

# Introduction

## About AdaCore

# The Company

- Founded in 1994
- Centered around helping developers build **safe, secure and reliable** software
- Headquartered in New York and Paris
  - Representatives in countries around the globe
- Roots in Open Source software movement
  - GNAT compiler is part of GNU Compiler Collection (GCC)

## About This Training

# Your Trainer

- Experience in software development
  - Languages
  - Methodology
- Experience teaching this class

## Goals of the training session

- What you should know by the end of the training
- Syllabus overview
  - The syllabus is a guide, but we might stray off of it
  - ...and that's OK: we're here to cover **your needs**

# Roundtable

- 5 minute exercise
  - Write down your answers to the following
  - Then share it with the room
- Experience in software development
  - Languages
  - Methodology
- Experience and interest with the syllabus
  - Current and upcoming projects
  - Curious for something?
- Your personal goals for this training
  - What do you want to have coming out of this?
- Anecdotes, stories... feel free to share!
  - Most interesting or funny bug you've encountered?
  - Your own programming interests?

# Course Presentation

- Slides
- Quizzes
- Labs
  - Hands-on practice
  - Recommended setup: latest GNAT Studio
  - Class reflection after some labs
- Demos
  - Depending on the context
- Daily schedule

# Styles

- *This* is a definition
- `this/is/a.path`
- code **is** highlighted
- **commands are emphasised --like-this**

## **Warning**

This is a warning

## **Note**

This is an important piece of info

## **Tip**

This is a tip

# Overview

## A Little History

# The Name

- First called DoD-1
- Augusta Ada Byron, "first programmer"
  - Lord Byron's daughter
  - Planned to calculate **Bernoulli's numbers**
  - **First** computer program
  - On **Babbage's Analytical Engine**
- International Standards Organization standard
  - Updated about every 10 years
- Writing **ADA** is like writing **CPLUSPLUS**

# Ada Evolution Highlights

**Ada 83** Abstract Data Types  
Modules  
Concurrency  
Generics  
Exceptions

**Ada 95** OOP  
Child Packages  
Annexes

**Ada 2005** Multiple Inheritance  
Containers  
Ravenscar

**Ada 2012** Contracts  
Iterators  
Flexible Expressions

**Ada 2022** 'Image for all types  
Declare expression

## Note

Ada was created to be a **compiled, multi-paradigm** language with a **static** and **strong** type model

# Big Picture

# Language Structure (Ada95 and Onward)

- **Required** *Core* implementation
  - Always present in each compiler/run-time
    - Basic language contents (types, subprograms, packages, etc.)
    - Interface to Other Languages
- **Optional** *Specialized Needs Annexes*
  - No additional syntax
  - May be present or not depending on compiler/run-time
    - Real-Time Systems
    - Distributed Systems
    - Numerics
    - High-Integrity Systems

# Core Language Content (1/3)

- Types
  - Language-defined types, including string
  - User-defined types
  - Static types keep things consistent
  - Strong types enforce constraints
- Subprograms
  - Syntax differs between *values* and *actions*
  - **function** for *value* and **procedure** for *action*
  - Overloading of names allowed
- Dynamic memory management
  - **access type** for abstraction of pointers
  - Access to static memory, allocated objects, subprograms
  - Accesses are **checked** (unless otherwise requested)
- Packages
  - Grouping of related entities
  - Separation of concerns
  - Information hiding

## Core Language Content (2/3)

- Exceptions
  - Dealing with **errors**, **unexpected** events
  - Separate error-handling code from logic
- Generic Units
  - Code templates
  - Extensive parameterization for customization
- Object-Oriented Programming
  - Inheritance
  - Run-time polymorphism
  - Dynamic **dispatching**
- Contract-Based Programming
  - Pre- and post-conditions on subprograms
    - Formalizes specifications
  - Type invariants and predicates
    - Complex contracts on type definitions

## Core Language Content (3/3)

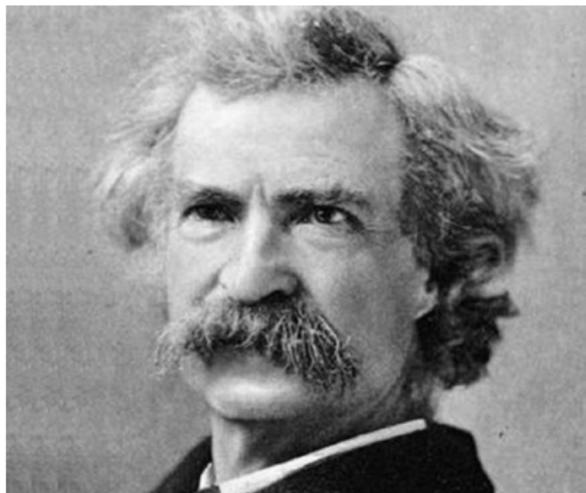
- Language-Based Concurrency
  - Explicit interactions
  - Run-time handling
  - Portable
- Low Level Programming
  - Define representation of types
  - Storage pools definition
  - Foreign language integration

# Language Examination Summary

- Three main goals
  - **Reliability**, maintainability
  - Programming as a **human** activity
  - Efficiency
- Easy-to-use
  - ...and hard to misuse
  - Very **few pitfalls** and exceptions

## So Why Isn't Ada Used Everywhere?

- "... in all matters of opinion our adversaries are insane"
  - *Mark Twain*



# Setup

# Canonical First Program

```
1 with Ada.Text_IO;  
2 -- Everyone's first program  
3 procedure Say_Hello is  
4 begin  
5   Ada.Text_IO.Put_Line ("Hello, World!");  
6 end Say_Hello;
```

- Line 1 - **with** - Package dependency
- Line 2 - **--** - Comment
- Line 3 - Say\_Hello - Subprogram name
- Line 4 - **begin** - Begin executable code
- Line 5 - Ada.Text\_IO.Put\_Line () - Subprogram call
- (cont) - "Hello, World!" - String literal (type-checked)

## "Hello World" Lab - Command Line

- Use an editor to enter the program shown on the previous slide
  - Use your favorite editor or just gedit/notepad/etc.
- Save and name the file `say_hello.adb` exactly
  - In a command prompt shell, go to where the new file is located and issue the following command:
    - `gprbuild say_hello`
- In the same shell, invoke the resulting executable:
  - `say_hello` (Windows)
  - `./say_hello` (Linux/Unix)

# "Hello World" Lab - GNAT STUDIO

- Start GNAT STUDIO from the command-line (`gnatstudio`) or Start Menu
- Create new project
  - Select `Simple Ada Project` and click `Next`
  - Fill in a location to to deploy the project
  - Set **main name** to `say_hello` and click `Apply`
- Expand the **src** level in the Project View and double-click `say_hello.adb`
  - Replace the code in the file with the program shown on the previous slide
- Execute the program by selecting `Build` → `Project` → `Build & Run` → `say_hello.adb`
  - Shortcut is the ► in the icons bar
- Result should appear in the bottom pane labeled *Run*:  
`say_hello.exe`

## Note on GNAT File Naming Conventions

- GNAT compiler assumes one compilable entity per file
  - Package specification, subprogram body, etc
  - So the body for `say_hello` should be the only thing in the file
- Filenames should match the name of the compilable entity
  - Replacing "." with "-"
  - File extension is ".ads" for specifications and ".adb" for bodies
  - So the body for `say_hello` will be in `say_hello.adb`
    - If there was a specification for the subprogram, it would be in `say_hello.ads`
- This is the **default** behavior. There are ways around both of these rules
  - For further information, see Section 3.3 *File Naming Topics and Utilities* in the **GNAT User's Guide**

# Declarations

## Introduction

# Ada Type Model

- Each *object* is associated a *type*
- **Static** Typing
  - Object type **cannot change**
  - ... but run-time polymorphism available (OOP)
- **Strong** Typing
  - **Compiler-enforced** operations and values
  - **Explicit** conversions for "related" types
  - **Unchecked** conversions possible
- Predefined types
- Application-specific types
  - User-defined
  - Checked at compilation and run-time

# Declarations

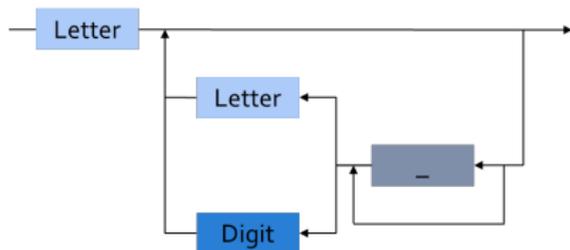
- **Declaration** associates a **name** to an **entity**
  - Objects
  - Types
  - Subprograms
  - et cetera
- In a **declarative part**
- Example: N : **Type** := Value;
  - N is usually an **identifier**
- **Some** implicit declarations
  - **Standard** types and operations
  - **Implementation**-defined

 **Warning**

Declaration **must precede** use

## Identifiers and Comments

# Identifiers



- Legal identifiers

Phase2

A

Space\_Person

- Not legal identifiers

Phase2\_\_1

A\_

\_space\_person

**Warning**

Reserved words are **forbidden**

- Character set **Unicode 4.0**

- Case **not significant**

- **SpacePerson**  $\iff$  **SPACEPERSON**

- ...but **different** from **Space\_Person**

# Reserved Words

<code>abort</code>	<code>else</code>	<code>null</code>	<code>reverse</code>
<code>abs</code>	<code>elsif</code>	<code>of</code>	<code>select</code>
<code>abstract</code> (95)	<code>end</code>	<code>or</code>	<code>separate</code>
<code>accept</code>	<code>entry</code>	<code>others</code>	<code>some</code> (2012)
<code>access</code>	<code>exception</code>	<code>out</code>	<code>subtype</code>
<code>aliased</code> (95)	<code>exit</code>	<code>overriding</code> (2005)	<code>synchronized</code> (2005)
<code>all</code>	<code>for</code>	<code>package</code>	<code>tagged</code> (95)
<code>and</code>	<code>function</code>	<code>parallel</code> (2022)	<code>task</code>
<code>array</code>	<code>generic</code>	<code>pragma</code>	<code>terminate</code>
<code>at</code>	<code>goto</code>	<code>private</code>	<code>then</code>
<code>begin</code>	<code>if</code>	<code>procedure</code>	<code>type</code>
<code>body</code>	<code>in</code>	<code>protected</code> (95)	<code>until</code> (95)
<code>case</code>	<code>interface</code> (2005)	<code>raise</code>	<code>use</code>
<code>constant</code>	<code>is</code>	<code>range</code>	<code>when</code>
<code>declare</code>	<code>limited</code>	<code>record</code>	<code>while</code>
<code>delay</code>	<code>loop</code>	<code>rem</code>	<code>with</code>
<code>delta</code>	<code>mod</code>	<code>renames</code>	<code>xor</code>
<code>digits</code>	<code>new</code>	<code>requeue</code> (95)	
<code>do</code>	<code>not</code>	<code>return</code>	

# Comments

- Terminate at end of line (i.e., no comment terminator sequence)

```
-- This is a multi-
```

```
-- line comment
```

```
A : B; -- this is an end-of-line comment
```

## Declaring Constants / Variables (simplified)

- An *expression* is a piece of Ada code that returns a **value**.

```
<identifier> : constant := <expression>;
```

```
<identifier> : <type> := <expression>;
```

```
<identifier> : constant <type> := <expression>;
```

# Quiz

Which statement(s) is (are) legal?

- A. `Function : constant := 1;`
- B. `Fun_ction : constant := 1;`
- C. `Fun_ction : constant := --initial value-- 1;`
- D. `Integer Fun_ction;`

# Quiz

Which statement(s) is (are) legal?

- A. `Function : constant := 1;`
- B. `Fun_ction : constant := 1;`
- C. `Fun_ction : constant := --initial value-- 1;`
- D. `Integer Fun_ction;`

Explanations

- A. `function` is a reserved word
- B. Correct
- C. Cannot have inline comments
- D. C-style declaration not allowed

## Literals

# String Literals

- A *literal* is a *textual* representation of a value in the code

```
A_Null_String : constant String := "";  
    -- two double quotes with nothing inside  
String_Of_Length_One : constant String := "A";  
Embedded_Single_Quotes : constant String  
    := "Embedded 'single' quotes";  
Embedded_Double_Quotes : constant String  
    := "Embedded ""double"" quotes";
```

# Decimal Numeric Literals

## ■ Syntax

```
decimal_literal ::=  
    numeral [.numeral] E [+numeral|-numeral]  
numeral ::= digit {'_' digit}
```

### Tip

Underscore is **not** significant and helpful for grouping

## ■ E (exponent) must always be integer

## ■ Examples

12	0	1E6	123_456
12.0	0.0	3.14159_26	2.3E-4

## Based Numeric Literals

```
based_literal ::= base # numeral [.numeral] # exponent  
numeral ::= base_digit { '_' base_digit }
```

- Base can be 2 .. 16
- Exponent is always a base 10 integer

```
16#FFF#           => 4095
```

```
2#1111_1111_1111# => 4095 -- With underline
```

```
16#F.FF#E+2      => 4095.0
```

```
8#10#E+3         => 4096 (8 * 8**3)
```

## Comparison to C's Based Literals

- Design in reaction to C issues
- C has **limited** bases support
  - Bases 8, 10, 16
  - No base 2 in standard
- Zero-prefixed octal `0nnn`
  - **Hard** to read
  - **Error-prone**

# Quiz

Which statement(s) is (are) legal?

- A. `I : constant := 0_1_2_3_4;`
- B. `F : constant := 12.;`
- C. `I : constant := 8#77#E+1.0;`
- D. `F : constant := 2#1111;`

# Quiz

Which statement(s) is (are) legal?

- A. `I : constant := 0_1_2_3_4;`
- B. `F : constant := 12.;`
- C. `I : constant := 8#77#E+1.0;`
- D. `F : constant := 2#1111;`

## Explanations

- A. Underscores are not significant - they can be anywhere (except first and last character, or next to another underscore)
- B. Must have digits on both sides of decimal
- C. Exponents must be integers
- D. Missing closing #

## Object Declarations

# Object Declarations

- An object is either `variable` or `constant`

- Basic Syntax

```
<name> : <subtype> [:= <initial value>];  
<name> : constant <subtype> := <initial value>;
```

- Constant should have a value
  - Except for privacy (seen later)

- Examples

```
Z, Phase : Analog;  
Max : constant Integer := 200;  
-- variable with a constraint  
Count : Integer range 0 .. Max := 0;  
-- dynamic initial value via function call  
Root : Tree := F(X);
```

## Multiple Object Declarations

- Allowed for convenience

```
A, B : Integer := Next_Available (X);
```

- Identical to series of single declarations

```
A : Integer := Next_Available (X);
```

```
B : Integer := Next_Available (X);
```

### Warning

May get different value!

```
T1, T2 : Time := Current_Time;
```

# Predefined Declarations

- **Implicit** declarations
- Language standard
- Annex A for *Core*
  - Package Standard
  - Standard types and operators
    - Numerical
    - Characters
  - About **half the RM** in size
- "Specialized Needs Annexes" for *optional*
- Also, implementation specific extensions

# Implicit Vs Explicit Declarations

- **Explicit** → in the source

```
type Counter is range 0 .. 1000;
```

- **Implicit** → **automatically** by the compiler

```
function "+" (Left, Right : Counter) return Counter;  
function "-" (Left, Right : Counter) return Counter;  
function "*" (Left, Right : Counter) return Counter;  
function "/" (Left, Right : Counter) return Counter;  
...
```

- Compiler creates appropriate operators based on the underlying type
  - Numeric types get standard math operators
  - Array types get concatenation operator
  - Most types get assignment operator

# Elaboration

- *Elaboration* has several facets:
  - **Initial value** calculation
    - Evaluation of the expression
    - Done at **run-time** (unless static)
  - Object creation
    - Memory **allocation**
    - Initial value assignment (and type checks)
- Runs in linear order
  - Follows the program text
  - Top to bottom

**declare**

```
First_One : Integer := 10;
```

```
Next_One  : Integer := First_One;
```

```
Another_One : Integer := Next_One;
```

**begin**

```
...
```

# Quiz

Which block(s) is (are) legal?

- A. `A, B, C : Integer;`
- B. `Integer : Standard.Integer;`
- C. `Null : Integer := 0;`
- D. `A : Integer := 123;`  
`B : Integer := A * 3;`

## Quiz

Which block(s) is (are) legal?

