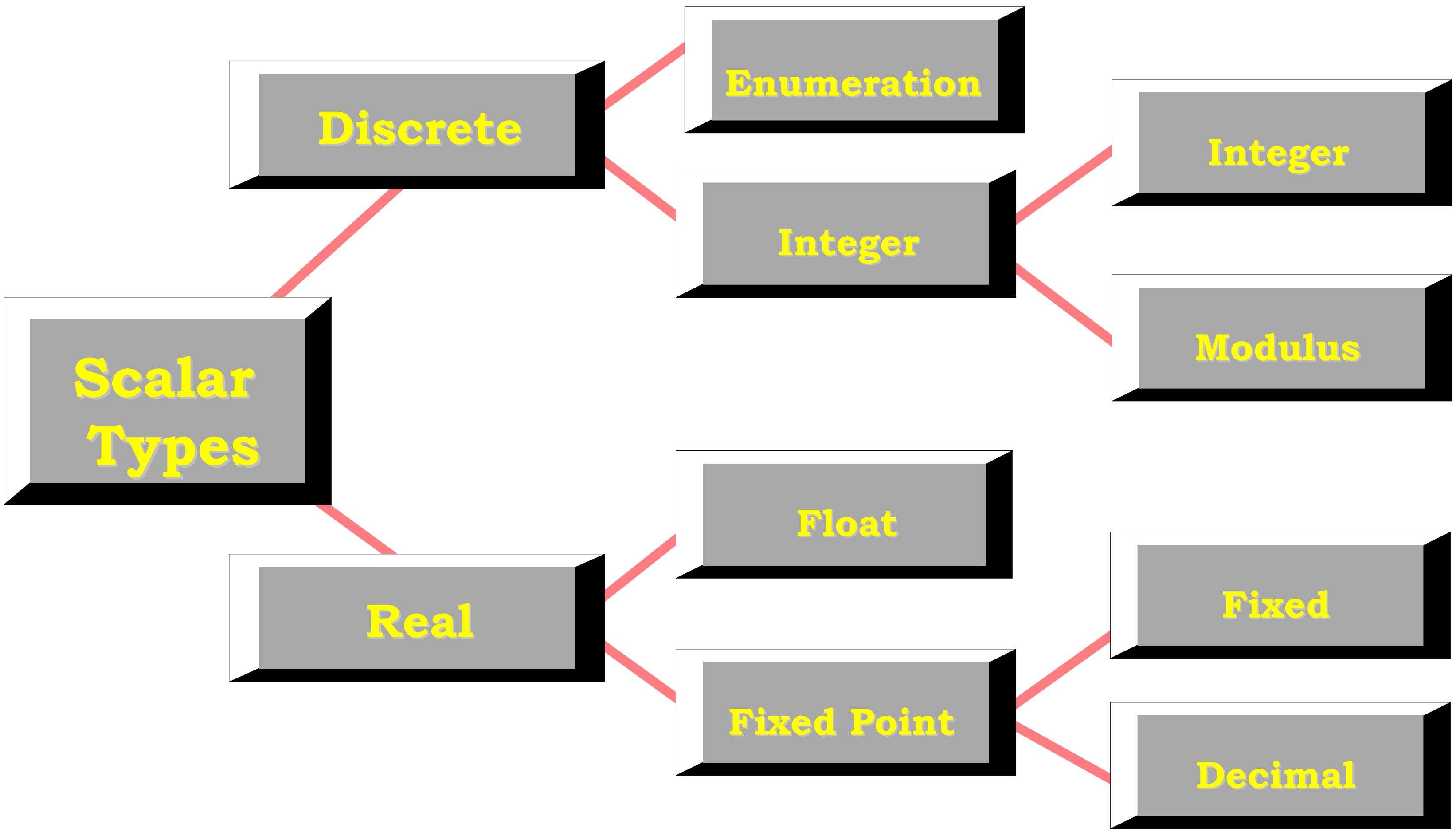
Ada Types

SCALAR	Objects are single values
COMPOSITE	Objects contain other components
PRIVATE	Objects are 'abstract'
ACCESS	Objects 'point' to other objects & subprograms
TASK	Objects are parallel processes
Protected	Coordinated access to shared data
Tagged	Inheritance & Run-time polymorphism

Predefined Types

- **Boolean**
- **Integer**
  - **Natural** (Subtype) 0..Integer'Last
  - **Positive** (Subtype) 1..Integer'Last
  - **Mod** (Modulus)
- **Float**
- **Character**
- **Wide_Character**
- **String**
- **Wide_String**
- **Fixed**
  - **Decimal_Fixed_Point**

Scalar Types



Discrete Types

This type defines only whole values (exact values).