- A. `A, B, C : Integer;`
- B. `Integer : Standard.Integer;`
- C. `Null : Integer := 0;`
- D. `A : Integer := 123;`  
`B : Integer := A * 3;`

## Explanations

- A. Multiple objects can be created in one statement
- B. `Integer` is *predefined* so it can be overridden
- C. `null` is *reserved* so it can **not** be overridden
- D. Elaboration happens in order, so B will be 369

## Universal Types

# Universal Types

- Implicitly defined
- Entire *classes* of numeric types
  - `universal_integer`
  - `universal_real`
  - `universal_fixed` (not seen here)
- Match any integer / real type respectively
  - **Implicit** conversion, as needed

```
X : Integer64 := 2;  
Y : Integer8  := 2;  
F : Float    := 2.0;  
D : Long_Float := 2.0;
```

## Numeric Literals Are Universally Typed

- No need to type them
  - e.g 0UL as in C
- Compiler handles typing

** Note**

No bugs with precision

```
X : Unsigned_Long := 0;  
Y : Unsigned_Short := 0;
```

## Literals Must Match "Class" of Context

- **universal\_integer** literals → **Integer**
- **universal\_real** literals → **fixed** or **floating** point
- Legal

```
X : Integer := 2;
```

```
Y : Float := 2.0;
```

- Not legal

```
X : Integer := 2.0;
```

```
Y : Float := 2;
```

## Named Numbers

# Named Numbers

- Associate a **name** with an **expression**
  - Used as **constant**
  - **universal\_integer**, or **universal\_real**
  - Compatible with integer / real respectively
  - Expression must be **static**

- Syntax

```
<name> : constant := <static_expression>;
```

- Example

```
Pi : constant := 3.141592654;  
One_Third : constant := 1.0 / 3.0;
```

## A Sample Collection of Named Numbers

```
package Physical_Constants is
  Polar_Radius : constant := 20_856_010.51;
  Equatorial_Radius : constant := 20_926_469.20;
  Earth_Diameter : constant :=
    2.0 * ((Polar_Radius + Equatorial_Radius)/2.0);
  Gravity : constant := 32.1740_4855_6430_4;
  Sea_Level_Air_Density : constant :=
    0.002378;
  Altitude_Of_Tropopause : constant := 36089.0;
  Tropopause_Temperature : constant := -56.5;
end Physical_Constants;
```

# Named Number Benefit

- Evaluation at **compile time**
  - As if **used directly** in the code

 **Tip**

Useful due to their **perfect** accuracy

```
Named_Number      : constant :=      1.0 / 3.0;
```

```
Typed_Constant   : constant Float := 1.0 / 3.0;
```

Object	Named_Number	Typed_Constant
F32 : Float_32;	3.33333E-01	3.33333E-01
F64 : Float_64;	3.333333333333333E-01	3.333333_43267441E-01
F128 : Float_128;	3.3333333333333333E-01	3.333333_43267440796E-01

## Scope and Visibility

# Scope and Visibility

- **Scope** of a name
  - Where the name is **potentially** available
  - Determines **lifetime**
  - Scopes can be **nested**
- **Visibility** of a name
  - Where the name is **actually** available
  - Defined by **visibility rules**
  - **Hidden** → *in scope* but not **directly** visible

# Introducing Block Statements

- **Sequence** of statements

- Optional *declarative part*
- Can be **nested**
- Declarations **can hide** outer variables

- **Syntax**

```
[<block-name> :] declare
    <declarative part>
begin
    <statements>
end [block-name];
```

- **Example**

```
Swap: declare
    Temp : Integer;
begin
    Temp := U;
    U := V;
    V := Temp;
end Swap;
```

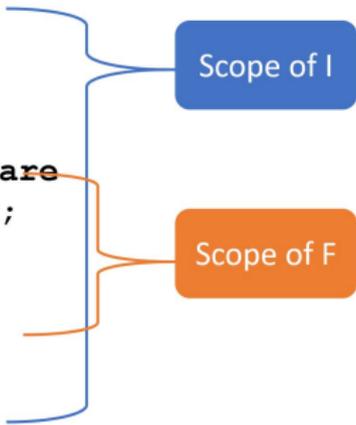
# Scope and "Lifetime"

- Object in scope → exists

## Note

No *scoping* keywords (C's **static**, **auto** etc...)

```
Outer : declare
  I : Integer;
begin
  I := 1;
  Inner : declare
    F : Float;
  begin
    F := 1.0;
  end Inner;
  I := I + 1;
end Outer;
```



Scope of I

Scope of F

# Name Hiding

- Caused by **homographs**
  - **Identical** name
  - **Different** entity

```
declare
  M : Integer;
begin
  M := 123;
  declare
    M : Float;
  begin
    M := 12.34; -- OK
    M := 0;    -- compile error: M is a Float
  end;
  M := 0.0; -- compile error: M is an Integer
  M := 0;   -- OK
end;
```

# Overcoming Hiding

- Add a **prefix**
  - Needs named scope

## ⚠ Warning

- Homographs are a *code smell*
  - May need **refactoring**...

```
Outer : declare
  M : Integer;
begin
  M := 123;
  declare
    M : Float;
  begin
    M := 12.34;
    Outer.M := Integer (M);  -- reference "hidden" Integer M
  end;
end Outer;
```

# Quiz

What output does the following code produce? (Assume Print prints the current value of its argument)

```
1 declare
2   M : Integer := 1;
3 begin
4   M := M + 1;
5   declare
6     M : Integer := 2;
7   begin
8     M := M + 2;
9     Print (M);
10  end;
11  Print (M);
12 end;
```

A. 2, 2

B. 2, 4

C. 4, 4

D. 4, 2

# Quiz

What output does the following code produce? (Assume Print prints the current value of its argument)

```
1 declare
2   M : Integer := 1;
3 begin
4   M := M + 1;
5   declare
6     M : Integer := 2;
7   begin
8     M := M + 2;
9     Print (M);
10  end;
11  Print (M);
12 end;
```

- A. 2, 2
- B. 2, 4
- C. 4, 4
- D. 4, 2

## Explanation

- Inner M gets printed first. It is initialized to 2 and incremented by 2
- Outer M gets printed second. It is initialized to 1 and incremented by 1

## Aspects

# Pragmas

- Originated as a compiler directive for things like

- Specifying the type of optimization

```
pragma Optimize (Space);
```

- Inlining of code

```
pragma Inline (Some_Procedure);
```

- Properties (`aspects`) of an entity

- Appearance in code

- Unrecognized pragmas

```
pragma My_Own_Pragma;
```

- No effect
- Cause **warning** (standard mode)

- Must follow correct syntax

```
pragma Page;           -- parameterless  
pragma Optimize (Off); -- with parameter
```

**⚠ Warning**

Malformed pragmas are **illegal**

```
pragma Illegal One;    -- compile error
```

# Aspect Clauses

- Define **additional** properties of an entity
  - Representation (eg. `with Pack`)
  - Operations (eg. `Inline`)
  - Can be **standard** or **implementation**-defined
- Usage close to pragmas
  - More **explicit, typed**
  - **Recommended** over pragmas
- Syntax

```
with aspect_mark [ => expression]  
    {, aspect_mark [ => expression] }
```

## Note

Aspect clauses always part of a **declaration**

## Aspect Clause Example: Objects

### ■ Updated **object syntax**

```
<name> : <subtype_indication> [:= <initial value>]  
      with aspect_mark [ => expression]  
      {, aspect_mark [ => expression] };
```

### ■ Usage

```
-- using aspects
```

```
CR1 : Control_Register with  
    Size      => 8,  
    Address => To_Address (16#DEAD_BEEF#);
```

```
-- using representation clauses
```

```
CR2 : Control_Register;  
for CR2'Size use 8;  
for CR2'Address use To_Address (16#DEAD_BEEF#);
```

## Boolean Aspect Clauses

- **Boolean** aspects only

- Longhand

```
procedure Foo with Inline => True;
```

- Aspect name only → **True**

```
procedure Foo with Inline; -- Inline is True
```

- No aspect → **False**

```
procedure Foo; -- Inline is False
```

- Original form!

## Summary

# Summary

- Declarations of a **single** type, permanently
  - OOP adds flexibility
- Named-numbers
  - **Infinite** precision, **implicit** conversion
- **Elaboration** concept
  - Value and memory initialization at **run-time**
- Simple **scope** and **visibility** rules
  - **Prefixing** solves **hiding** problems
- Pragmas, Aspects
- Detailed syntax definition in Annex P (using BNF)

# Basic Types

# Introduction

# Strong Typing

- Definition of *type*
  - Applicable **values**
  - Applicable *primitive* **operations**
- Compiler-enforced
  - **Check** of values and operations
  - Easy for a computer

 **Tip**

Developer can focus on **earlier** phase: requirement

## Strongly-Typed Vs Weakly-Typed Languages

- Weakly-typed:
  - Conversions are **unchecked**
  - Type errors are easy

```
typedef enum {north, south, east, west} direction;  
typedef enum {sun, mon, tue, wed, thu, fri, sat} days;  
direction heading = north;
```

```
heading = 1 + 3 * south/sun; // what?
```

- Strongly-typed:
  - Conversions are **checked**
  - Type errors are hard

```
type Directions is (North, South, East, West);  
type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
Heading : Directions := North;
```

```
...
```

```
Heading := 1 + 3 * South/Sun; -- Compile Error
```

## A Little Terminology

- **Declaration** creates a **type name**

```
type <name> is <type definition>;
```

- **Type-definition** defines its structure

- Characteristics, and operations
- Base "class" of the type

```
type Type_1 is digits 12; -- floating-point  
type Type_2 is range -200 .. 200; -- signed integer  
type Type_3 is mod 256; -- unsigned integer
```

- **Representation** is the memory-layout of an **object** of the type

# Abstract Data Types (ADT)

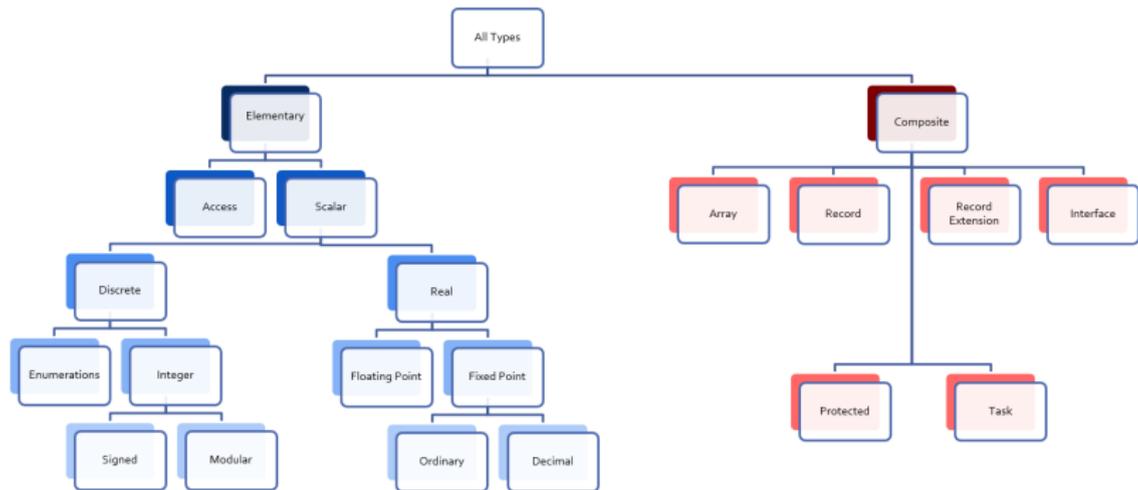
- **Variables** of the **type** encapsulate the **state**
- Classic definition of an ADT
  - Set of **values**
  - Set of **operations**
  - **Hidden** compile-time **representation**
- Compiler-enforced
  - Check of values and operation
  - Easy for a computer
  - Developer can focus on **earlier** phase: requirements

## Ada "Named Typing"

- **Name** differentiate types
- Structure does **not**
- Identical structures may **not** be interoperable

```
type Yen is range 0 .. 100_000_000;  
type Ruble is range 0 .. 100_000_000;  
Mine : Yen;  
Yours : Ruble;  
...  
Mine := Yours; -- not legal
```

# Categories of Types



# Scalar Types

- Indivisible: No **components** (also known as *fields* or *elements*)
- **Relational** operators defined (<, =, ...)
  - **Ordered**
- Have common **attributes**
- **Discrete** Types
  - Integer
  - Enumeration
- **Real** Types
  - Floating-point
  - Fixed-point

# Discrete Types

- **Individual** ("discrete") values
  - 1, 2, 3, 4 ...
  - Red, Yellow, Green
- Integer types
  - Signed integer types
  - Modular integer types
    - Unsigned
    - **Wrap-around** semantics
    - Bitwise operations
- Enumeration types
  - Ordered list of **logical** values

# Attributes

- Properties of entities that can be queried like a function
  - May take input parameters
- Defined by the language and/or compiler
  - Language-defined attributes found in RM K.2
  - *May* be implementation-defined
    - GNAT-defined attributes found in GNAT Reference Manual
  - Cannot be user-defined
- Attribute behavior is generally pre-defined
  - `Type_T'Digits` gives number of digits used in `Type_T` definition
- Some attributes can be modified by coding behavior
  - `Typemark'Size` gives the size of `Typemark`
    - Determined by compiler **OR** by using a representation clause
  - `Object'Image` gives a string representation of `Object`
    - Default behavior which can be replaced by aspect `Put_Image`
- Examples

```
J := Object'Size;  
K := Array_Object'First(2);
```

## Type Model Run-Time Costs

- Checks at compilation **and** run-time
- **Same performance** for identical programs
  - Run-time type checks can be disabled

### Note

Compile-time check is *free*

**C**

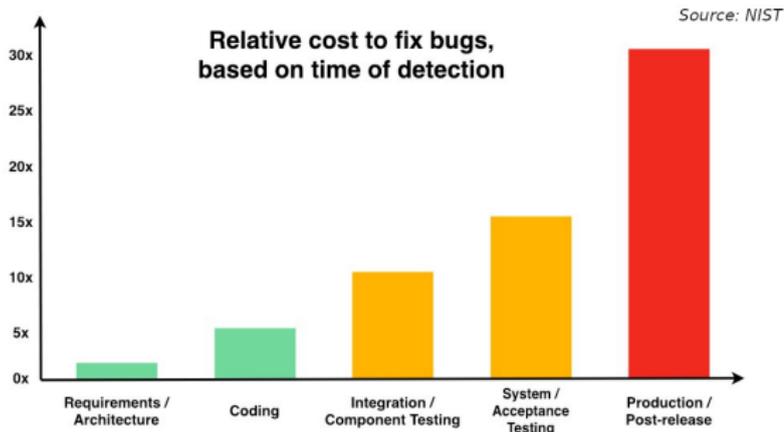
```
int X;  
int Y; // range 1 .. 10  
...  
if (X > 0 && X < 11)  
    Y = X;  
else  
    // signal a failure
```

**Ada**

```
X : Integer;  
Y, Z : Integer range 1 .. 10;  
...  
Y := X;  
Z := Y; -- no check required
```

# The Type Model Saves Money

- Shifts fixes and costs to **early phases**
- Cost of an error *during a flight?*



## Discrete Numeric Types

# Signed Integer Types

- Range of signed **whole** numbers
  - Symmetric about zero ( $-0 = +0$ )
- Syntax

```
type <identifier> is range <lower> .. <upper>;
```

- Implicit numeric operators

```
-- 12-bit device
```

```
type Analog_Conversions is range 0 .. 4095;
```

```
Count : Analog_Conversions := 0;
```

```
...
```

```
begin
```

```
...
```

```
Count := Count + 1;
```

```
...
```

```
end;
```

# Signed Integer Bounds

- Must be **static**
  - Compiler selects **base type**
  - Hardware-supported integer type
  - Compilation **error** if not possible

## Predefined Signed Integer Types

- `Integer`  $\geq$  **16 bits** wide
- Other **probably** available
  - `Long_Integer`, `Short_Integer`, etc.
  - Guaranteed ranges: `Short_Integer`  $\leq$  `Integer`  $\leq$  `Long_Integer`
  - Ranges are all **implementation-defined**

### **Warning**

Portability not guaranteed

- But usage may be difficult to avoid

## Operators for Signed Integer Type

- By increasing precedence

relational operator = | /= | < | <= | > | >=

binary adding operator + | -

unary adding operator + | -

multiplying operator \* | / | **mod** | **rem**

highest precedence operator \*\* | **abs**

### Note

Exponentiation (\*\*) result will be a signed integer

- So power **must** be **Integer** >= 0

### Warning

Division by zero → `Constraint_Error`

## Signed Integer Overflows

- Finite binary representation
- Common source of bugs

```
K : Short_Integer := Short_Integer'Last;
```

```
...
```

```
K := K + 1;
```

```
2#0111_1111_1111_1111# = (2**15)-1
```

```
+                1
```

```
=====
```

```
2#1000_0000_0000_0000# = -32,768
```

# Signed Integer Overflow: Ada Vs Others

- Ada
  - `Constraint_Error` standard exception
  - Incorrect numerical analysis
- Java
  - Silently **wraps** around (as the hardware does)
- C/C++
  - **Undefined** behavior (typically silent wrap-around)

## String Attributes for All Scalars

- T'Image (input)
  - Converts  $T \rightarrow \text{String}$
- T'Value (input)
  - Converts  $\text{String} \rightarrow T$

```
Number : Integer := 12345;  
Input   : String (1 .. N);  
...  
Put_Line (Integer'Image (Number));  
...  
Get (Input);  
Number := Integer'Value (Input);
```

## Range Attributes for All Scalars

- T'First
  - First (**smallest**) value of type T
- T'Last
  - Last (**greatest**) value of type T
- T'Range
  - Shorthand for T'First .. T'Last

```
type Signed_T is range -99 .. 100;  
Smallest : Signed_T := Signed_T'First;  -- -99  
Largest  : Signed_T := Signed_T'Last;   -- 100
```

## Neighbor Attributes for All Scalars

- T'Pred (Input)
  - Predecessor of specified value
  - Input type must be T
- T'Succ (Input)
  - Successor of specified value
  - Input type must be T

```
type Signed_T is range -128 .. 127;
type Unsigned_T is mod 256;
Signed    : Signed_T := -1;
Unsigned  : Unsigned_T := 0;
...
Signed := Signed_T'Succ (Signed); -- Signed = 0
...
Unsigned := Unsigned_T'Pred (Unsigned); -- Signed = 255
```

## Min/Max Attributes for All Scalars

- `T'Min (Value_A, Value_B)`
  - **Lesser** of two T
- `T'Max (Value_A, Value_B)`
  - **Greater** of two T

```
Safe_Lower : constant := 10;  
Safe_Upper : constant := 30;  
C : Integer := 15;  
...  
C := Integer'Max (Safe_Lower, C - 1);  
...  
C := Integer'Min (Safe_Upper, C + 1);
```

# Quiz

What happens when you try to compile/run this code?

```
C1 : constant := 2 ** 1024;  
C2 : constant := 2 ** 1024 + 10;  
C3 : constant := C1 - C2;  
V  : Integer := C1 - C2;
```

- A. Compile error
- B. Run-time error
- C. V is assigned to -10
- D. Unknown - depends on the compiler

# Quiz

What happens when you try to compile/run this code?

```
C1 : constant := 2 ** 1024;  
C2 : constant := 2 ** 1024 + 10;  
C3 : constant := C1 - C2;  
V  : Integer := C1 - C2;
```

- A. Compile error
- B. Run-time error
- C. *V is assigned to -10*
- D. Unknown - depends on the compiler

## Explanations

- $2^{1024}$  too big for most runtimes BUT
- C1, C2, and C3 are named numbers, not typed constants
  - Compiler uses unbounded precision for named numbers
  - Large intermediate representation does not get stored in object code
- For assignment to V, subtraction is computed by compiler
  - V is assigned the value -10

## Modular Types

## Bit Pattern Values and Range Constraints

- Binary based assignments possible
- No `Constraint_Error` when in range
- **Even if** they would be  $\leq 0$  as a **signed** integer type

```
procedure Demo is
  type Byte is mod 256;  -- 0 .. 255
  B : Byte;
begin
  B := 2#1000_0000#;  -- not a negative value
end Demo;
```

## Modular Range Must Be Respected

```
procedure P_Unsigned is
  type Byte is mod 2**8;  -- 0 .. 255
  B : Byte;
  type Signed_Byte is range -128 .. 127;
  SB : Signed_Byte;
begin
  ...
  B := -256;           -- compile error
  SB := -1;
  B := Byte (SB);    -- run-time error
  ...
end P_Unsigned;
```

## Safely Converting Signed to Unsigned

- Conversion may raise `Constraint_Error`
- Use `T'Mod` to return argument `mod` `T'Modulus`
  - `Universal_Integer` argument
  - So **any** integer type allowed

```
procedure Test is
  type Byte is mod 2**8;  -- 0 .. 255
  B : Byte;
  type Signed_Byte is range -128 .. 127;
  SB : Signed_Byte;
begin
  SB := -1;
  B := Byte'Mod (SB);  -- OK (255)
```

# Package Interfaces

- **Standard** package
- Integer types with **defined bit length**

```
type My_Base_Integer is new Integer;  
pragma Assert (My_Base_Integer'First = -2**31);  
pragma Assert (My_Base_Integer'Last = 2**31-1);
```

- Dealing **with** hardware registers

- Note: Shorter may not be faster for integer maths
  - Modern 64-bit machines are not efficient at 8-bit maths

```
type Integer_8 is range -2**7 .. 2**7-1;  
for Integer_8'Size use 8;  
-- and so on for 16, 32, 64 bit types...
```

# Shift/Rotate Functions

- In Interfaces package
  - Shift\_Left
  - Shift\_Right
  - Shift\_Right\_Arithmetic
  - Rotate\_Left
  - etc.
- See RM B.2 - *The Package Interfaces*

## Bit-Oriented Operations Example

- Assuming `Unsigned_16` is used
  - 16-bits modular

```
with Interfaces;  
use Interfaces;  
...  
procedure Swap (X : in out Unsigned_16) is  
begin  
  X := (Shift_Left (X,8) and 16#FF00#) or  
       (Shift_Right (X,8) and 16#00FF#);  
end Swap;
```

## Why No Implicit Shift and Rotate?

- Arithmetic, logical operators available **implicitly**
- **Why not** Shift, Rotate, etc. ?
- By **excluding** other solutions
  - As functions in **standard** → May **hide** user-defined declarations
  - As new **operators** → New operators for a **single type**
  - As **reserved words** → Not **upward compatible**

# Shift/Rotate for User-Defined Types

- **Must** be modular types
- Approach 1: use Interfaces's types
  - `Unsigned_8`, `Unsigned_16` ...
- Approach 2: derive from Interfaces's types
  - Operations are **inherited**
  - More on that later

```
type Byte is new Interfaces.Unsigned_8;
```

- Approach 3: use GNAT's intrinsic
  - Conditions on function name and type representation
  - See GNAT UG 8.11

```
function Shift_Left  
  (Value : T;  
   Amount : Natural) return T with Import,  
                                     Convention => Intrinsic;
```

# Quiz

```
type T is mod 256;
```

```
V : T := 255;
```

Which statement(s) is (are) legal?

A. `V := V + 1`

B. `V := 16#ff#`

C. `V := 256`

D. `V := 255 + 1`

# Quiz

```
type T is mod 256;
```

```
V : T := 255;
```

Which statement(s) is (are) legal?

A. `V := V + 1`

B. `V := 16#ff#`

C. `V := 256`

D. `V := 255 + 1`

# Quiz

```
with Interfaces; use Interfaces;
```

```
type T1 is new Unsigned_8;  
V1 : T1 := 255;
```

```
type T2 is mod 256;  
V2 : T2 := 255;
```

Which statement(s) is (are) legal?

- A. V1 := Rotate\_Left (V1, 1)
- B. V1 := Positive'First
- C. V2 := 1 and V2
- D. V2 := Rotate\_Left (V2, 1)
- E. V2 := T2'Mod (2.0)

# Quiz

```
with Interfaces; use Interfaces;
```

```
type T1 is new Unsigned_8;  
V1 : T1 := 255;
```

```
type T2 is mod 256;  
V2 : T2 := 255;
```

Which statement(s) is (are) legal?

- A. `V1 := Rotate_Left (V1, 1)`
- B. `V1 := Positive'First`
- C. `V2 := 1 and V2`
- D. `V2 := Rotate_Left (V2, 1)`
- E. `V2 := T2'Mod (2.0)`

## Enumeration Types

# Enumeration Types

- Enumeration of **logical** values
  - Integer value is an implementation detail
- Syntax

```
type <identifier> is (<identifier-list>) ;
```

- Literals
  - Distinct, ordered
  - Can be in **multiple** enumerations

```
type Colors is (Red, Orange, Yellow, Green, Blue, Violet);
```

```
type Stop_Light is (Red, Yellow, Green);
```

```
...
```

```
-- Red both a member of Colors and Stop_Light
```

```
Shade : Colors := Red;
```

```
Light : Stop_Light := Red;
```

# Enumeration Type Operations

- Assignment, relationals
- **Not** numeric quantities
  - *Possible* with attributes
  - Not recommended

```
type Directions is (North, South, East, West);
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
Heading : Directions;
Today, Tomorrow : Days;
...
Today := Mon;
Today := North; -- compile error
Heading := South;
Heading := East + 1; -- compile error
if Today < Tomorrow then ...
```

# Character Types

- Literals
  - Enclosed in single quotes eg. 'A'
  - Case-sensitive
- **Special-case** of enumerated type
  - At least one character enumeral
- System-defined **Character**
- Can be user-defined

```
type EBCDIC is (nul, ..., 'a' , ..., 'A', ..., del);  
Control : EBCDIC := 'A';  
Nullo : EBCDIC := nul;
```

## Language-Defined Type Boolean

- Enumeration

```
type Boolean is (False, True);
```

- Supports assignment, relational operators, attributes

```
A : Boolean;
```

```
Counter : Integer;
```

```
...
```

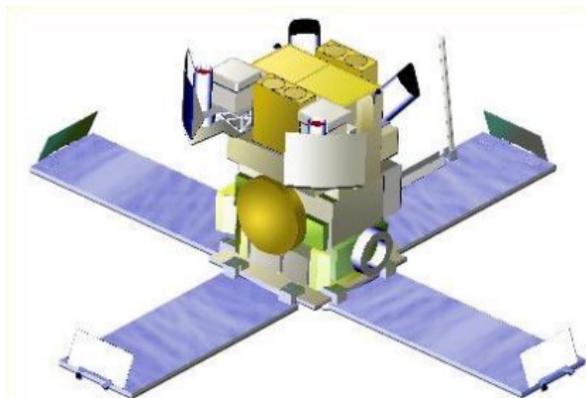
```
A := (Counter = 22);
```

- Logical operators **and**, **or**, **xor**, **not**

```
A := B or (not C); -- For A, B, C boolean
```

# Why Boolean Isn't Just an Integer?

- Example: Real-life error
  - HETE-2 satellite **attitude control** system software (ACS)
  - Written in **C**
- Controls four "solar paddles"
  - Deployed after launch



# Why Boolean Isn't Just an Integer!

- **Initially** variable with paddles' state
  - Either **all** deployed, or **none** deployed

- Used `int` as a boolean

```
if (rom->paddles_deployed == 1)
    use_deployed_inertia_matrix();
else
    use_stowed_inertia_matrix();
```

- Later `paddles_deployed` became a **4-bits** value
  - One bit per paddle
  - `0` → none deployed, `0xF` → all deployed
- Then, `use_deployed_inertia_matrix()` if only first paddle is deployed!
- Better: boolean function `paddles_deployed()`
  - Single line to modify

## Boolean Operators' Operand Evaluation

- Evaluation order **not specified**
- May be needed
  - Checking value **before** operation
  - Dereferencing null pointers
  - Division by zero

```
if Divisor /= 0 and K / Divisor = Max then ... -- Problem!
```

## Short-Circuit Control Forms

- **Short-circuit** → **fixed** evaluation order
- Left-to-right
- Right only evaluated **if necessary**

- **and then**: if left is False, skip right

Divisor /= 0 **and then** K / Divisor = Max

- **or else**: if left is True, skip right

Divisor = 0 **or else** K / Divisor = Max

# Quiz

```
type Enum_T is (Able, Baker, Charlie);
```

Which statement(s) is (are) legal?

- A. V1 : Enum\_T := Enum\_T'Value ("Able");
- B. V2 : Enum\_T := Enum\_T'Value ("BAKER");
- C. V3 : Enum\_T := Enum\_T'Value (" charlie ");
- D. V4 : Enum\_T := Enum\_T'Value ("Able Baker Charlie");

# Quiz

```
type Enum_T is (Able, Baker, Charlie);
```

Which statement(s) is (are) legal?

- A. `V1 : Enum_T := Enum_T'Value ("Able");`
- B. `V2 : Enum_T := Enum_T'Value ("BAKER");`
- C. `V3 : Enum_T := Enum_T'Value (" charlie ");`
- D. `V4 : Enum_T := Enum_T'Value ("Able Baker Charlie");`

Explanations

- A. Legal
- B. Legal - conversion is case-insensitive
- C. Legal - leading/trailing blanks are ignored
- D. Value tries to convert entire string, which will fail at run-time

## Representation Values

# Enumeration Representation Values

- Numeric **representation** of enumerals

- Position, unless redefined
- Redefinition syntax

```
type Enum_T is (Able, Baker, Charlie, David);  
for Enum_T use  
  (Able => 3, Baker => 15, Charlie => 63, David => 255);
```

- Enumerals are ordered **logically** (not by value)

- Prior to Ada 2022

- Only way to get value is through `Unchecked_Conversion`

```
function Value is new Ada.Unchecked_Conversion  
  (Enum_T, Integer_8);  
I : Integer_8;  
  
begin  
  I := Value (Charlie);
```

- New attributes in Ada 2022

- `'Enum_Rep` to get representation value

```
Charlie'Enum_Rep → 63
```

- `'Enum_Val` to convert integer to enumeral (if possible)

```
Enum_T'Enum_Val (15) → Baker
```

```
Enum_T'Enum_Val (16) → raise Constraint_Error
```

## Order Attributes for All Discrete Types

- **All discrete** types, mostly useful for enumerated types
- T'Pos (Input)
  - "Logical position number" of Input
- T'Val (Input)
  - Converts "logical position number" to T

```
type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat); -- 0 .. 6
Today    : Days := Some_Value;
Position : Integer;
...
Position := Days'Pos (Today);
...
Get (Position);
Today := Days'Val (Position);
```

# Quiz

```
type T is (Left, Top, Right, Bottom);  
V : T := Left;
```

Which of the following proposition(s) are true?

- A.  $T'Value(V) = 1$
- B.  $T'Pos(V) = 0$
- C.  $T'Image(T'Pos(V)) = Left$
- D.  $T'Val(T'Pos(V) - 1) = Bottom$

# Quiz

```
type T is (Left, Top, Right, Bottom);  
V : T := Left;
```

Which of the following proposition(s) are true?

- A.  $T'Value(V) = 1$
- B.  $T'Pos(V) = 0$
- C.  $T'Image(T'Pos(V)) = Left$
- D.  $T'Val(T'Pos(V) - 1) = Bottom$

## Character Types

# Language-Defined Character Types

## ■ Character

- 8-bit Latin-1
- Base component of `String`
- Uses attributes `'Image` / `'Value`

## ■ `Wide_Character`

- 16-bit Unicode
- Base component of `Wide_Strings`
- Uses attributes `'Wide_Image` / `'Wide_Value`

## ■ `Wide_Wide_Character`

- 32-bit Unicode
- Base component of `Wide_Wide_Strings`
- Uses attributes `'Wide_Wide_Image` / `'Wide_Wide_Value`

# Character Oriented Packages

- Language-defined
- `Ada.Characters.Handling`
  - Classification
  - Conversion
- `Ada.Characters.Latin_1`
  - Characters as constants
- See RM Annex A for details

# Ada.Characters.Latin\_1 Sample Content

```
package Ada.Characters.Latin_1 is
  NUL : constant Character := Character'Val (0);
  ...
  LF  : constant Character := Character'Val (10);
  VT  : constant Character := Character'Val (11);
  FF  : constant Character := Character'Val (12);
  CR  : constant Character := Character'Val (13);
  ...
  Commercial_At : constant Character := '@'; -- Character'Val (64)
  ...
  LC_A : constant Character := 'a'; -- Character'Val (97)
  LC_B : constant Character := 'b'; -- Character'Val (98)
  ...
  Inverted_Exclamation : constant Character := Character'Val (161);
  Cent_Sign             : constant Character := Character'Val (162);
  ...
  LC_Y_Diaeresis       : constant Character := Character'Val (255);
end Ada.Characters.Latin_1;
```

# Ada.Characters.Handling Sample Content

```
package Ada.Characters.Handling is
  function Is_Control      (Item : Character) return Boolean;
  function Is_Graphic     (Item : Character) return Boolean;
  function Is_Letter      (Item : Character) return Boolean;
  function Is_Lower       (Item : Character) return Boolean;
  function Is_Upper       (Item : Character) return Boolean;
  function Is_Basic       (Item : Character) return Boolean;
  function Is_Digit       (Item : Character) return Boolean;
  function Is_Decimal_Digit (Item : Character) return Boolean renames Is_Digit;
  function Is_Hexadecimal_Digit (Item : Character) return Boolean;
  function Is_Alphanumeric (Item : Character) return Boolean;
  function Is_Special     (Item : Character) return Boolean;
  function To_Lower (Item : Character) return Character;
  function To_Upper (Item : Character) return Character;
  function To_Basic (Item : Character) return Character;
  function To_Lower (Item : String) return String;
  function To_Upper (Item : String) return String;
  function To_Basic (Item : String) return String;
  ...
end Ada.Characters.Handling;
```

# Quiz

```
type T1 is (NUL, A, B, 'C');  
for T1 use (NUL => 0, A => 1, B => 2, 'C' => 3);  
type T2 is array (Positive range <>) of T1;  
Obj : T2 := "CC" & A & NUL;
```

Which of the following proposition(s) is (are) true

- A. The code fails at run-time
- B. Obj'Length = 3
- C. Obj (1) = 'C'
- D. Obj (3) = A

## Quiz

```
type T1 is (NUL, A, B, 'C');  
for T1 use (NUL => 0, A => 1, B => 2, 'C' => 3);  
type T2 is array (Positive range <>) of T1;  
Obj : T2 := "CC" & A & NUL;
```

Which of the following proposition(s) is (are) true

- A. The code fails at run-time
- B. `Obj'Length = 3`
- C. `Obj (1) = 'C'`
- D. `Obj (3) = A`

# Quiz

```
with Ada.Characters.Latin_1;  
use  Ada.Characters.Latin_1;  
with Ada.Characters.Handling;  
use  Ada.Characters.Handling;
```

Which of the following proposition(s) are true?