Apple

1

“A”

“a”

Enumeration Types

- allows an abstraction to be represented directly
- can be used in indexing, iteration, case statements and record variants
- Use name of real world items

Enumeration Types

set of values: order set of distinct values

structure: (E1, E2, E3, … , En)

operations: assignment (:=)

membership (in, not in)

relation (=, /=, <, <=, >, >=)

Defining Enumeration Types

```
type Week_Type is (Sun, Mon, Tue,
        Wed, Thu, Fri, Sat);

type European_Week_Type is
        (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

subtype Work_Week_Type is Week_Type
        range Mon..Fri;
```

Predefined Enumeration Types

Boolean (False, True)

Character ISO 10646 256 code set (8 bits)

Latin 1 character set

Wide_Character ISO 10646 65536 code set (16 bits)

Attributes for Enumeration

format: S'Attribute [()]

First	Last	Range	base
Min	Max	Succ	Pred
Val	Pos	Value	
Image	Wide_ Image		
Width	Wide_Width		

Examples of Attributes

Week_Type'First	=	Sun
Week_Type'Last	=	Sat
Week_Type'Pos(Mon)	=	1
Week_Type'Val(0)	=	Sun
Work_Week_Type'Range	=	Mon..Fri

Enumeration Objects

```
Day         : Week_Type  := Tue;

Tomorrow    : Week_Type  :=
              Week_Type'Succ(Day);

Yesterday   : Week_Type  :=
              Week_Type'Pred(Day);

Work_Day    : Week_Type range
              Mon..Fri       := Mon;

Class_Day   : Work_Week_Type;
```

Enumeration Objects

```
Bored    : Boolean      := True;
Yes      : Character    := 'y';

Bell     : Character    :=

      Ada.Character.Latin_1.BEL;
```

Sample Program

```
With Ada.Text_IO;
procedure Main_Driver is

type Week_Type is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
package Week_Type_IO is new Ada.Text_IO.Enumeration_IO(Week_Type);

Day : Week_Type;
                    -- Attributes
begin
    Day := Week_Type ‘First;           -- Is Mon
    Week_Type_IO.Put(Day);
    Day := Week_Type 'Succ (Thu);      -- Is Fri
    Day := Week_Type 'Succ (Sun);      -- Error
end Main_Driver;
```

Representation Specs

Address	Object'address
Alignment	Object'alignment
Size (Object)	Object'size
Size (type)	Type'size

internal code (numeric) representation

for Type use (E1 => 1, E2 => 3, …En => 999);

--entries must be in ascending order

Integer Types

consecutive numeric literals that do not have a radix point

two forms

decimal

12000 12e3 12_000

based (any base from 2 to 16)

2#1110_0000# 16#e#e1

Integer Type

Set of Values:	set of consecutive numeric literals	
Structure:	range L..U (System.Min_Int..System_Max is the greatest range)	
Set of Operations:	assignment	(:=)
	membership	(in, not in)
	relation	(=, /=, <, <=, >, >=)
	math	(+, -, *, /, mod, rem)
	unary	(+, -, abs)
	exponent	(**)

Integer Rem and Mod

A	B	A/B	A rem B	A mod B
10	5	2	0	0
11	5	2	1	1
12	5	2	2	2
13	5	2	3	3
14	5	2	4	4

A	B	A/B	A rem B	A mod B
-10	5	-2	0	0
-11	5	-2	-1	4
-12	5	-2	-2	3
-13	5	-2	-3	2
-14	5	-2	-4	1

A	B	A/B	A rem B	A mod B
10	-5	-2	0	0
11	-5	-2	1	-4
12	-5	-2	2	-3
13	-5	-2	3	-2
14	-5	-2	4	-1

A	B	A/B	A rem B	A mod B
-10	-5	2	0	0
-11	-5	2	-1	-1
-12	-5	2	-2	-2
-13	-5	2	-3	-3
-14	-5	2	-4	-4

Modulus Type

Set of Values:	set of consecutive numeric literials starting at 0	
Structure:	mod U (0..System_Max *2 + 2 is the greatest range)	
Set of Operations:	assignment	(:=)
	membership	(in, not in)
	relation	(=, /=, <, <=, >, >=)
	math	(+, -, *, /, mod, rem)
	unary	(+, -, abs)
	exponent	(**)
	(Auto wrap)	

Defining Integer Types

```
type Pages_Type is
            range 0..System.Max_Int;
subtype Accnt_Type is integer
                  range -1000..100_000;
type Byte is mod 255; -- an unsigned byte
type Hash_Index is mod 97;
```

Predefined Integer Types

Integers

- Integer range (-2**15)+1..(2**15)-1
- Natural range 0..Integer'Last (subtype)
- Positive range 1..Integer'Last (subtype)
- Long_Integer range (-2**31)+1..(2**31)-1