- A. `NUL = 0`
- B. `NUL = '\0'`
- C. `Character'Pos (NUL) = 0`
- D. `Is_Control (NUL)`

# Quiz

```
with Ada.Characters.Latin_1;  
use Ada.Characters.Latin_1;  
with Ada.Characters.Handling;  
use Ada.Characters.Handling;
```

Which of the following proposition(s) are true?

- A. `NUL = 0`
- B. `NUL = '\0'`
- C. `Character'Pos (NUL) = 0`
- D. `Is_Control (NUL)`

## Real Types

# Real Types

- Approximations to **continuous** values
  - 1.0, 1.1, 1.11, 1.111 ... 2.0, ...
  - Finite hardware → approximations
- Floating-point
  - **Variable** exponent
  - **Large** range
  - Constant **relative** precision
- Fixed-point
  - **Constant** exponent
  - **Limited** range
  - Constant **absolute** precision
  - Subdivided into Binary and Decimal
- Class focuses on floating-point

## Real Type (Floating and Fixed) Literals

- **Must** contain a fractional part
- No silent promotion

```
type Phase is digits 8; -- floating-point
```

```
OK : Phase := 0.0;
```

```
Bad : Phase := 0 ; -- compile error
```

# Declaring Floating Point Types

## ■ Syntax

```
type <identifier> is  
    digits <expression> [range constraint];
```

- *digits* → **minimum** number of significant digits
- **Decimal** digits, not bits

## ■ Compiler chooses representation

- From **available** floating point types
- May be **more** accurate, but not less
- If none available → declaration is **rejected**

## ■ System.Max\_Digits - constant specifying maximum digits of precision available for runtime

```
type Very_Precise_T is digits System.Max_Digits;
```

*Need to do **with** System; to get visibility*

## Predefined Floating Point Types

- Type `Float`  $\geq$  6 digits
- Additional implementation-defined types
  - `Long_Float`  $\geq$  11 digits
- General-purpose

 **Tip**

It is best, and easy, to **avoid** predefined types

- To keep **portability**

## Floating Point Type Operators

- By increasing precedence

relational operator = | /= | < | >= | > | >=

binary adding operator + | -

unary adding operator + | -

multiplying operator \* | /

highest precedence operator \*\* | **abs**

### Note

Exponentiation (\*\*) result will be real

- So power must be **Integer**
  - Not possible to ask for root
  - `X**0.5` → `sqrt (x)`

# Floating Point Type Attributes

## ■ Core attributes

```
type My_Float is digits N;  -- N static
```

### ■ My\_Float'Digits

- Number of digits **requested** (N)

### ■ My\_Float'Base'Digits

- Number of **actual** digits

### ■ My\_Float'Rounding (X)

- Integral value nearest to X
- *Note:* Float'Rounding (0.5) = 1 and  
Float'Rounding (-0.5) = -1

## ■ Model-oriented attributes

- Advanced machine representation of the floating-point type
- Mantissa, strict mode

## Numeric Types Conversion

- Ada's integer and real are **numeric**
  - Holding a numeric value
- Special rule: can always convert between numeric types
  - Explicitly

### ⚠ Warning

Float → Integer causes **rounding**

**declare**

```
N : Integer := 0;
```

```
F : Float := 1.5;
```

**begin**

```
N := Integer (F); -- N = 2
```

```
F := Float (N); -- F = 2.0
```

# Quiz

What is the output of this code?

```
declare
  F : Float := 7.6;
  I : Integer := 10;
begin
  F := Float (Integer (F) / I);
  Put_Line (Float'Image (F));
end;
```

- A. 7.6E-01
- B. Compile Error
- C. 8.0E-01
- D. 0.0

# Quiz

What is the output of this code?

```
declare
  F : Float := 7.6;
  I : Integer := 10;
begin
  F := Float (Integer (F) / I);
  Put_Line (Float'Image (F));
end;
```

- A. 7.6E-01
- B. Compile Error
- C. 8.0E-01
- D. **0.0**

Explanations

- A. Result of `F := F / Float (I);`
- B. Result of `F := F / I;`
- C. Result of `F := Float (Integer (F)) / Float (I);`
- D. Integer value of F is 8. Integer result of dividing that by 10 is 0. Converting to float still gives us 0

Base Type

# Base Ranges

- Actual **hardware-supported** numeric type used
  - GNAT makes consistent and predictable choices on all major platforms
- **Predefined** operators
  - Work on full-range
    - **No range checks** on inputs or result
    - Best performance
  - Implementation may use wider registers
    - Intermediate values
- Can be accessed with 'Base attribute

```
type Foo is range -30_000 .. 30_000;  
function "+" (Left, Right : Foo'Base) return Foo'Base;
```

- Base range
  - Signed
  - 8 bits → -128 .. 127
  - 16 bits → -32\_768 .. 32767

# Compile-Time Constraint Violation

- *May* produce **warnings**
  - And compile successfully
- *May* produce **errors**
  - And fail at compilation
- Requirements for rejection
  - Static value
  - Value not in range of **base** type
  - Compilation is **impossible**

```
procedure Test is
  type Some_Integer is range -200 .. 200;
  Object : Some_Integer;
begin
  Object := 50_000; -- probable error
end;
```

# Range Check Failure

- Compile-time rejection
  - Depends on **base** type
  - Selected by the compiler
  - Depends on underlying **hardware**
  - Early error → "Best" case
- Else run-time **exception**
  - Most cases
  - Be happy when compilation failed instead

## Real Base Decimal Precision

- Real types precision may be **better** than requested
- Example:
  - Available: 6, 12, or 24 digits of precision
  - Type with **8 digits** of precision

```
type My_Type is digits 8;
```
  - My\_Type will have 12 **or** 24 digits of precision

## Floating Point Division by Zero

- Language-defined do as the machine does
  - If T'Machine\_Overflows attribute is True raises Constraint\_Error
  - Else  $+\infty / -\infty$ 
    - Better performance
- User-defined types always raise Constraint\_Error

```
subtype MyFloat is Float range Float'First .. Float'Last;  
type MyFloat is new Float range Float'First .. Float'Last;
```

## Using Equality for Floating Point Types

- Questionable: representation issue
  - Equality  $\rightarrow$  identical bits
  - Approximations  $\rightarrow$  hard to **analyze**, and **not portable**
  - Related to floating-point, not Ada
- Perhaps define your own function
  - Comparison within tolerance ( $+\varepsilon / -\varepsilon$ )

## Miscellaneous

## Checked Type Conversions

- Between "closely related" types
  - Numeric types
  - Inherited types
  - Array types
- Illegal conversions **rejected**
  - Unsafe **Unchecked\_Conversion** available
- Called as if it was a function
  - Named using destination type name
    - Target\_Float := Float (Source\_Integer);
  - Implicitly defined
  - **Must** be explicitly called

# Default Value

- Not defined by language for **scalars**
- Can be done with an **aspect clause**
  - Only during type declarations
  - <value> must be static

```
type Type_Name is <type_definition>  
    with Default_Value => <value>;
```

- Example

```
type Tertiary_Switch is (Off, On, Neither)  
    with Default_Value => Neither;  
Implicit : Tertiary_Switch; -- Implicit = Neither  
Explicit : Tertiary_Switch := Neither;
```

# Simple Static Type Derivation

- New type from an existing type
  - **Limited** form of inheritance: operations
  - **Not** fully OOP
  - More details later
- Strong type benefits
  - Only **explicit** conversion possible
  - eg. Meters can't be set from a Feet value

- Syntax

```
type identifier is new Base_Type [<constraints>]
```

- Example

```
type Measurement is digits 6;  
type Distance is new Measurement  
    range 0.0 .. Measurement'Last;
```

## Subtypes

# Subtype

- May **constrain** an existing type
- Still the **same** type
- Syntax

```
subtype Defining_Identifier is Type_Name [constraints];
```

- Type\_Name is an existing **type** or **subtype**

## Note

If no constraint → type alias

## Subtype Example

- Enumeration type with **range** constraint

```
type Days is (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);  
subtype Weekdays is Days range Mon .. Fri;  
Workday : Weekdays; -- type Days limited to Mon .. Fri
```

- Equivalent to **anonymous** subtype

```
Same_As_Workday : Days range Mon .. Fri;
```

## Kinds of Constraints

- Range constraints on scalar types

```
subtype Positive is Integer range 1 .. Integer'Last;  
subtype Natural is Integer range 0 .. Integer'Last;  
subtype Weekdays is Days range Mon .. Fri;  
subtype Symmetric_Distribution is  
    Float range -1.0 .. +1.0;
```

- Other kinds, discussed later
- Constraints apply only to values
- Representation and set of operations are **kept**

# Subtype Constraint Checks

- Constraints are checked
  - At initial value assignment
  - At assignment
  - At subprogram call
  - Upon return from subprograms
- Invalid constraints
  - Will cause `Constraint_Error` to be raised
  - May be detected at compile time
    - If values are **static**
    - Initial value → error
    - ... else → warning

```
Max : Integer range 1 .. 100 := 0; -- compile error
```

```
...
```

```
Max := 0; -- run-time error
```

# Performance Impact of Constraints Checking

- Constraint checks have run-time performance impact
- The following code

```
procedure Demo is
  K : Integer := F;
  P : Integer range 0 .. 100;
begin
  P := K;
```

- Generates assignment checks similar to

```
if K < 0 or K > 100 then
  raise Constraint_Error;
else
  P := K;
end if;
```

- These checks can be disabled with `-gnatp`

## Optimizations of Constraint Checks

- Checks happen only if necessary
- Compiler assumes variables to be **initialized**
- So this code generates **no check**

```
procedure Demo is
  P, K : Integer range 0 .. 100;
begin
  P := K;
  -- But K is not initialized!
```

## Range Constraint Examples

```
subtype Proper_Subset is Positive range 1 .. 10;
subtype Same_Constraints is Positive
    range 1 .. Integer'Last;
subtype Letter is Character range 'A' .. 'z';
subtype Upper_Case is Letter range 'A' .. 'Z';
subtype Lower_Case is Letter range 'a' .. 'z';
subtype Null_Range is Integer
    range 1 .. 0;  -- silly when hard-coded...
-- evaluated when subtype defined, not when object declared
subtype Dynamic is Integer range Lower .. Upper;
```

# Quiz

```
type Enum_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);  
subtype Enum_Sub_T is Enum_T range Mon .. Fri;
```

Which subtype definition is valid?

- A.** `subtype A is Enum_Sub_T range Enum_Sub_T'Pred  
    (Enum_Sub_T'First) .. Enum_Sub_T'Last;`
- B.** `subtype B is range Sat .. Mon;`
- C.** `subtype C is Integer;`
- D.** `subtype D is digits 6;`

# Quiz

```
type Enum_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);  
subtype Enum_Sub_T is Enum_T range Mon .. Fri;
```

Which subtype definition is valid?

- A. `subtype A is Enum_Sub_T range Enum_Sub_T'Pred (Enum_Sub_T'First) .. Enum_Sub_T'Last;`
- B. `subtype B is range Sat .. Mon;`
- C. `subtype C is Integer;`
- D. `subtype D is digits 6;`

Explanations

- A. This generates a run-time error because the first enumerals specified is not in the range of Enum\_Sub\_T
- B. Compile error - no type specified
- C. Correct - standalone subtype
- D. `Digits 6` is used for a type definition, not a subtype

## Subtypes - Full Picture

# Implicit Subtype

- The declaration

```
type Typ is range L .. R;
```

- Is short-hand for

```
type <Anon> is new Predefined_Integer_Type;  
subtype Typ is <Anon> range L .. R;
```

- <Anon> is the *Base* type of Typ
  - Accessed with Typ'Base

## Implicit Subtype Explanation

```
type <Anon> is new Predefined_Integer_Type;  
subtype Typ is <Anon> range L .. R;
```

- Compiler chooses a standard integer type that includes L .. R
  - `Integer`, `Short_Integer`, `Long_Integer`, etc.
  - **Implementation-defined** choice, non portable
- New anonymous type <Anon> is derived from the predefined type
- <Anon> inherits the type's operations (+, - ...)
- Typ, subtype of <Anon> is created with `range L .. R`
- Typ'Base will return the type <Anon>

## Stand-Alone (Sub)Type Names

- Denote all the values of the type or subtype
  - Unless explicitly constrained

```
subtype Constrained_Sub is Integer range 0 .. 10;
subtype Just_A_Rename is Integer;
X : Just_A_Rename;
...
for I in Constrained_Sub loop
  X := I;
end loop;
```

# Subtypes Localize Dependencies

- Single points of change
- Relationships captured in code
- No subtypes

```
type Vector is array (1 .. 12) of Some_Type;
```

```
K : Integer range 0 .. 12 := 0; -- anonymous subtype
```

```
Values : Vector;
```

```
...
```

```
if K in 1 .. 12 then ...
```

```
for J in Integer range 1 .. 12 loop ...
```

- Subtypes

```
type Counter is range 0 .. 12;
```

```
subtype Index is Counter range 1 .. Counter'Last;
```

```
type Vector is array (Index) of Some_Type;
```

```
K : Counter := 0;
```

```
Values : Vector;
```

```
...
```

```
if K in Index then ...
```

```
for J in Index loop ...
```

## Subtypes May Enhance Performance

- Provides compiler with more information
- Redundant checks can more easily be identified

```
subtype Index is Integer range 1 .. Max;  
type Vector is array (Index) of Float;  
K : Index;  
Values : Vector;  
...  
K := Some_Value;    -- range checked here  
Values (K) := 0.0;  -- so no range check needed here
```

# Subtypes Don't Cause Overloading

- Illegal code: re-declaration of **F**

```
type A is new Integer;  
subtype B is A;  
function F return A is (0);  
function F return B is (1);
```

# Default Values and Option Types

- Not allowed: Defaults on new **type** only
  - **subtype** is still the same type
- **Note:** Default value may violate subtype constraints
  - Compiler error for static definition
  - `Constraint_Error` otherwise

```
type Tertiary_Switch is (Off, On, Neither)
  with Default_Value => Neither;
subtype Toggle_Switch is Tertiary_Switch
  range Off .. On;
Safe : Toggle_Switch := Off;
Implicit : Toggle_Switch; -- compile error: out of range
```

## Tip

Using a meaningless value (Neither) to extend the range of the type is turning it into an **option type**. This idiom is very rich and allows for e.g. "in-flow" errors handling.

# Attributes Reflect the Underlying Type

```
type Color is
```

```
(White, Red, Yellow, Green, Blue, Brown, Black);
```

```
subtype Rainbow is Color range Red .. Blue;
```

- T'First and T'Last respect constraints

- Rainbow'First → Red *but* Color'First → White

- Rainbow'Last → Blue *but* Color'Last → Black

- Other attributes reflect base type

- Color'Succ (Blue) = Brown = Rainbow'Succ (Blue)

- Color'Pos (Blue) = 4 = Rainbow'Pos (Blue)

- Color'Val (0) = White = Rainbow'Val (0)

- Assignment must still satisfy target constraints

```
Shade : Color range Red .. Blue := Brown;  -- run-time error
```

```
Hue : Rainbow := Rainbow'Succ (Blue);      -- run-time error
```

# Valid attribute

- The\_Type'Valid is a **Boolean**
- True → the current representation for the given scalar is valid

```
procedure Main is
  subtype Small_T is Integer range 1 .. 3;
  Big   : aliased Integer := 0;
  Small : Small_T with Address => Big'Address;
begin
  for V in 0 .. 5 loop
    Big := V;
    Put_Line (Big'Image & " => " & Boolean'Image (Small'Valid));
  end loop;
end Main;
```

0 => FALSE

1 => TRUE

2 => TRUE

3 => TRUE

4 => FALSE

5 => FALSE

## Idiom: Extended Ranges

- Count / Positive\_Count
  - Sometimes as Type\_Ext (extended) / Type
  - For counting vs indexing
    - An index goes from 1 to max length
    - A count goes from 0 to max length

-- *ARM A.10.1*

```
package Text_IO is
...
type Count is range 0 .. implementation-defined;
subtype Pos_Count is Count range 1 .. Count'Last;
```

# Idiom: Partition

- Useful for splitting-up large enums

### ⚠ Warning

Be careful about checking that the partition is complete when items are added/removed.  
With a **case**, the compiler automatically checks that for you.

### 💡 Tip

Can have non-consecutive values with the Predicate aspect.

```

type Commands_T is (Lights_On, Lights_Off, Read, Write, Accelerate, Stop);
-- Complete partition of the commands
subtype IO_Commands_T is Commands_T range Read .. Write;
subtype Lights_Commands_T is Commands_T range Lights_On .. Lights_Off;
subtype Movement_Commands_T is Commands_T range Accelerate .. Stop;

subtype Physical_Commands_T is Commands_T
  with Predicate => Physical_Commands_T in Lights_Commands_T | Movement_Commands_T;

procedure Execute_Light_Command (C : Lights_Commands_T);

procedure Execute_Command (C : Commands_T) is
begin
  case C in -- partition must be exhaustive
    when Lights_Commands_T => Execute_Light_Command (C);
  ...

```

# Idiom: Subtypes as Local Constraints

- Can replace defensive code
- Can be very useful in some identified cases
- Subtypes accept dynamic bounds, unlike types
- Checks happen through type-system
  - Can be disabled with `-gnatp`, unlike conditionals
  - Can also be a disadvantage

**⚠ Warning**

Do not use for checks that should **always** happen, even in production.