Modulus

- NONE

Attributes for Integers

format: S'Attribute [()]

First Last	Range	Base	
Min	Max	Succ	Pred
Val	Pos	Value	
Modulus	Image	Wide_ Image	
Width	Wide_Width		

INTEGER Objects

type Hour_Type is range 0 .. 12;

type Minute_Type is range 0 .. 59;

Integer.Object Declaration:

Hour : Hour_Type;

Minutes : Minute_Type : = 0;

Hour	Minutes
undef	***0***

INTEGER Attributes

```
With Ada.Text_Io;
Procedure Test is
Type Hours is range 0 .. 12;
Type Days is ( Mon, Tues, Wed, Thr, Fri, Sat, Sun );
-- My_Int : Integer := 0;
-- My_Hour : Hours := Hours'Last;

--Hours'First          – 0
--Hours' Last          – 12
--Hours'Succ(10)       -- 11
--Hours'Succ(12)       -- Error
--Days'Val (3)         --Thr
--Hours'Val (3)        -- 3
--Hours'Image          -- "12"
--Days'Image (Mon)     -- "Mon"
Begin
    ada.text_io.put_line (Hours'Image (Hours'Val (2) ));
    ada.text_io.put_line (Days'Image (Days'Val (2) ));

My_Int : Hour_Type := Hour_Type'First;
```

Sample Program

```
procedure Main_Driver is
   type ALTITUDE is range 0 .. 100;
   type DEPTH is range -100 .. 0;
   type DISTANCE is range 0 .. 200;

   METERS : ALTITUDE := 10;
   FATHOMS : DEPTH    := -25;
   MILES : DISTANCE   := 50;

begin
    FATHOMS:= 10;                              -- error
    MILES       := MILES + 50;
    METERS  :=  METERS + FATHOMS    -- error
    MILES       :=  MILES + DISTANCE (Meters);
end Main_Driver;
```

EXPLICIT TYPE CONVERSION

Representation Specs

Address	Object'address
Alignment	Object'alignment
Size (Object)	Object'size
Size (type)	Type'size

```
for Some_Name use expression;
for Some_Name use Name;
```

Rep Specs Examples

```
type Medium is range 0..65_000;

for Medium use 2*Byte;

Device_Register : Medium;

for Device_Register use Medium'Size;

for Device_Register use
   System.Storage_Elements.To_Address
   (16#FFFF_0020#);
```

Discreet Summary

- ✓ an ordered set of distinct values
- ✓ a Primitive set of operations
- ✓ a set of attributes to enhance understandability

Real Types

- Ada real numbers give only approximate representation of quantities.
  - A real type defines a set of model numbers that can be represented exactly.
  - The accuracy of predefined real types will vary among implementations.

Real Types

- For portability and for sake of abstraction, Ada allows you to define the error bounds of real types:
  - relative accuracy - float e.g. space systems
  - absolute accuracy - fixed e.g. voltage measuring equipment, Money

Real Types

- consecutive numeric literals that have a radix points
- two form
  - decimal
    - 98.6 9.86e1 0.986e2
  - Based (bases 2 thru 16)
    - 2#1010_1101.1010# 16#A.da#e1

Floating Point

- an approximation of real numbers based on the number of digits
- accuracy is at least the precision of the number of decimal digits representable by objects of that type.

type Some_Name is digits X [range L..U];

Float Type

Set of Values:	set of approximations of real Numbers
Structure:	digits N [range L.X..U.X]
Set of Operations:	assignment :=
	membership in not in
	relation = /= < <= > >=
	math + - * /
	unary + - abs
	exponentiating **
	Math Functions (see A.5.1)
	Random Number (see A.5.2

Attributes for Floats

Address	Base	Ceiling	Compose
Copy_Sign	Denorm	Digits	First
Exponent	Floor	Fraction	Image
Leading_Part	Last	Model	Max
Machine_Emax	Machine_Emin		Machine
Machine_Radix	Machine_Overflow		Min
Machine_Rounds	Machine_Mantissa		Pred
Model_Emin	Model_Epsilon		Model_Small
Remainder	Rounding		Safe_First
Safe_last	Scaling	Size	Succ
Signed_Zero	Truncation	Unbiased_Rounding	
Valid	Wide_Image	Wide_Value	Value
Wide_Width	Width	Adjacent	

Floating Point Type Attributes

T ' Digits -- # of decimal digits in mantissa
T ' Mantissa
T ' Epsilon
T ' Emax = 4 * T ' Mantissa
T ' Model_Small = 2.0 ** (-T ' Emax-1)
-- smallest possible model #
T ' ??? = 2.0 ** T ' EMAX * (10 - 20** (-T ' MANTISSA))
-- largest possible model #
T ' Safe_First yield lower bound of safe # range
T ' Safe_Last yield upper bound of safe # range