- Constrain input range

```
subtype Incrementable_Integer is Integer range Integer'First .. Integer'Last - 1;  
function Increment (I : Incrementable_Integer) return Integer;
```

- Constrain output range

```
subtype Valid_Fingers_T is Integer range 1 .. 5;  
Fingers : Valid_Fingers_T := Prompt_And_Get_Integer ("Give me the number of a finger");
```

- Constrain array index

```
procedure Read_Index_And_Manipulate_Char (S : String) is  
  subtype S_Index is Positive range S'Range;  
  I : constant S_Index := Read_Positive;  
  C : Character renames S (I);
```

# Quiz

```
1  type T1 is range 0 .. 10;  
2  function "-" (V : T1) return T1;  
3  subtype T2 is T1 range 1 .. 9;  
4  function "-" (V : T2) return T2;  
5  
6  Obj : T2 := -T2 (1);
```

Which function is executed at line 6?

- A. The one at line 2
- B. The one at line 4
- C. A predefined "-" operator for integer types
- D. None: The code is illegal

# Quiz

```
1 type T1 is range 0 .. 10;
2 function "-" (V : T1) return T1;
3 subtype T2 is T1 range 1 .. 9;
4 function "-" (V : T2) return T2;
5
6 Obj : T2 := -T2 (1);
```

Which function is executed at line 6?

- A. The one at line 2
- B. The one at line 4
- C. A predefined "-" operator for integer types
- D. **None: The code is illegal**

The **type** is used for the overload profile, and here both T1 and T2 are of type T1, which means line 4 is actually a redeclaration, which is forbidden.

# Quiz

```
type T is range 0 .. 10;  
subtype S is T range 1 .. 9;
```

What is the value of `S'Succ (S (9))`?

- A. 9
- B. 10
- C. None, this fails at run-time
- D. None, this does not compile

## Quiz

```
type T is range 0 .. 10;  
subtype S is T range 1 .. 9;
```

What is the value of `S'Succ (S (9))`?

- A. 9
- B. **10**
- C. None, this fails at run-time
- D. None, this does not compile

`T'Succ` and `T'Pred` are defined on the **type**, not the **subtype**.

# Quiz

```
type T is new Integer range 0 .. Integer'Last;  
subtype S is T range 0 .. 10;
```

Obj : S;

What is the result of Obj := S'Last + 1?

- A. 0
- B. 11
- C. None, this fails at run-time
- D. None, this does not compile

# Quiz

```
type T is new Integer range 0 .. Integer'Last;  
subtype S is T range 0 .. 10;
```

Obj : S;

What is the result of Obj := S'Last + 1?

- A. 0
- B. 11
- C. ***None, this fails at run-time***
- D. None, this does not compile

## Very Large Numbers

## Why Float or Integer Just Doesn't Cut It

- Standard types (**Integer**, **Float**) have limited precision
- Overflow, underflow, and rounding errors are common
- Some applications need *arbitrary precision*

# Meet Ada.Numerics.Big\_Numbers

- Introduced in Ada 2022
- Provides arbitrary-precision arithmetic:
  - `type Big_Integer in package  
Ada.Numerics.Big_Numbers.Big_Integers`
  - `type Big_Real in package  
Ada.Numerics.Big_Numbers.Big_Reals`
- Ideal for high-accuracy applications
  - Create giant numbers or highly precise real values
  - Standard arithmetic operations
  - Control precision and rounding

## Big\_Integer in Action

```
with Ada.Numerics.Big_Numbers.Big_Integers;  
use Ada.Numerics.Big_Numbers.Big_Integers;  
  
procedure Test_Big_Integer is  
  X : Big_Integer := To_Big_Integer (2) ** 100;  
  Y : Big_Integer := X * 123456789;  
begin  
  Put_Line ("Result: " & Y'Image);  
end Test_Big_Integer;
```

- Handles numbers with hundreds or thousands of digits

## Big\_Real: Real Precision

```
with Ada.Numerics.Big_Numbers.Big_Reals;  
use Ada.Numerics.Big_Numbers.Big_Reals;  
  
procedure Test_Big_Real is  
  A : Big_Real := 3.141592653589793238462643383279;  
  B : Big_Real := A ** 10;  
begin  
  Put_Line ("Pi ^ 10 = " & B'Image);  
end Test_Big_Real;
```

- Arbitrary-precision floating-point numbers

## Who Needs Big\_Numbers?

- Scientific computing
- Finance and accounting
- Exact algorithms (symbolic math, computer algebra)
- Cryptography

### Note

Note that you typically wouldn't use `Big_Numbers` for cryptography because it's vulnerable to timing side-channels attacks.

## Issues with Big Numbers

- Slower than native types
- Be selective: use when precision is critical
- Not all math functions implemented (e.g., trig/log)
- Watch memory usage for huge values

Lab

# Basic Types Lab

- Create types to handle the following concepts
  - Determining average test score
    - Number of tests taken
    - Total of all test scores
  - Number of degrees in a circle
  - Collection of colors
- Create objects for the types you've created
  - Assign initial values to the objects
  - Print the values of the objects
- Modify the objects you've created and print the new values
  - Determine the average score for all the tests
  - Add 359 degrees to the initial circle value
  - Set the color object to the value right before the last possible value

## Using the "Prompts" Directory

- Course material should have a link to a **Prompts** folder
- Folder contains everything you need to get started on the lab
  - GNAT STUDIO project file **default.gpr**
  - Annotated / simplified source files
    - Source files are templates for lab solutions
    - Files compile as is, but don't implement the requirements
    - Comments in source files give hints for the solution
- To load prompt, either
  - From within GNAT STUDIO, select **File** → **Open Project** and navigate to and open the appropriate **default.gpr** **OR**
  - From a command prompt, enter

```
gnatstudio -P <full path to GPR file>
```

    - If you are in the appropriate directory, and there is only one GPR file, entering **gnatstudio** will start the tool and open that project
- These prompt folders should be available for most labs

## Basic Types Lab Hints

- Understand the properties of the types
  - Do you need fractions or just whole numbers?
  - What happens when you want the number to wrap?
- Predefined package **Ada.Text\_IO** is handy...
  - Procedure **Put\_Line** takes a **String** as the parameter
- Remember attribute **'Image** returns a **String**

<typemark>'Image (Object)

Object 'Image

## Basic Types Extra Credit

- See what happens when your data is invalid / illegal
  - Number of tests = 0
  - Assign a very large number to the test score total
  - Color type only has one value
  - Add a number larger than 360 to the circle value

# Basic Types Lab Solution - Declarations

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Main is
3
4      type Number_Of_Tests_T is range 0 .. 100;
5      type Test_Score_Total_T is digits 6 range 0.0 .. 10_000.0;
6
7      type Degrees_T is mod 360;
8
9      type Cymk_T is (Cyan, Magenta, Yellow, Black);
10
11     Number_Of_Tests   : Number_Of_Tests_T;
12     Test_Score_Total : Test_Score_Total_T;
13
14     Angle : Degrees_T;
15
16     Color : Cymk_T;
```

# Basic Types Lab Solution - Implementation

```
18 begin
19
20   -- assignment
21   Number_Of_Tests := 15;
22   Test_Score_Total := 1_234.5;
23   Angle           := 180;
24   Color           := Magenta;
25
26   Put_Line (Number_Of_Tests'Image);
27   Put_Line (Test_Score_Total'Image);
28   Put_Line (Angle'Image);
29   Put_Line (Color'Image);
30
31   -- operations / attributes
32   Test_Score_Total := Test_Score_Total / Test_Score_Total_T (Number_Of_Tests);
33   Angle           := Angle + 359;
34   Color           := Cymk_T'Pred (Cymk_T'Last);
35
36   Put_Line (Test_Score_Total'Image);
37   Put_Line (Angle'Image);
38   Put_Line (Color'Image);
39
40 end Main;
```

## Summary

## Benefits of Strongly Typed Numerics

- **Prevent** subtle bugs
- Cannot mix Apples and Oranges
- Force to clarify **representation** needs
  - eg. constant with or with fractional part

```
type Yen is range 0 .. 1_000_000;  
type Ruble is range 0 .. 1_000_000;  
Mine : Yen := 1;  
Yours : Ruble := 1;  
Mine := Yours; -- illegal
```

# User-Defined Numeric Type Benefits

- Close to **requirements**
  - Types with **explicit** requirements (range, precision, etc.)
  - Best case: Incorrect state **not possible**
- Either implemented/respected or rejected
  - No run-time (bad) surprise
- **Portability** enhanced
  - Reduced hardware dependencies

# Summary

- User-defined types and strong typing is **good**
  - Programs written in application's terms
  - Computer in charge of checking constraints
  - Security, reliability requirements have a price
  - Performance **identical**, given **same requirements**
- User definitions from existing types *can* be good
- Right **trade-off** depends on **use-case**
  - More types → more precision → less bugs
  - Storing **both** feet and meters in **Float** has caused bugs
  - More types → more complexity → more bugs
  - A `Green_Round_Object_Altitude` type is probably **never needed**
- Default initialization is **possible**
  - Use **sparingly**

# Statements

# Introduction

# Statement Kinds

- Simple
  - `null`
  - `A := B` (assignments)
  - `exit`
  - `goto`
  - `delay`
  - `raise`
  - `P (A, B)` (procedure calls)
  - `return`
  - Tasking-related: `requeue`, entry call T.E (A, B), `abort`
- Compound
  - `if`
  - `case`
  - `loop` (and variants)
  - `declare`
  - Tasking-related: `accept`, `select`

*Tasking-related are seen in the tasking chapter*

## Procedure Calls (Overview)

- Procedures must be defined before they are called

```
procedure Activate (This : in out Foo;  
                  Flag : Boolean);
```

- Procedure calls are statements

- Traditional call notation

```
Activate (Idle, True);
```

- "Distinguished Receiver" notation

```
Idle.Activate (True);
```

- More details in "Subprograms" section

## Block Statements

# Block Statements

- Local **scope**
- Optional declarative part
- Used for
  - Temporary declarations
  - Declarations as part of statement sequence
  - Local catching of exceptions
- Syntax

```
[block-name :]  
[declare <declarative part> ]  
begin  
    <statements>  
end [block-name];
```

# Block Statements Example

```
begin
  Get (V);
  Get (U);
  if U > V then -- swap them
    Swap: declare
      Temp : Integer;
    begin
      Temp := U;
      U := V;
      V := Temp;
    end Swap;
    -- Temp does not exist here
  end if;
  Print (U);
  Print (V);
end;
```

## Null Statements

# Null Statements

- Explicit no-op statement
- Constructs with required statement
- Explicit statements help compiler
  - Oversights
  - Editing accidents

```
case Today is
  when Monday .. Thursday =>
    Work (9.0);
  when Friday =>
    Work (4.0);
  when Saturday .. Sunday =>
    null;
end case;
```

## Assignment Statements

# Assignment Statements

- Syntax

```
<variable> := <expression>;
```

- Value of expression is copied to target variable
- The type of the RHS must be same as the LHS
  - Rejected at compile-time otherwise

## declare

```
type Miles_T is range 0 .. Max_Miles;  
type Km_T is range 0 .. Max_Kilometers
```

```
M : Miles_T := 2; -- universal integer legal for any integer  
K : Km_T := 2; -- universal integer legal for any integer
```

## begin

```
M := K; -- compile error
```

# Assignment Statements, Not Expressions

- Separate from expressions

- No Ada equivalent for these:

```
int a = b = c = 1;
while (line = readline(file))
    { ...do something with line... }
```

- No assignment in conditionals

- E.g. `if (a == 1)` compared to `if (a = 1)`

# Assignable Views

- A `view` controls the way an entity can be treated
  - At different points in the program text
- The named entity must be an assignable variable
  - Thus the view of the target object must allow assignment
- Various un-assignable views
  - Constants
  - Variables of `limited` types
  - Formal parameters of mode `in`

```
Max : constant Integer := 100;
```

```
...
```

```
Max := 200; -- illegal
```

# Aliasing the Assignment Target

Ada 2022

- C allows you to simplify assignments when the target is used in the expression. This avoids duplicating (possibly long) names.

```
total = total + value;  
// becomes  
total += value;
```

- Ada 2022 implements this by using the target name symbol @

```
Total := Total + Value;  
-- becomes  
Total := @ + Value;
```

- Benefit

- Symbol can be used multiple times in expression

```
Value := (if @ > 0 then @ else -(@));
```

- Limitation

- Symbol is read-only (so it can't change during evaluation)

```
function Update (X : in out Integer) return Integer;  
function Increment (X: Integer) return Integer;
```

```
13 Value := Update (@);  
14 Value := Increment (@);
```

```
example.adb:13:21: error: actual for "X" must be a  
variable
```

## Quiz

```
type One_T is range 0 .. 100;  
type Two_T is range 0 .. 100;  
A : constant := 100;  
B : constant One_T := 99;  
C : constant Two_T := 98;  
X : One_T := 0;  
Y : Two_T := 0;
```

Which block(s) is (are) legal?

- A. X := A;  
Y := A;
- B. X := B;  
Y := C;
- C. X := One\_T(X + C);
- D. X := One\_T(Y);  
Y := Two\_T(X);

# Quiz

```
type One_T is range 0 .. 100;
type Two_T is range 0 .. 100;
A : constant := 100;
B : constant One_T := 99;
C : constant Two_T := 98;
X : One_T := 0;
Y : Two_T := 0;
```

Which block(s) is (are) legal?

- A. `X := A;`  
`Y := A;`
- B. `X := B;`  
`Y := C;`
- C. `X := One_T(X + C);`
- D. `X := One_T(Y);`  
`Y := Two_T(X);`

Explanations

- A. Legal - A is an untyped constant
- B. Legal - B, C are correctly typed
- C. Illegal - No such "+" operator: must convert operand individually
- D. Legal - Correct conversion and types

## Conditional Statements

# If-then-else Statements

- Control flow using Boolean expressions
- Syntax

```
if <boolean expression> then -- No parentheses
    <statements>;
[else
    <statements>;]
end if;
```

- At least one statement must be supplied
  - `null` for explicit no-op

# If-then-elsif Statements

- Sequential choice with alternatives
- Avoids **if** nesting
- **elsif** alternatives, tested in textual order
- **else** part still optional

```
1  if Valve (N) /= Closed then 1  if Valve (N) /= Closed then
2    Isolate (Valve (N));      2    Isolate (Valve (N));
3    Failure (Valve (N));      3    Failure (Valve (N));
4  else                        4    elsif System = Off then
5    if System = Off then      5    Failure (Valve (N));
6    Failure (Valve (N));      6  end if;
7  end if;
8  end if;
```

# Case Statements

- Exclusionary choice among alternatives
- Syntax

```
case <expression> is
  when <choice> => <statements>;
  { when <choice> => <statements>; }
end case;
```

```
choice ::= <expression> | <discrete range>
         | others { "|" <other choice> }
```

## Simple "case" Statements

```
type Directions is (Forward, Backward, Left, Right);
Direction : Directions;
...
case Direction is
  when Forward =>
    Set_Mode (Forward);
    Move (1);
  when Backward =>
    Set_Mode (Backup);
    Move (-1);
  when Left =>
    Turn (1);
  when Right =>
    Turn (-1);
end case;
```

*Note:* No fall-through between cases

# Case Statement Rules

- More constrained than a if-elsif structure
- **All** possible values must be covered
  - Explicitly
  - ... or with **others** keyword
- Choice values cannot be given more than once (exclusive)
  - Must be known at **compile** time

## Others Choice

- Choice by default
  - "everything not specified so far"
- Must be in last position

```
case Today is    -- work schedule
  when Monday =>
    Go_To (Work, Arrive=>Late, Leave=>Early);
  when Tuesday | Wednesday | Thursday => -- Several choices
    Go_To (Work, Arrive=>Early, Leave=>Late);
  when Friday =>
    Go_To (Work, Arrive=>Early, Leave=>Early);
  when others => -- weekend
    Go_To (Home, Arrive=>Day_Before, Leave=>Day_After);
end case;
```

## Case Statements Range Alternatives

```
case Altitude_Ft is
  when 0 .. 9 =>
    Set_Flight_Indicator (Ground);
  when 10 .. 40_000 =>
    Set_Flight_Indicator (In_The_Air);
  when others => -- Large altitude
    Set_Flight_Indicator (Too_High);
end case;
```

## Dangers of *Others* Case Alternative

- Maintenance issue: new value requiring a new alternative?
  - Compiler won't warn: `others` hides it

```
type Agencies_T is (NASA, ESA, RFSA); -- could easily grow
Bureau : Agencies_T;
...
case Bureau is
  when ESA =>
    Set_Region (Europe);
  when NASA =>
    Set_Region (America);
  when others =>
    Set_Region (Russia); -- New agencies will be Russian!
end case;
```

# Quiz

```
A : Integer := 100;
```

```
B : Integer := 200;
```

Which choice needs to be modified to make a valid `if` block

**A.** `if A == B and then A != 0 then`