– Useful to determine the properties of the type (i.e. the computer being used)

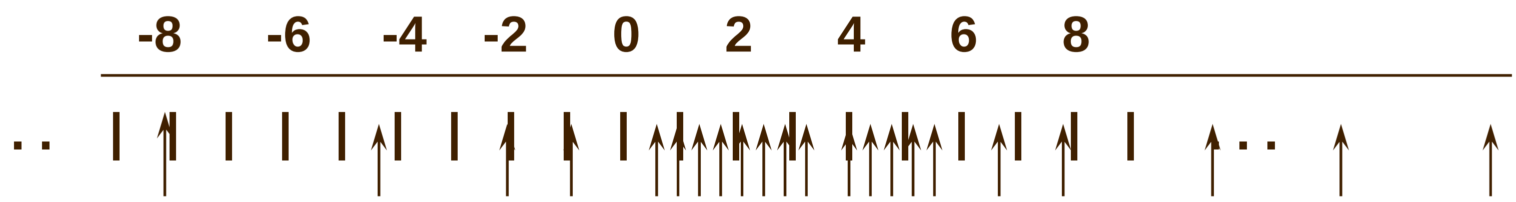
Defining Floating Point

- Specify number of significant digits

  type percentage is digits 4;

  type Gpa is digits 2 range 0.0 .. 4.0;

  type Mass is digits 6 range 0.0.. 1.0E35;



- model numbers not equally spaced
- let compiler worry about implementation

Predefined Floats

- Float
- Short_Float
- Long_Float
- Short_Short_Float *
- Long_Long_Float *

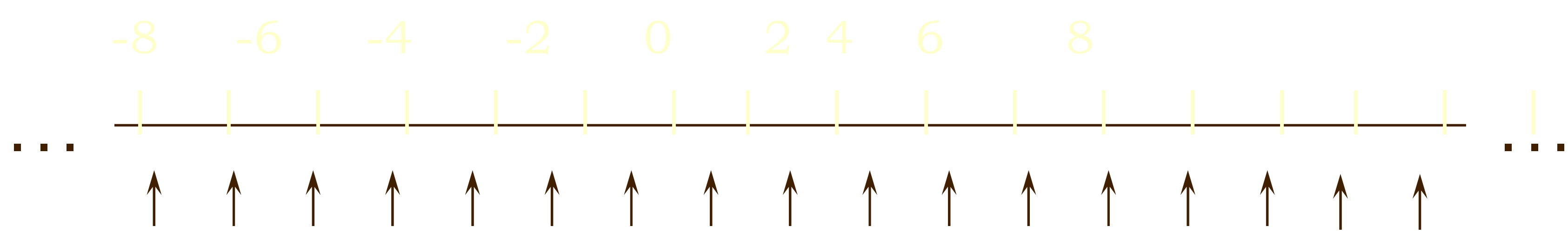
* Optional

Fixed Point Types

- an approximation of real numbers based on the user defined error bound (delta).
- accuracy is the precision of the number to the defined delta.
- Two types
  - Fixed Point (ordinary)
  - Decimal Fixed Point

Fixed Point

- Fixed point uses a fixed distance between consecutive values:



Fixed Point
(ordinary)

- ✓ Use for precise number calculations
- ✓ Multiplication results need to be converted to the type you desire
- ✓ Fixed point numbers can be VERY expensive in terms of conversion. Use with great care, especially in real-time applications.
- ✓ Delta can be any number

Fixed Point Type
(ordinary)

Set of Values:	set of approximations of real Numbers
Structure:	delta N range L.X..U.X
Set of Operations:	assignment :=
	membership in not in
	relation = /= < <= > >=
	math + - * /
	unary + - abs
	exponentiating **

Attributes for Fixed

Small	Delta	Fore	Aft	Address
Base	First	Image	Last	
Machine_Overflows		Machine_Radix		
Machine_Rounds		Max	Min	Pred
Succ	Range	Size	Small	
Valid	Wide_Image		Wide_Value	
Value	Wide_Width		Width	

Decimal Attributes

```
T ' DELTA                -- specified delta value
T ' MANTISSA
T ' SMALL                       -- smallest positive model #
T ' LARGE =  (2.0**T ' MANTISSA-1) * T ' SMALL
                                        -- largest model #
T ' FORE
T ' AFT
T ' FIRST
T ' LAST
```

Decimal Types

- Use for precise number calculations
- Multiplication results need to be converted to the type you desire
- Fixed point numbers can be VERY expensive in terms of conversion. Use with great care, especially in real-time applications.
- Delta must be a multiply of ten