```
    A := Integer'First;
```

```
    B := Integer'Last;
```

**B.** `elsif A < B then`

```
    A := B + 1;
```

**C.** `elsif A > B then`

```
    B := A - 1;
```

**D.** `end if;`

# Quiz

```
A : Integer := 100;
```

```
B : Integer := 200;
```

Which choice needs to be modified to make a valid `if` block

**A.** `if A == B and then A != 0 then`

```
    A := Integer'First;
```

```
    B := Integer'Last;
```

**B.** `elsif A < B then`

```
    A := B + 1;
```

**C.** `elsif A > B then`

```
    B := A - 1;
```

**D.** `end if;`

Explanations

- A uses the C-style equality/inequality operators
- D is legal because `else` is not required for an `if` block

# Quiz

```
type Enum_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
A : Enum_T;
```

Which choice needs to be modified to make a valid `case` block

```
case A is
```

- A. when Sun =>  
    Put\_Line ("Day Off");
- B. when Mon | Fri =>  
    Put\_Line ("Short Day");
- C. when Tue .. Thu =>  
    Put\_Line ("Long Day");
- D. end case;

# Quiz

```
type Enum_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
A : Enum_T;
```

Which choice needs to be modified to make a valid `case` block

```
case A is
```

- A. `when Sun =>`  
    `Put_Line ("Day Off");`
- B. `when Mon | Fri =>`  
    `Put_Line ("Short Day");`
- C. `when Tue .. Thu =>`  
    `Put_Line ("Long Day");`
- D. `end case;`

Explanations

- Ada requires all possibilities to be covered
- Add `when others` or `when Sat`

## Loop Statements

# Basic Loops and Syntax

- All kind of loops can be expressed
  - Optional iteration controls
  - Optional exit statements
- Syntax

```
[<name> :] [iteration_scheme] loop  
    <statements>  
end loop [<name>];
```

```
iteration_scheme ::= while <boolean expression>  
                  | for <loop_parameter_specification>  
                  | for <loop_iterator_specification>
```

- Example

```
Wash_Hair : loop  
    Lather (Hair);  
    Rinse (Hair);  
end loop Wash_Hair;
```

# Loop Exit Statements

- Leaves innermost loop
  - Unless loop name is specified
- Syntax

```
exit [<loop name>] [when <boolean expression>];
```

- `exit when` exits with condition

```
loop
...
-- If it's time to go then exit
exit when Time_to_Go;
...
end loop;
```

## Exit Statement Examples

- Equivalent to C's `do while`

```
loop
  Do_Something;
  exit when Finished;
end loop;
```

- Nested named loops and exit

```
Outer : loop
  Do_Something;
  Inner : loop
    ...
    exit Outer when Finished; -- will exit all the way out
    ...
  end loop Inner;
end loop Outer;
```

# While-loop Statements

## ■ Syntax

```
while boolean_expression loop
    sequence_of_statements
end loop;
```

## ■ Identical to

```
loop
    exit when not boolean_expression;
    sequence_of_statements
end loop;
```

## ■ Example

```
while Count < Largest loop
    Count := Count + 2;
    Display (Count);
end loop;
```

# For-loop Statements

- One low-level form
  - General-purpose (looping, array indexing, etc.)
  - Explicitly specified sequences of values
  - Precise control over sequence
- Two high-level forms
  - Focused on objects
  - Seen later with Arrays

# For in Statements

- Successive values of a **discrete** type
  - eg. enumerations values
- Syntax

```
for name in [reverse] discrete_subtype_definition loop
...
end loop;
```

- Example

```
for Day in Days_T loop
  Refresh_Planning (Day);
end loop;
```

## Variable and Sequence of Values

- Variable declared implicitly by loop statement
  - Has a view as constant
  - No assignment or update possible
- Initialized as 'First, incremented as 'Succ
- Syntactic sugar: several forms allowed

*-- All values of a type or subtype*

```
for Day in Days_T loop
```

```
for Day in Days_T range Mon .. Fri -- anonymous subtype
```

*-- Constant and variable range*

```
for Day in Mon .. Fri loop
```

```
Today, Tomorrow : Days_T;
```

...

```
for Day in Today .. Tomorrow loop
```

# Low-Level For-loop Parameter Type

- The type can be implicit
  - As long as it is clear for the compiler
  - Warning: same name can belong to several enums

```
1 procedure Main is
2   type Color_T is (Red, White, Blue);
3   type Rgb_T is (Red, Green, Blue);
4 begin
5   for Color in Red .. Blue loop -- which Red and Blue?
6     null;
7   end loop;
8   for Color in Rgb_T'(Red) .. Blue loop -- OK
9     null;
10  end loop;
```

```
main.adb:5:21: error: ambiguous bounds in range of iteration
main.adb:5:21: error: possible interpretations:
main.adb:5:21: error: type "Rgb_T" defined at line 3
main.adb:5:21: error: type "Color_T" defined at line 2
main.adb:5:21: error: ambiguous bounds in discrete range
```

- If bounds are **universal\_integer**, then type is **Integer** unless otherwise specified

```
for Idx in 1 .. 3 loop -- Idx is Integer
```

```
for Idx in Short range 1 .. 3 loop -- Idx is Short
```

# Null Ranges

- **Null range** when lower bound  $>$  upper bound
  - `1 .. 0, Fri .. Mon`
  - Literals and variables can specify null ranges
- No iteration at all (not even one)
- Shortcut for upper bound validation

```
-- Null range: loop not entered  
for Today in Fri .. Mon loop
```

## Reversing Low-Level Iteration Direction

- Keyword **reverse** reverses iteration values
  - Range must still be ascending
  - Null range still cause no iteration

```
for This_Day in reverse Mon .. Fri loop
```

## For-Loop Parameter Visibility

- Scope rules don't change
- Inner objects can hide outer objects

Block: **declare**

```
Counter : Float := 0.0;
```

**begin**

```
-- For_Loop.Counter hides Block.Counter
```

```
For_Loop : for Counter in Integer range A .. B loop
```

```
...
```

```
end loop;
```

**end**;

## Referencing Hidden Names

- Must copy for-loop parameter to some other object if needed after the loop exits
- Use dot notation with outer scope name when hiding occurs

Foo:

**declare**

```
Counter : Float := 0.0;
```

**begin**

...

```
for Counter in Integer range 1 .. Number_Read loop
```

```
    -- set declared "Counter" to loop counter
```

```
    Foo.Counter := Float (Counter);
```

...

```
end loop;
```

...

```
end Foo;
```

# Iterations Exit Statements

- Early loop exit

- Syntax

```
exit [<loop_name>] [when <condition>]
```

- No name: Loop exited **entirely**

- Not only current iteration

```
for K in 1 .. 1000 loop
  exit when K > F(K);
end loop;
```

- With name: Specified loop exited

```
for J in 1 .. 1000 loop
  Inner: for K in 1 .. 1000 loop
    exit Inner when K > F(K);
  end loop;
end loop;
```

## For-Loop with Exit Statement Example

```
-- find position of Key within Table
Found := False;
-- iterate over Table
Search : for Index in Table'Range loop
  if Table (Index) = Key then
    Found := True;
    Position := Index;
    exit Search;
  elsif Table (Index) > Key then
    -- no point in continuing
    exit Search;
  end if;
end loop Search;
```

# Quiz

```
A, B : Integer := 123;
```

Which loop block(s) is (are) legal?

- A** for A in 1 .. 10 loop  
    A := A + 1;  
end loop;
- B** for B in 1 .. 10 loop  
    Put\_Line (Integer'Image (B));  
end loop;
- C** for C in reverse 1 .. 10 loop  
    Put\_Line (Integer'Image (C));  
end loop;
- D** for D in 10 .. 1 loop  
    Put\_Line (Integer'Image (D));  
end loop;

# Quiz

A, B : **Integer** := 123;

Which loop block(s) is (are) legal?

- A** for A in 1 .. 10 loop  
    A := A + 1;  
end loop;
- B** for B in 1 .. 10 loop  
    Put\_Line (Integer'Image (B));  
end loop;
- C** for C in reverse 1 .. 10 loop  
    Put\_Line (Integer'Image (C));  
end loop;
- D** for D in 10 .. 1 loop  
    Put\_Line (Integer'Image (D));  
end loop;

Explanations

- A** Cannot assign to a loop parameter
- B** Legal - 10 iterations
- C** Legal - 10 iterations
- D** Legal - 0 iterations

## GOTO Statements

# GOTO Statements

## ■ Syntax

```
goto_statement ::= goto label;  
label ::= << identifier >>
```

## ■ Rationale

- Historic usage
- Arguably cleaner for some situations

## ■ Restrictions

- Based on common sense
- Example: cannot jump into a **case** statement

# GOTO Use

- Mostly discouraged
- May simplify control flow
- For example in-loop **continue** construct

**loop**

*-- lots of code*

...

**goto** continue;

*-- lots more code*

...

<<continue>>

**end loop;**

- As always maintainability beats hard set rules

Lab

# Statements Lab

## ■ Requirements

- Create a simple algorithm to count number of hours worked in a week
  - Use **Ada.Text\_IO.Get\_Line** to ask user for hours worked on each day
  - Any hours over 8 gets counted as 1.5 times number of hours (e.g. 10 hours worked will get counted as 11 hours towards total)
  - Saturday hours get counted at 1.5 times number of hours
  - Sunday hours get counted at 2 times number of hours
- Print total number of hours "worked"

## ■ Hints

- Use **for** loop to iterate over days of week
- Use **if** statement to determine overtime hours
- Use **case** statement to determine weekend bonus

## Statements Lab Extra Credit

- Use an inner loop when getting hours worked to check validity
  - Less than 0 should exit outer loop
  - More than 24 should not be allowed

# Statements Lab Solution

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure Main is
3   type Days_Of_Week_T is
4     (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
5   type Hours_Worked is digits 6;
6
7   Total_Worked : Hours_Worked := 0.0;
8   Hours_Today  : Hours_Worked;
9   Overtime     : Hours_Worked;
10 begin
11   Day_Loop :
12   for Day in Days_Of_Week_T loop
13     Put_Line (Day'Image);
14     Input_Loop :
15     loop
16       Hours_Today := Hours_Worked'Value (Get_Line);
17       exit Day_Loop when Hours_Today < 0.0;
18       if Hours_Today > 24.0 then
19         Put_Line ("I don't believe you");
20       else
21         exit Input_Loop;
22       end if;
23     end loop Input_Loop;
24     if Hours_Today > 8.0 then
25       Overtime := Hours_Today - 8.0;
26       Hours_Today := Hours_Today + 0.5 * Overtime;
27     end if;
28     case Day is
29     when Monday .. Friday => Total_Worked := Total_Worked + Hours_Today;
30     when Saturday      => Total_Worked := Total_Worked + Hours_Today * 1.5;
31     when Sunday        => Total_Worked := Total_Worked + Hours_Today * 2.0;
32     end case;
33   end loop Day_Loop;
34
35   Put_Line (Total_Worked'Image);
36 end Main;
```

## Summary

# Summary

- Assignments must satisfy any constraints of LHS
  - Invalid assignments don't alter target
- Intent to do nothing must be explicitly specified
- Case statements alternatives don't fall through
- Any kind of loop can be expressed with building blocks

# Array Types

## Introduction

# What Is an Array?

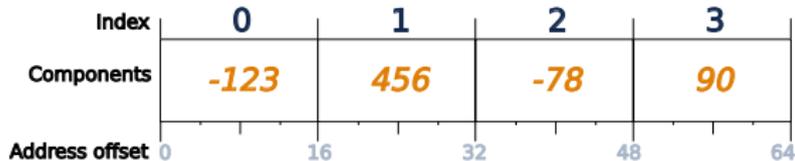
- Definition: collection of components of the same type, stored in contiguous memory, and indexed using a discrete range
- Syntax (simplified):

```
type <typename> is array (Index_Type) of Component_Type;
```

where

- *Index\_Type*
  - Discrete range of values to be used to access the array components
- *Component\_Type*
  - Type of values stored in the array
  - All components are of this same type and size

```
type Array_T is array (0 .. 3) of Interfaces.Integer_32;
```



## Arrays in Ada

- Traditional array concept supported to any dimension

**declare**

```
type Hours is digits 6;
```

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

```
type Schedule is array (Days) of Hours;
```

```
Workdays : Schedule;
```

**begin**

```
...
```

```
Workdays (Mon) := 8.5;
```

## Array Type Index Constraints

- Must be of an integer or enumeration type
- May be dynamic
- Default to predefined **Integer**
  - Same rules as for-loop parameter default type
- Allowed to be null range
  - Defines an empty array
  - Meaningful when bounds are computed at run-time
- Used to define constrained array types

```
type Schedule is array (Days range Mon .. Fri) of Float;  
type Flags_T is array (-10 .. 10) of Boolean;
```

- Or to constrain unconstrained array types

```
subtype Line is String (1 .. 80);  
subtype Translation is Matrix (1..3, 1..3);
```

## Run-Time Index Checking

- Array indices are checked at run-time as needed
- Invalid index values result in `Constraint_Error`

```
procedure Test is
  type Int_Arr is array (1..10) of Integer;
  A : Int_Arr;
  K : Integer;
begin
  A := (others => 0);
  K := FOO;
  A (K) := 42; -- run-time error if Foo returns < 1 or > 10
  Put_Line (A(K)'Image);
end Test;
```

## Kinds of Array Types

- **Constrained** Array Types
  - Bounds specified by type declaration
  - **All** objects of the type have the same bounds
- **Unconstrained** Array Types
  - Bounds not constrained by type declaration
  - Objects share the type, but not the bounds
  - More flexible

```
type Unconstrained is array (Positive range <>)
  of Integer;
```

```
U1 : Unconstrained (1 .. 10);
```

```
S1 : String (1 .. 50);
```

```
S2 : String (35 .. 95);
```

## Constrained Array Types

# Constrained Array Type Declarations

## Syntax (simplified)

```
type <typename> is array (<index constraint>) of <constrained type>;
```

where

***typename*** - identifier

***index constraint*** - discrete range or type

***constrained type*** - type with size known at compile time

## Examples

```
type Integer_Array_T is array (1 .. 3) of Integer;  
type Boolean_Array_T is array (Boolean) of Integer;  
type Character_Array_T is array (character range 'a' .. 'z') of Boolean;  
type Copycat_T is array (Boolean_Array_T'Range) of Integer;
```

# Quiz

```
type Array1_T is array (1 .. 8) of Boolean;  
type Array2_T is array (0 .. 7) of Boolean;  
X1, Y1 : Array1_T;  
X2, Y2 : Array2_T;
```

Which statement(s) is (are) legal?

- A. X1 (1) := Y1 (1);
- B. X1 := Y1;
- C. X1 (1) := X2 (1);
- D. X2 := X1;

## Quiz

```
type Array1_T is array (1 .. 8) of Boolean;  
type Array2_T is array (0 .. 7) of Boolean;  
X1, Y1 : Array1_T;  
X2, Y2 : Array2_T;
```

Which statement(s) is (are) legal?

- A. `X1 (1) := Y1 (1);`
- B. `X1 := Y1;`
- C. `X1 (1) := X2 (1);`
- D. `X2 := X1;`

Explanations

- A. Legal - components are `Boolean`
- B. Legal - object types match
- C. Legal - components are `Boolean`
- D. Although the sizes are the same and the components are the same, the type is different

## Unconstrained Array Types

# Unconstrained Array Type Declarations

- Do not specify bounds for objects
- Thus different objects of the same type may have different bounds
- Bounds cannot change once set
- Syntax (with simplifications)

```
unconstrained_array_definition ::=  
    array (index_subtype_definition  
          {, index_subtype_definition})  
    of subtype_indication  
index_subtype_definition ::= subtype_mark range <>
```

- Examples

```
type Index is range 1 .. Integer'Last;  
type Char_Arr is array (Index range <>) of Character;
```

## Supplying Index Constraints for Objects

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);  
type Schedule is array (Days range <>) of Float;
```

- Bounds set by:

- Object declaration

```
Weekdays : Schedule(Mon..Fri);
```

- Object (or constant) initialization

```
Weekend : Schedule := (Sat => 4.0, Sun => 0.0);  
-- (Note this is an array aggregate, explained later)
```

- Further type definitions (shown later)

- Actual parameter to subprogram (shown later)

- Once set, bounds never change

```
Weekdays(Sat) := 0.0; -- Constraint error  
Weekend(Mon) := 0.0; -- Constraint error
```

## Bounds Must Satisfy Type Constraints

- Must be somewhere in the range of possible values specified by the type declaration
- `Constraint_Error` otherwise

```
type Index is range 1 .. 100;
type Char_Arr is array (Index range <>) of Character;
...
Wrong : Char_Arr (0 .. 10);    -- run-time error
OK : Char_Arr (50 .. 75);
```

## Null Index Range

- When 'Last of the range is smaller than 'First
  - Array is empty - no components
- When using literals, the compiler will allow out-of-range numbers to indicate empty range
  - Provided values are within the index's base type

```
type Index_T is range 1 .. 100;  
-- Index_T'Size = 8
```

```
type Array_T is array (Index_T range <>) of Integer;
```

```
Typical_Empty_Array : Array_T (1 .. 0);  
Weird_Empty_Array   : Array_T (123 .. -5);  
Illegal_Empty_Array : Array_T (999 .. 0);
```

- When the index type is a single-valued enumerated type, no empty array is possible

# "String" Types

- Language-defined unconstrained array types
  - Allow double-quoted literals as well as aggregates
  - Always have a character component type
  - Always one-dimensional
- Language defines various types
  - **String**, with **Character** as component

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is array (Positive range <>) of Character;
```
  - **Wide\_String**, with **Wide\_Character** as component
  - **Wide\_Wide\_String**, with **Wide\_Wide\_Character** as component
    - Ada 2005 and later
- Can be defined by applications too

## Application-Defined String Types

- Like language-defined string types
  - Always have a character component type
  - Always one-dimensional
- Recall character types are enumeration types with at least one character literal value

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');  
type Roman_Number is array (Positive range <>)  
  of Roman_Digit;  
Orwellian : constant Roman_Number := "MCMLXXXIV";
```

## Specifying Constraints Via Initial Value

- Lower bound is `Index_subtype'First`
- Upper bound is taken from number of items in value

```
subtype Positive is Integer range 1 .. Integer'Last;  
type String is array (Positive range <>  
  of Character;
```

...