Decimal Type

Set of Values: set of approximations of real Numbers

Structure: delta N digits X [range L..U]

Set of Operations:

- assignment :=
- membership in not in
- relation = /= < <= > >=
- math + - * /
- unary + - abs
- exponentiating **

Attributes for Decimals

Small Delta	Fore		Aft
Address	Base		Image
Digits	Round		Scale
Last	First	Machine_Overflows	
Machine_Radix	Machine_Rounds		
Max	Min	Pred	Succ
Range	Size	Small	Valid
Wide_Image	Wide_Value		
Value	Wide_Width		Width

Predefined Fixed Point and Decimal Types

- ✓ Duration
- ✓ Money (Annex F)

Currency Example

```
Procedure Main_Driver is
   type Currency is delta 0.01 digits  9 range
                                  0.0..1_000_000.0;
   My_Dollars,
   Your_Dollars : Currency := 0.0;
begin
   My_Dollars      := 100.53;
   Your_Dollars := My_Dollars;
   My_Dollars      := Your_Dollars * 5.0;
   Your_Dollars := -5.03;                    -- error
end Main_Driver;
```

Summary Real Types

- A representation of actual values (to some precision)
- a Primitive set of operations
- a set of attributes to enhance understandability

Summary of Scalar Types

- Two main class
  - Discrete
    - an ordered set of distinct values
  - Real
    - A representation of actual values (to some precision)
- a Primitive set of operations
- a set of attributes to enhance understandability