```
M : String := "Hello World!";  
-- M'First is Positive'First (1)
```

```
type Another_String is array (Integer range <>  
  of Character;
```

...

```
M : Another_String := "Hello World!";  
-- M'First is Integer'First
```

# Indefinite Types

- An *indefinite type* does not provide enough information to be instantiated
  - Size
  - Representation
- Unconstrained arrays types are indefinite
  - They do not have a definite 'Size
- Other indefinite types exist (seen later)

## No Indefinite Component Types

- Arrays: consecutive components of the exact **same type**
- Component size must be **defined**
  - No indefinite types
  - No unconstrained types
  - Constrained subtypes allowed

```
type Good is array (1 .. 10) of String (1 .. 20); -- OK
type Bad is array (1 .. 10) of String; -- Illegal
```

## Arrays of Arrays

- Allowed (of course!)
  - As long as the "component" array type is constrained
- Indexed using multiple parenthesized values
  - One per array

**declare**

```
type Array_of_10 is array (1..10) of Integer;
```

```
type Array_of_Array is array (Boolean) of Array_of_10;
```

```
A : Array_of_Array;
```

**begin**

```
...
```

```
A (True)(3) := 42;
```

# Quiz

```
type Bit_T is range 0 .. 1;  
type Bit_Array_T is array (Positive range <>) of Bit_T;
```

Which declaration(s) is (are)  
legal?

- A. AAA : Array\_T (0..99);
- B. BBB : Array\_T (1..32);
- C. CCC : Array\_T (17..16);
- D. DDD : Array\_T;

## Quiz

```
type Bit_T is range 0 .. 1;  
type Bit_Array_T is array (Positive range <>) of Bit_T;
```

Which declaration(s) is (are) legal?

- A. AAA : Array\_T (0..99);
- B. BBB : Array\_T (1..32);
- C. CCC : Array\_T (17..16);
- D. DDD : Array\_T;

Explanations

- A. Array\_T index is Positive which starts at 1
- B. OK, indices are in range
- C. OK, indicates a zero-length array
- D. Object must be constrained

## Attributes

# Array Attributes

- Return info about array index bounds
  - `O'Length` number of array components
  - `O'First` value of lower index bound
  - `O'Last` value of upper index bound
  - `O'Range` another way of saying `T'First .. T'Last`
- Meaningfully applied to constrained array types
  - Only constrained array types provide index bounds
  - Returns index info specified by the type (hence all such objects)
- Meaningfully applied to array objects
  - Returns index info for the object
  - Especially useful for objects of unconstrained array types

# Attributes' Benefits

- Allow code to be more robust
  - Relationships are explicit
  - Changes are localized
- Optimizer can identify redundant checks

```
declare
```

```
  type Int_Arr is array (5 .. 15) of Integer;  
  Vector : Int_Arr;
```

```
begin
```

```
  ...
```

```
  for Idx in Vector'Range loop  
    Vector (Idx) := Idx * 2;  
  end loop;
```

- Compiler understands Idx has to be a valid index for Vector, so no run-time checks are necessary

## Nth Dimension Array Attributes

- Attribute with **parameter**

T'Length (n)

T'First (n)

T'Last (n)

T'Range (n)

- n is the dimension
  - defaults to 1

```
type Two_Dimensioned is array
```

```
(1 .. 10, 12 .. 50) of T;
```

```
TD : Two_Dimensioned;
```

- TD'First (2) = 12
- TD'Last (2) = 50
- TD'Length (2) = 39
- TD'First = TD'First (1) = 1

## Quiz

```
subtype Index1_T is Integer range 0 .. 7;  
subtype Index2_T is Integer range 1 .. 8;  
type Array_T is array (Index1_T, Index2_T) of Integer;  
X : Array_T;
```

Which comparison is False?

- A.  $X'Last(2) = Index2\_T'Last$
- B.  $X'Last(1) * X'Last(2) = X'Length(1) * X'Length(2)$
- C.  $X'Length(1) = X'Length(2)$
- D.  $X'Last(1) = 7$

# Quiz

```
subtype Index1_T is Integer range 0 .. 7;  
subtype Index2_T is Integer range 1 .. 8;  
type Array_T is array (Index1_T, Index2_T) of Integer;  
X : Array_T;
```

Which comparison is False?

- A.  $X'Last(2) = Index2\_T'Last$
- B.  $X'Last(1) * X'Last(2) = X'Length(1) * X'Length(2)$
- C.  $X'Length(1) = X'Length(2)$
- D.  $X'Last(1) = 7$

Explanations

- A.  $8 = 8$
- B.  $7 * 8 \neq 8 * 8$
- C.  $8 = 8$
- D.  $7 = 7$

## Operations

# Object-Level Operations

- Assignment of array objects

```
A := B;
```

- Equality and inequality

```
if A = B then
```

- Conversions

- Component types must be the same type
- Index types must be the same or convertible
- Dimensionality must be the same
- Bounds must be compatible (not necessarily equal)

```
declare
```

```
type Index1_T is range 1 .. 2;  
type Index2_T is range 101 .. 102;  
type Array1_T is array (Index1_T) of Integer;  
type Array2_T is array (Index2_T) of Integer;  
type Array3_T is array (Boolean) of Integer;
```

```
One   : Array1_T;  
Two   : Array2_T;  
Three : Array3_T;
```

```
begin
```

```
One := Array1_T (Two);    -- OK  
Two := Array2_T (Three); -- Illegal (indices not convertible)
```

## Extra Object-Level Operations

- *Only for 1-dimensional arrays!*

- Concatenation

```
type String_Type is array
  (Integer range <>) of Character;
A : constant String_Type := "foo";
B : constant String_Type := "bar";
C : constant String_Type := A & B;
-- C now contains "foobar"
```

- Comparison (for discrete component types)

- Not for all scalars

- Logical (for **Boolean** component type)

- Slicing

- Portion of array

# Slicing

- Contiguous subsection of an array
- On any **one-dimensional** array type
  - Any component type

```
procedure Test is
  S1 : String (1 .. 9) := "Hi Adam!!";
  S2 : String := "We love    !";
begin
  S2 (9..11) := S1 (4..6);
  Put_Line (S2);
end Test;
```

Result: We love Ada!

## Example: Slicing with Explicit Indexes

- Imagine a requirement to have a ISO date
  - Year, month, and day with a specific format

**declare**

```
  Iso_Date : String (1 .. 10) := "2024-03-27";
```

**begin**

```
  Put_Line (Iso_Date);
```

```
  Put_Line (Iso_Date (1 .. 4)); -- year
```

```
  Put_Line (Iso_Date (6 .. 7)); -- month
```

```
  Put_Line (Iso_Date (9 .. 10)); -- day
```

## Idiom: Named Subtypes for Indexes

- Subtype name indicates the slice index range
  - Names for constraints, in this case index constraints
- Enhances readability and robustness

**procedure** Test **is**

```
subtype Iso_Index is Positive range 1 .. 10;  
subtype Year is Iso_Index  
  range Iso_Index'First .. Iso_Index'First + 3;  
subtype Month is Iso_Index  
  range Year'Last + 2 .. Year'Last + 3;  
subtype Day is Iso_Index  
  range Month'Last + 2 .. Month'Last + 3;  
Iso_Date : String (Iso_Index) := "2024-03-27";
```

**begin**

```
Put_Line (Iso_Date (Year));  -- 2024  
Put_Line (Iso_Date (Month)); -- 03  
Put_Line (Iso_Date (Day));  -- 27
```

## Dynamic Subtype Constraint Example

- Useful when constraints not known at compile-time
- Example: remove file name extension

File\_Name

```
(File_Name'First
```

```
..
```

```
Index (File_Name, '.', Direction => Backward));
```

# Quiz

```
type Index_T is range 1 .. 10;
type OneD_T is array (Index_T) of Boolean;
type TwoD_T is array (Index_T) of OneD_T;
A : TwoD_T;
B : OneD_T;
```

Which statement(s) is (are) legal?

- A. B(1) := A(1,2) or A(4,3);
- B. B := A(2) and A(4);
- C. A(1..2)(4) := A(5..6)(8);
- D. B(3..4) := B(4..5)

# Quiz

```
type Index_T is range 1 .. 10;
type OneD_T is array (Index_T) of Boolean;
type TwoD_T is array (Index_T) of OneD_T;
A : TwoD_T;
B : OneD_T;
```

Which statement(s) is (are) legal?

- A.** `B(1) := A(1,2) or A(4,3);`
- B.** `B := A(2) and A(4);`
- C.** `A(1..2)(4) := A(5..6)(8);`
- D.** `B(3..4) := B(4..5)`

Explanations

- A.** All objects are just Boolean values
- B.** A component of A is the same type as B
- C.** Slice must be of outermost array
- D.** Slicing allowed on single-dimension arrays

## Looping Over Array Components

## Note on Default Initialization for Array Types

- In Ada, objects are not initialized by default
- To initialize an array, you can initialize each component
  - But if the array type is used in multiple places, it would be better to initialize at the type level
  - No matter how many dimensions, there is only one component type
- Uses aspect **Default\_Component\_Value**

```
type Vector is array (Positive range <>) of Float  
  with Default_Component_Value => 0.0;
```

- Note that creating a large object of type Vector might incur a run-time cost during initialization

## Two High-Level For-Loop Kinds

- For arrays and containers
  - Arrays of any type and form
  - Iterable containers
    - Those that define iteration (most do)
    - Not all containers are iterable (e.g., priority queues)!
- For iterator objects
  - Known as "generalized iterators"
  - Language-defined, e.g., most container data structures
- User-defined iterators too
- We focus on the arrays/containers form for now

## Array/Container For-Loops

- Work in terms of components within an object
- Syntax hides indexing/iterator controls

```
for name of [reverse] array_or_container_object loop  
  ...  
end loop;
```

- Starts with "first" component unless you reverse it
- Loop parameter name is a constant if iterating over a constant, a variable otherwise

## Array Component For-Loop Example

- Given an array

```
type T is array (Positive range <>) of Integer;  
Primes : T := (2, 3, 5, 7, 11);
```

- Component-based looping would look like

```
for P of Primes loop  
    Put_Line (Integer'Image (P));  
end loop;
```

- While index-based looping would look like

```
for P in Primes'Range loop  
    Put_Line (Integer'Image (Primes (P)));  
end loop;
```

# Quiz

```
declare
  type Array_T is array (1..5) of Integer
    with Default_Component_Value => 1;
  A : Array_T;
begin
  for I in A'First + 1 .. A'Last - 1 loop
    A (I) := I * A'Length;
  end loop;
  for I of reverse A loop
    Put (I'Image);
  end loop;
end;
```

Which output is correct?

- A. 1 10 15 20 1
- B. 1 20 15 10 1
- C. 0 10 15 20 0
- D. 25 20 15 10 5

NB: Without `Default_Component_Value`, init. values are random

# Quiz

```
declare
  type Array_T is array (1..5) of Integer
    with Default_Component_Value => 1;
  A : Array_T;
begin
  for I in A'First + 1 .. A'Last - 1 loop
    A (I) := I * A'Length;
  end loop;
  for I of reverse A loop
    Put (I'Image);
  end loop;
end;
```

Which output is correct?

- A. 1 10 15 20 1
- B. **1 20 15 10 1**
- C. 0 10 15 20 0
- D. 25 20 15 10 5

Explanations

- A. There is a **reverse**
- B. Yes
- C. Default value is 1
- D. No

NB: Without `Default_Component_Value`, init. values are random

## Aggregates

# Aggregates

- Literals for composite types
  - Array types
  - Record types
- Two distinct forms
  - Positional
  - Named
- Syntax (simplified):

```
component_expr ::=  
  expression -- Defined value  
  | <>      -- Default value
```

```
array_aggregate ::= (  
  {component_expr ,} -- Positional  
  | {discrete_choice_list => component_expr ,}) -- Named  
  -- Default "others" indices  
  [others => expression]
```

## Aggregate "Positional" Form

- Specifies array component values explicitly
- Uses implicit ascending index values

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

```
type Working is array (Days) of Boolean;
```

```
Week : Working;
```

```
...
```

```
-- Saturday and Sunday are False, everything else true
```

```
Week := (True, True, True, True, True, False, False);
```

## Aggregate "Named" Form

- Explicitly specifies both index and corresponding component values
- Allows any order to be specified
- Ranges and choice lists are allowed (like case choices)

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

```
type Working is array (Days) of Boolean;
```

```
Week : Working;
```

```
...
```

```
Week := (Sat => False, Sun => False, Mon..Fri => True);
```

```
Week := (Sat | Sun => False, Mon..Fri => True);
```

## Combined Aggregate Forms Not Allowed

- Some cases lead to ambiguity, therefore never allowed for array types
- Are only allowed for record types (shown in subsequent section)

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Working is array (Days) of Boolean;
Week : Working;
...
Week := (True, True, True, True, True, False, False);
Week := (Sat => False, Sun => False, Mon..Fri => True);
Week := (True, True, True, True, True,
         Sat => False, Sun => False); -- invalid
Week := (Sat | Sun => False, Mon..Fri => True);
```

## Aggregates Are True Literal Values

- Used any place a value of the type may be used

```
type Schedule is array (Mon .. Fri) of Float;  
Work : Schedule;  
Normal : constant Schedule := (8.0, 8.0, 8.0, 8.0, 8.0);  
...  
Work := (8.5, 8.5, 8.5, 8.5, 6.0);  
...  
if Work = Normal then  
...  
if Work = (10.0, 10.0, 10.0, 10.0, 0.0) then -- 4-day week
```

# Aggregate Consistency Rules

- Must always be complete
  - They are literals, after all
  - Each component must be given a value
  - But defaults are possible (more in a moment)
- Must provide only one value per index position
  - Duplicates are detected at compile-time
- Compiler rejects incomplete or inconsistent aggregates

```
Week := (Sat => False,  
         Sun => False,  
         Mon .. Fri => True,  
         Wed => False);
```

## "Others"

- Indicates all components not yet assigned a value
- All remaining components get this single value
- Similar to case statement's **others**
- Can be used to apply defaults too

```
type Schedule is array (Days) of Float;
```

```
Work : Schedule;
```

```
Normal : constant Schedule := (8.0, 8.0, 8.0, 8.0, 8.0,  
                                others => 0.0);
```

## Nested Aggregates

- For arrays of composite component types

```
type Col_T is array (1 .. 3) of Float;  
type Matrix_T is array (1 .. 3) of Col_T;  
Matrix : Matrix_T := (1 => (1.2, 1.3, 1.4),  
                      2 => (2.5, 2.6, 2.7),  
                      3 => (3.8, 3.9, 3.0));
```

## Defaults Within Array Aggregates

- Specified via the box notation
- Value for component is thus taken as for stand-alone object declaration
  - So there may or may not be a defined default!
- Can only be used with "named association" form
  - But **others** counts as named form
- Syntax

```
discrete_choice_list => <>
```

- Example

```
type Int_Arr is array (1 .. N) of Integer;  
Primes : Int_Arr := (1 => 2, 2 .. N => <>);
```

## Named Format Aggregate Rules

- Bounds cannot overlap
  - Index values must be specified once and only once
- All bounds must be static
  - Avoids run-time cost to verify coverage of all index values
  - Except for single choice format

```
type Float_Arr is array (Integer range <>) of Float;  
Ages : Float_Arr (1 .. 10) := (1 .. 3 => X, 4 .. 10 => Y);  
-- illegal: 3 and 4 appear twice  
Overlap : Float_Arr (1 .. 10) := (1 .. 4 => X, 3 .. 10 => Y);  
N, M, K, L : Integer;  
-- illegal: cannot determine if  
-- every index covered at compile time  
Not_Static : Float_Arr (1 .. 10) := (M .. N => X, K .. L => Y);  
-- This is legal  
Values : Float_Arr (1 .. N) := (1 .. N => X);
```

## Quiz

```
type Array_T is array (1 .. 5) of Integer;  
X : Array_T;  
J : Integer := X'First;
```

Which statement is correct?

- A. X := (1, 2, 3, 4 => 4, 5 => 5);
- B. X := (1..3 => 100, 4..5 => -100, others => -1);
- C. X := (J => -1, J + 1..X'Last => 1);
- D. X := (1..3 => 100, 3..5 => 200);

## Quiz

```
type Array_T is array (1 .. 5) of Integer;  
X : Array_T;  
J : Integer := X'First;
```

Which statement is correct?