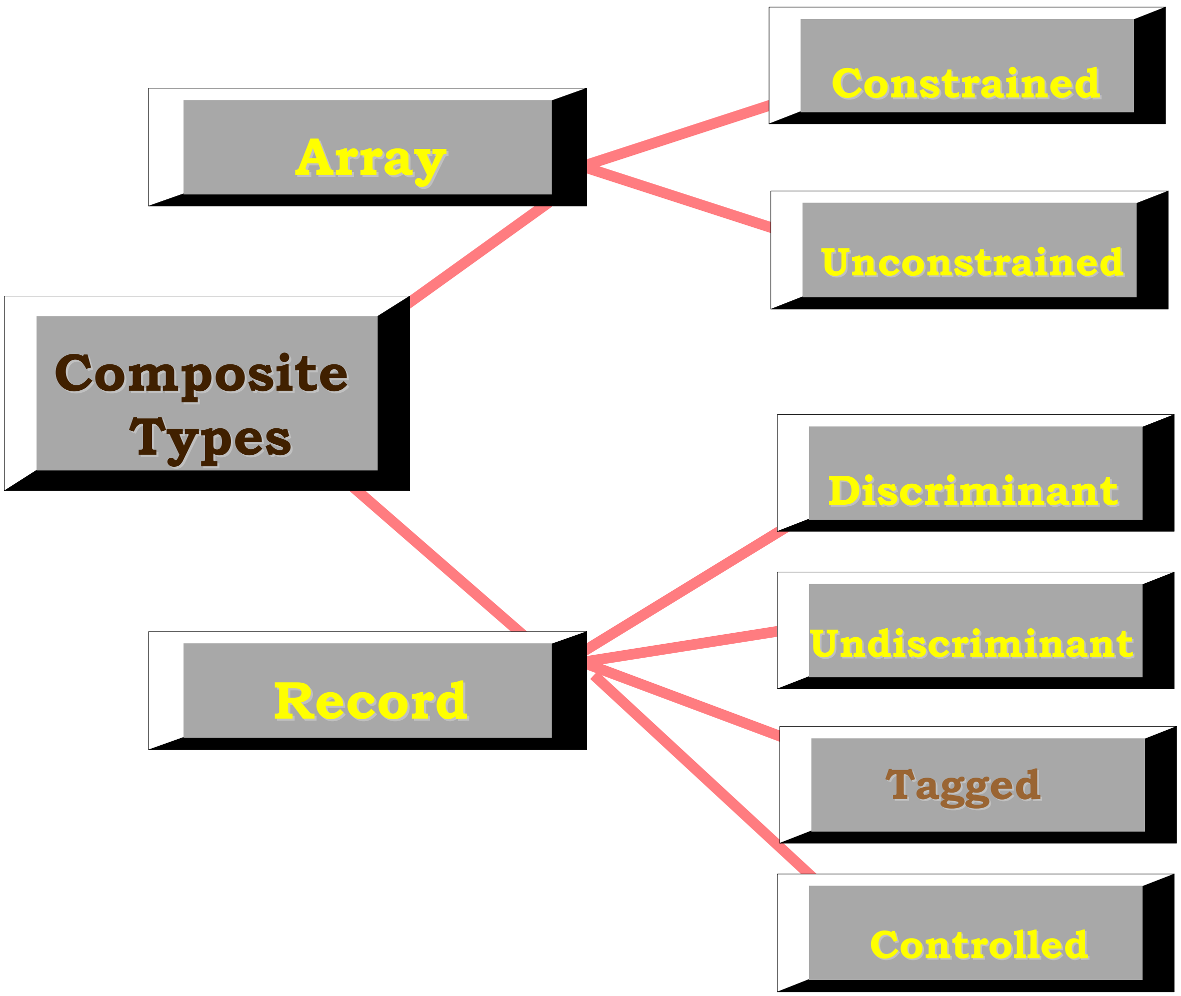
Summary of Scalar Operators

Operators	Identifiers	Integer	Real	Enumeration
Addition	+ -	X	X	
Explicit Conversion		X	X	X
Exponentiation	**	X	X	
Membership	In not in	X	X	X
Multiplication	*	X	X	
Division	/	X	X	
Qualification		X	X	X
Relational	= /= < ,= > >=	X	X	X
Unary	+ - abs	X	X	
Mod	mod	X		
Rem	Rem	X		

Summary of Types

- ✓ Types are use to define the world set you are programming to
- ✓ Typing helps to assist in error prevention and detection
- ✓ Typing enhances maintainability

Composite Types



Array Types

- ✓ Components have the same subtype (homogeneous)
- ✓ Components are referenced by a discreet (enumeration, boolean, or integer) index
- ✓ Kinds of arrays
  - Constrained
    - limits defined at type declaration
  - Unconstrained
    - limits defined at object declaration

STRING Literals

- ✓ DELIMITED BY QUOTATION MARKS
- ✓ ANY NUMBER OF CHARACTERS ALLOWED, INCLUDING NONE
- ✓ EXAMPLES:

```
"This is a message"
"first part of a STRING "&
"that continues on the next line"
""For Score ...""          -- YIELDS SINGLE QUOTES
""                         -- YIELDS A NULL STRING
```

Ada STRINGs

type STRING is a composite type

```
type STRING is array (NATURAL range <>) of character;
        -- predefined in package Ada
        STR_5 :  STRING (1 .. 5);
        STR_6 :  STRING (1 .. 6) := "Framus";
        WARNING :  constant STRING := "DANGER";
        subtype TEN_LONG is STRING (1 .. 10);
FIRST_TEN : TEN_LONG := "HEADER     ";
```

STR_6

F	r	a	m	u	s
1	2	3	4	5	6

WARNING

D	A	N	G	E	R
1	2	3	4	5	6

FIRST_TEN

H	E	A	D	E	R				
1	2	3	4	5	6	7	8	9	10

Concatenation (&)

```
STR_A : STRING (1 .. 4) := "Dear";
STR_B : STRING (1 .. 3) := "Ada";
STR_C : STRING (1 .. 10);
...
    STR_C := STR_A & " " & STR_B & ", " ;
```

STR_A

D	e	a	r
1	2	3	4

STR_B

A	d	a
1	2	3

STR_C

D	e	a	r		A	d	a	,	
1	2	3	4	5	6	7	8	9	10

STRINGs Example

```
procedure MAIN_DRIVER is
    NAME              : STRING (1..7);
    Other_Name        : String         := “Cookie”;
begin
   NAME := "JACKSON";
   NAME := "JACK";                  -- error
   NAME := "JACKSONS";              -- error
   NAME (1..4) := "JACK";           -- Called a slice
    Name := Other_Name & “ ”;
end MAIN_DRIVER;
```

More Array Examples

```
subtype Day_Type is integer range 1..31;

subtype Hours_type is integer range 0..24;

type Month_Type is (Jan, Feb, Mar, Apr, May,
                    Jun, Jul, Aug, Sep, Oct, Nov, Dec);

type Week_Type is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

type Year_Type is array (Month_Type) of Day_Type;

type Work_week is array (Week_Type range Mon..Fri) of
                                        Hours_type;
```

Array Objects

```
Year : Year_Type := (31 , 29, 31, 30, 31, 30, 31,
                              31, 30, 31, 30, 31);

--or--

Leap_Year : Year_Type := (Jun => 30, Feb => 29, Sep => 30,
   Apr => 30, Nov => 30, others => 31);


begin

…..

Year (Jul) := 31;
```

Multi-Dimensional Arrays

The number of dimensions of an array are determined by the number of discrete ranges given in the index constraint.

Components of a multi-dimensional array are referenced by an index value for each of the possible indexes in the order they appear in the type definition.

Record Types

- ✓ Defines a collection of types that are potentially different (heterogeneous)
- ✓ referenced internals by object name
- ✓ can contain any Ada type or subtype

Record Example

```
-- RECORD TYPE DECLARATION
type DAY_TYPE is range 1..31;
type MONTH_TYPE is (Jan,Feb,Mar,...Dec);
type YEAR_TYPE is range 0..2085;
type DATE_TYPE is
record
      DAY  :DAY_TYPE;
      MONTH:MONTH_TYPE;
      YEAR :YEAR_TYPE;
end record;
-- RECORD OBJECT DECLARATION
TODAY :DATE_TYPE;
```

Record Example (cont)

```
-- RECORD COMPONENT REFERENCE
TODAY.DAY := 15;
TODAY.MONTH := JUN;
TODAY.YEAR := 1990;
if TODAY.DAY = 16 and TODAY.MONTH = June then
        PUT_LINE("Yesterday was 15 June");
end if;

    type WEEK_TYPE is array(1..7) of DATE_TYPE;
    ARRAY_OBJECT : WEEK_TYPE;
    ...
    ARRAY_OBJECT(INDEX).COMPONENT
```

The way to reference a component in an array of records

Records - More Examples

RECORD OBJECT REFERENCE

```
TODAY := (15,JUN,1990);    -- an aggregate
                -- or --
if TODAY /= (6, DEC, 1942) then ...
                -- or --
TODAY := DATE_TYPE'(15,JUN,1990);
                -- or --
TODAY := (DAY   => 15,
          MONTH => JUN,
          YEAR  => 1990);
```

Default Record Component Values

- ✓ If a component of a record type has a default value, every object declared to be of the record type will have that initial value at declaration time.
- ✓ Can be specified for all or some components
- ✓ At record object elaboration, if no initial value is given, default values are used

Default Record Component Values

```
type DEFAULT_EXAMPLE is
record
    TOTAL :FLOAT :=0.0;
    STATE :STATE_CODE;
    VET   :BOOLEAN :=TRUE;
end record;
SAMPLE:  DEFAULT_EXAMPLE;
```

Records initializing

```
type FRACTION is
  record
    DIVIDEND : DND_TYPE    := 0;
    DIVISOR    : DIV_TYPE       := 1;
end record;

F  : FRACTION;        --  initial value of (0,1)
G : FRACTION  :=  (2,3);
```

Nested Records

- Components of records may be of any type, including other records
- The value of a nested record is a nested aggregate
- Component selection used extended 'dotted' notation

Nested Records Example

```
type TEMPERATURE_LOG is
record
      TEMP:INTEGER;
      DATE:DATE_TYPE;
end record;
LOG:TEMPERATURE_LOG;
...
LOG.TEMP :=50;
LOG.DATE.DAY :=15;
LOG.DATE.MONTH := JUN;
LOG.DATE.YEAR := 1990;
```

```
   -- or
LOG.DATE :=(15,JUN,1990);
-- or even
LOG:= (TEMP => 50,
          DATE => (15,JUN,1990));

-- or, using positional notation
LOG := (50, (15,JUN,1990));
```

Records

```
type DAYS is (MON, TUE, WED, THU, FRI, SAT, SUN);
type DAY_TYPE is range 1..31;
type MONTH_TYPE is (Jan,Feb,Mar,...Dec);
type YEAR_TYPE is range 0..2085;

type NEW_DATE_TYPE is
record
        DAY_OF_WEEK : DAYS;
        DAY : DAY_TYPE;
        MONTH : MONTH_TYPE;
        YEAR : YEAR_TYPE;
end record;
TODAY : NEW_DATE_TYPE;

begin
        TODAY.DAY_OF_WEEK := FRI;
        TODAY.DAY := 15;
        TODAY.MONTH   := JUN;
        TODAY.YEAR := 1990;
```

Records

- No discriminant
  - Components do not depend on a discriminant
- Discriminant
  - Components of a record depend on another component called a discriminant

Record Variant Parts

The actual existence of certain fields can depend on a discriminant value

```
type DRIVER is (GOOD,BAD);
type INSURANCE_RATE is range 1..5000;
type DISCOUNT is delta 0.001 range 0.0..1.0;
type INSURANCE (KIND:DRIVER) is
record
        NORMAL_RATE:INSURANCE_RATE;
        case KIND is
                when GOOD => DISCOUNT_RATE:DISCOUNT;
                when BAD =>   ADDITIONAL:INSURANCE_RATE;
                              ACCIDENT_DATE:  DATE_TYPE;  -- Good and Bad have different sizes
        end case;
end record;
```

More Examples

```
A_DRIVER : INSURANCE (GOOD);
ANOTHER : INSURANCE (BAD);

begin
    A_DRIVER.NORMAL_RATE        := 250;
    A_DRIVER.DISCOUNT_RATE      := 0.015;

    ANOTHER.NORMAL_RATE         := 250;
    ANOTHER.ADDITIONAL          := 1000;
```

Other Record Types

- Tagged
  - Used when you need to expand the data structure in different ways
- Controlled
  - Used when you want to do automatic set-up and clean-up of data structures

Record Summary

- ✓ NO DISCRIMINANT
- ✓ DISCRIMINANT
  - – components of the record depend on a component called a discriminant
- ✓ VARIANT
  - – record structures of the same type contain different components, based on discriminant

Record with a Variant part

```
with Ada.Text_Io, Ada.Strings.Bounded, Ada.Calendar;
 procedure Variantr is
   Max_Buffer :  constant Positive := 256;
package Message_Buffer is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max_Buffer);
   type Message_Classifications is (Unclass,Flash, Alpi);
     type Messages (Mesg_Class : Message_Classifications := Unclass) is
   record
     Buffer : Message_Buffer.Bounded_String;
     case Mesg_Class is
       when Unclass =>
         Save : Boolean := True;
       when Flash =>
         Time_Stamp : Ada.Calendar.Time;
            when Alpi =>
         Flag_Officer : Boolean := True;
end case;
  end record;

Buf : Message_Buffer.Bounded_String;
   TS : Ada.Calendar.Time;

M1 : Messages;
M2 : Messages ( Mesg_Class => Flash);

begin
  M1 := (Mesg_Class=>Alpi, Buffer => Buf, Flag_Officer => False);
  M1 := (Mesg_Class=>Unclass, Buffer => Buf, Save => False);
  M1.Save :=  False;

M2 := (Mesg_Class=>Unclass, Buffer => Buf, Save => False); --Discriminant
check will fail at run-time, exception will be raised

M1.Flag_Officer :=  False; --  CONSTRAINT_ERROR

null;
end VariantR;
```

Composite Types

- Arrays
  - Homogeneous collection
  - indexed by a scalar object
- Records
  - heterogeneous collection
  - referenced internals by object name

Exercises

Objective: Demonstrate the use of records, arrays, and loops in an Ada program.

Problem Write a program to count the number of vowels (A,E,I,O or U) in its input. Allow the user to type or read from a file, sequences of characters (as many as they like) terminate by a null string or end of file. Display the number of occurrences of each vowel as well as a grand total.

Source Code

```
--Filename ex1b.adb
with Ada.Text_Io;
--with Enumeration_Io;
procedure ex1b is
 type Counters is range 0..1_000_000;

type Vowels is (‘A’, ‘E’, ‘I’, ‘O’, ‘U’);

type Vowels_Count is
record
        Vowel : String := ‘A’;
        Count  : Counters := 0;
End record;

type Tables is Array (Vowels’First..Vowels’Last) of Vowels_Count;   -- Change the area index to ‘range

Table : Tables :=(
begin   --Main
        Ada.Text_Io.Put (" ");
end ex1b;
```

Access Types

(Pointers)

Access Types

- ✓ Designates an object by an allocator
- ✓ Only object in Ada that has a default value (set to null if not initialized)
- ✓ must be defined with a data structure or subprogram type to pointed at.

Access Types

Access types point to data structures, subprograms, or other access types

```
type Name_Type is string (1..10);
Type Name_Ptr_Type is access Name_Type;
type Int_Ptr is access Integer;
type Vehicle_Ptr is access all Vehicle'class;
type Procedure_Ptr is access procedure;  --points to any
                                         --parameter less procedure
IP : Int_Ptr;
I : aliased Integer;        --aliased allows I to be pointed to
IP := I'Access;
IP.all := 42;               --I must be de-referenced as all
IP := new Integer'(I);      --makes new values, copies I into it
```

Problem with Pointers

•It is often convenient to declare a pointer to a data object and use this pointer as a parameter.

•The problem: even if you make this parameter a *read only* parameter via the *in* mode, the function/procedure can change what the pointer points to (rather than the pointer itself).

•To prevent this, there is a mechanism that makes a pointer and what it points to *read only*.

Constant Access Types

Replace the word *all* in the type definition by the word *constant*.

type Int_Ptr is access constant Integer;

Now, variables of type Int_Ptr can point to Integers, but what they point to may not be modified.

```
IP : Int_Ptr;
I : aliased Integer; --aliased allows I to be pointed to

IP := I'Access;

IP := new Integer' (I);          --legal, not modifying what IP points to
IP := new Integer' (5);          --new value, original still not modified

IP.all := 5;                     --illegal, compiler error.  IP is read only
```

This allows you to declare a pointer type, pass it as a parameter, and prevent the procedure/function from modifying what the pointer points to.