- A. `X := (1, 2, 3, 4 => 4, 5 => 5);`
- B. `X := (1..3 => 100, 4..5 => -100, others => -1);`
- C. `X := (J => -1, J + 1..X'Last => 1);`
- D. `X := (1..3 => 100, 3..5 => 200);`

## Explanations

- A. Cannot mix positional and named notation
- B. Correct - others not needed but is allowed
- C. Dynamic values must be the only choice. (This could be fixed by making J a constant.)
- D. Overlapping index values (3 appears more than once)

# Aggregates in Ada 2022

Ada 2022

- Ada 2022 allows us to use square brackets "[...]" in defining aggregates

```
type Array_T is array (positive range <>) of Integer;
```

- So common aggregates can use either square brackets or parentheses

```
Ada2012 : Array_T := (1, 2, 3);  
Ada2022 : Array_T := [1, 2, 3];
```

- But square brackets help in more problematic situations

- Empty array

```
Ada2012 : Array_T := (1..0 => 0);  
Illegal  : Array_T := ();  
Ada2022  : Array_T := [];
```

- Single component array

```
Ada2012 : Array_T := (1 => 5);  
Illegal  : Array_T := (5);  
Ada2022 : Array_T := [5];
```

# Iterated Component Association

Ada 2022

- With Ada 2022, we can create aggregates with `iterators`
  - Basically, an inline looping mechanism
- Index-based iterator

```
type Array_T is array (positive range <>) of Integer;  
Object1 : Array_T(1..5) := (for J in 1 .. 5 => J * 2);  
Object2 : Array_T(1..5) := (for J in 2 .. 3 => J,  
                           5 => -1,  
                           others => 0);
```

- Object1 will get initialized to the squares of 1 to 5
  - Object2 will give the equivalent of (0, 2, 3, 0, -1)
- Component-based iterator

```
Object2 := [for Item of Object => Item * 2];
```

- Object2 will have each component doubled

# More Information on Iterators

Ada 2022

- You can nest iterators for arrays of arrays

```
type Col_T is array (1 .. 3) of Integer;
type Matrix_T is array (1 .. 3) of Col_T;
Matrix : Matrix_T :=
  [for J in 1 .. 3 =>
    [for K in 1 .. 3 => J * 10 + K]];
```

- You can even use multiple iterators for a single dimension array

```
Ada2012 : Array_T(1..5) :=
  [for I in 1 .. 2 => -1,
   for J in 4 ..5 => 1,
   others => 0];
```

- Restrictions

- You cannot mix index-based iterators and component-based iterators in the same aggregate
- You still cannot have overlaps or missing values

# Delta Aggregates

Ada 2022

```
type Coordinate_T is array (1 .. 3) of Float;  
Location : constant Coordinate_T := (1.0, 2.0, 3.0);
```

- Sometimes you want to copy an array with minor modifications
  - Prior to Ada 2022, it would require two steps

```
declare  
  New_Location : Coordinate_T := Location;  
begin  
  New_Location(3) := 0.0;  
  -- OR  
  New_Location := (3 => 0.0, others => <>);  
end;
```

- Ada 2022 introduces a **delta aggregate**
  - Aggregate indicates an object plus the values changed - the *delta*  

```
New_Location : Coordinate_T := [Location with delta 3 => 0.0];
```
- Notes
  - You can use square brackets or parentheses
  - Only allowed for single dimension arrays

*This works for records as well (see that chapter)*

## Detour - 'Image for Complex Types

# 'Image Attribute

Ada 2022

- Previously, we saw the string attribute 'Image is provided for scalar types
  - e.g. `Integer'Image(10+2)` produces the string `" 12"`
- Starting with Ada 2022, the Image attribute can be used for any type

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
  type Colors_T is (Red, Yellow, Green);
  type Array_T is array (Colors_T) of Boolean;
  Object : Array_T :=
    (Green => False,
     Yellow => True,
     Red    => True);
begin
  Put_Line (Object'Image);
end Main;
```

Yields an output of

```
[TRUE, TRUE, FALSE]
```

# Overriding the 'Image Attribute

Ada 2022

- We don't always want to rely on the compiler defining how we print a complex object
- We can define it - by using 'Image and attaching a procedure to the Put\_Image aspect

```
type Colors_T is (Red, Yellow, Green);  
type Array_T is array (Colors_T) of Boolean with  
  Put_Image => Array_T_Image;
```

# Defining the 'Image Attribute

Ada 2022

- Then we need to declare the procedure

```
procedure Array_T_Image  
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;  
   Value  :      Array_T);
```

- Which uses the  
Ada.Strings.Text\_Buffers.Root\_Buffer\_Type as an output  
buffer
- (No need to go into detail here other than knowing you do  
Output.Put to add to the buffer)

- And then we define it

```
procedure Array_T_Image  
  (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;  
   Value  :      Array_T) is  
begin  
  for Color in Value'Range loop  
    Output.Put (Color'Image & "=>" & Value (Color)'Image & ASCII.LF);  
  end loop;  
end Array_T_Image;
```

# Using the 'Image Attribute

Ada 2022

- Now, when we call Image we get our "pretty-print" version

```
with Ada.Text_IO; use Ada.Text_IO;
with Types; use Types;
procedure Main is
  Object : Array_T := (Green => False,
                       Yellow => True,
                       Red    => True);

begin
  Put_Line (Object'Image);
end Main;
```

- Generating the following output

```
RED=>TRUE
```

```
YELLOW=>TRUE
```

```
GREEN=>FALSE
```

- Note this redefinition can be used on any type, even the scalars that have always had the attribute

## Anonymous Array Types

## Anonymous Array Types

- Array objects need not be of a named type  
A : **array** (1 .. 3) **of** B;
- Without a type name, no object-level operations
  - Cannot be checked for type compatibility
  - Operations on components are still ok if compatible

```
declare
```

```
-- These are not same type!
```

```
  A, B : array (Foo) of Bar;
```

```
begin
```

```
  A := B;  -- illegal
```

```
  B := A;  -- illegal
```

```
  -- legal assignment of value
```

```
  A(J) := B(K);
```

```
end;
```

Lab

# Array Lab

## ■ Requirements

- Create an array type whose index is days of the week and each component is a number
- Create two objects of the array type, one of which is constant
- Perform the following operations
  - Copy the constant object to the non-constant object
  - Print the contents of the non-constant object
  - Use an array aggregate to initialize the non-constant object
  - For each component of the array, print the array index and the value
  - Move part ("source") of the non-constant object to another part ("destination"), and then clear the source location
  - Print the contents of the non-constant object

## ■ Hints

- When you want to combine multiple strings (which are arrays!) use the concatenation operator (**&**)
- Slices are how you access part of an array
- Use aggregates (either named or positional) to initialize data

# Arrays of Arrays

## ■ Requirements

- For each day of the week, you need an array of three strings containing names of workers for that day
- Two sets of workers: weekend and weekday, but the store is closed on Wednesday (no workers)
- Initialize the array and then print it hierarchically

## Array Lab Solution - Declarations

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Main is
3
4      type Days_Of_Week_T is
5          (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
6      type Unconstrained_Array_T is
7          array (Days_Of_Week_T range <>) of Natural;
8
9      Const_Arr : constant Unconstrained_Array_T := (1, 2, 3, 4);
10     Array_Var : Unconstrained_Array_T (Days_Of_Week_T);
11
12     type Name_T is array (1 .. 6) of Character;
13     type Names_T is array (1 .. 3) of Name_T;
14     Weekly_Staff : array (Days_Of_Week_T) of Names_T;
```

# Array Lab Solution - Implementation

```
15 begin
16   Array_Var := Const_Arr;
17   for Item of Array_Var loop
18     Put_Line (Item'Image);
19   end loop;
20   New_Line;
21
22   Array_Var :=
23     (Mon => 111, Tue => 222, Wed => 333, Thu => 444, Fri => 555, Sat => 666,
24     Sun => 777);
25   for Index in Array_Var'Range loop
26     Put_Line (Index'Image & " => " & Array_Var (Index)'Image);
27   end loop;
28   New_Line;
29
30   Array_Var (Mon .. Wed) := Const_Arr (Wed .. Fri);
31   Array_Var (Wed .. Fri) := (others => Natural'First);
32   for Item of Array_Var loop
33     Put_Line (Item'Image);
34   end loop;
35   New_Line;
36
37   Weekly_Staff := (Mon | Tue | Thu | Fri => ("Fred ", "Barney", "Wilma "),
38   Wed => ("closed", "closed", "closed"),
39   others => ("Pinky ", "Inky ", "Blinky"));
40
41   for Day in Weekly_Staff'Range loop
42     Put_Line (Day'Image);
43     for Staff of Weekly_Staff(Day) loop
44       Put_Line (" " & String (Staff));
45     end loop;
46   end loop;
47 end Main;
```

## Summary

## Final Notes on Type **String**

- Any single-dimensional array of some character type is a *string type*
  - Language defines types **String**, **Wide\_String**, etc.
- Just another array type: no null termination
- Language-defined support defined in Appendix A
  - **Ada.Strings.\***
  - Fixed-length, bounded-length, and unbounded-length
  - Searches for pattern strings and for characters in program-specified sets
  - Transformation (replacing, inserting, overwriting, and deleting of substrings)
  - Translation (via a character-to-character mapping)

# Summary

- Any dimensionality directly supported
- Component types can be any (constrained) type
- Index types can be any discrete type
  - Integer types
  - Enumeration types
- Constrained array types specify bounds for all objects
- Unconstrained array types leave bounds to the objects
  - Thus differently-sized objects of the same type
- Default initialization for large arrays may be expensive!
- Anonymously-typed array objects used in examples for brevity but that doesn't mean you should in real programs

# Record Types

## Introduction

# Syntax and Examples

## ■ Syntax (simplified)

```
type T is record
  Component_Name : Type [:= Default_Value];
  ...
end record;
```

```
type T_Empty is null record;
```

## ■ Example

```
type Record1_T is record
  Component1 : Integer;
  Component2 : Boolean;
end record;
```

## ■ Records can be **discriminated** as well

```
type T (Size : Natural := 0) is record
  Text : String (1 .. Size);
end record;
```

## Components Rules

# Characteristics of Components

- **Heterogeneous** types allowed
- Referenced **by name**
- May be no components, for **empty records**
- **No** anonymous types (e.g., arrays) allowed

```
type Record_1 is record
  This_Is_Not_Legal : array (1 .. 3) of Integer;
end record;
```

- **No** constant components

```
type Record_2 is record
  This_Is_Not_Legal : constant Integer := 123;
end record;
```

- **No** recursive definitions

```
type Record_3 is record
  This_Is_Not_Legal : Record_3;
end record;
```

- **No** indefinite types

```
type Record_5 is record
  This_Is_Not_Legal : String;
  But_This_Is_Legal : String (1 .. 10);
end record;
```

## Multiple Declarations

- Multiple declarations are allowed (like objects)

```
type Several is record
  A, B, C : Integer := F;
end record;
```

- Equivalent to

```
type Several is record
  A : Integer := F;
  B : Integer := F;
  C : Integer := F;
end record;
```

## "Dot" Notation for Components Reference

```
type Months_T is (January, February, ..., December);
type Date is record
    Day : Integer range 1 .. 31;
    Month : Months_T;
    Year : Integer range 0 .. 2099;
end record;
Arrival : Date;
...
Arrival.Day := 27;  -- components referenced by name
Arrival.Month := November;
Arrival.Year := 1990;
```

- Can reference nested components

Employee

```
.Birth_Date
    .Month := March;
```

# Quiz

```
type Record_T is record
    -- Definition here
end record;
```

Which record definition(s) is (are) legal?

- A. Component\_1 : array (1 .. 3) of Boolean
- B. Component\_2, Component\_3 : Integer
- C. Component\_1 : Record\_T
- D. Component\_1 : constant Integer := 123

## Quiz

```
type Record_T is record
    -- Definition here
end record;
```

Which record definition(s) is (are) legal?

- A. Component\_1 : array (1 .. 3) of Boolean
  - B. *Component\_2, Component\_3 : Integer*
  - C. Component\_1 : Record\_T
  - D. Component\_1 : constant Integer := 123
- 
- A. Anonymous types not allowed
  - B. Correct
  - C. No recursive definition
  - D. No constant component

# Quiz

```
type Cell is record
  Val : Integer;
  Message : String;
end record;
```

Is the definition legal?

- A. Yes
- B. No

# Quiz

```
type Cell is record
  Val : Integer;
  Message : String;
end record;
```

Is the definition legal?

- A. Yes
- B. **No**

A **record** definition cannot have a component of an indefinite type. **String** is indefinite if you don't specify its size.

## Operations

# Available Operations

- Predefined
  - Equality (and thus inequality)  
`if A = B then`
  - Assignment  
`A := B;`
- User-defined
  - Subprograms

# Assignment Examples

```
declare
  type Complex is record
    Real : Float;
    Imaginary : Float;
  end record;
  ...
  Phase1 : Complex;
  Phase2 : Complex;
begin
  ...
  -- object reference
  Phase1 := Phase2; -- entire object reference
  -- component references
  Phase1.Real := 2.5;
  Phase1.Real := Phase2.Real;
end;
```

## Limited Types - Quick Intro

- A **record** type can be limited
  - And some other types, described later
- **limited** types cannot be **copied** or **compared**
  - As a result then cannot be assigned
  - May still be modified component-wise

```
type Lim is limited record
  A, B : Integer;
end record;
```

```
L1, L2 : Lim := Create_Lim (1, 2); -- Initial value OK
```

```
L1 := L2; -- Illegal
if L1 /= L2 then -- Illegal
[...]
```

# Aggregates

# Aggregates

- Literal values for composite types
  - As for arrays
  - Default value / selector: `<>`, **others**
- Can use both **named** and **positional**
  - Unambiguous
- Example:

```
(Pos_1_Value,  
Pos_2_Value,  
Component_3 => Pos_3_Value,  
Component_4 => <>, -- Default value (Ada 2005)  
others => Remaining_Value)
```

# Record Aggregate Examples

```
type Color_T is (Red);
type Car_T is record
    Color      : Color_T;
    Plate_No   : String (1 .. 6);
    Year       : Natural;
end record;
type Complex_T is record
    Real       : Float;
    Imaginary  : Float;
end record;

declare
    Car      : Car_T      := (Red, "ABC123", Year => 2_022);
    Phase   : Complex_T := (1.2, 3.4);
begin
    Phase := (Real => 5.6, Imaginary => 7.8);
end;
```

## Aggregate Completeness

- All component values must be accounted for
  - Including defaults via box
- Allows compiler to check for missed components
- Type definition

```
type Struct is record
```

```
  A : Integer;
```

```
  B : Integer;
```

```
  C : Integer;
```

```
  D : Integer;
```

```
end record;
```

```
S : Struct;
```

- Compiler will not catch the missing component

```
S.A := 10;
```

```
S.B := 20;
```

```
S.C := 12;
```

```
Send (S);
```

- Aggregate must be complete  
- compiler error

```
S := (10, 20, 12);
```

```
Send (S);
```

# Named Associations

- **Any** order of associations
- Provides more information to the reader
  - Can mix with positional
- Restriction
  - Must stick with named associations **once started**

```
type Complex is record
```

```
  Real : Float;
```

```
  Imaginary : Float;
```

```
end record;
```

```
Phase : Complex := (0.0, 0.0);
```

```
...
```

```
Phase := (10.0, Imaginary => 2.5);
```

```
Phase := (Imaginary => 12.5, Real => 0.212);
```

```
Phase := (Imaginary => 12.5, 0.212); -- illegal
```

## Nested Aggregates

```
type Months_T is (January, February, ..., December);
type Date is record
    Day    : Integer range 1 .. 31;
    Month  : Months_T;
    Year   : Integer range 0 .. 2099;
end record;
type Person is record
    Born   : Date;
    Hair   : Color;
end record;
John : Person    := ((21, November, 1990), Brown);
Julius : Person := ((2, August, 1995), Blond);
Heather : Person := ((2, March, 1989), Hair => Blond);
Megan : Person  := (Hair => Blond,
                    Born => (16, December, 2001));
```

## Aggregates with Only One Component

- **Must** use named form
- Same reason as array aggregates

```
type Singular is record  
  A : Integer;  
end record;
```

```
S : Singular := (3);           -- illegal  
S : Singular := (3 + 1);      -- illegal  
S : Singular := (A => 3 + 1); -- required
```

## Aggregates with **others**

- Indicates all components not yet specified (like arrays)
- All **others** get the same value
  - They must be the **exact same** type

```
type Poly is record
```

```
  A : Float;
```

```
  B, C, D : Integer;
```

```
end record;
```

```
P : Poly := (2.5, 3, others => 0);
```

```
type Homogeneous is record
```

```
  A, B, C : Integer;
```

```
end record;
```

```
Q : Homogeneous := (others => 10);
```

# Quiz

What is the result of building and running this code?

```
procedure Main is
  type Record_T is record
    A, B, C : Integer;
  end record;

  V : Record_T := (A => 1);
begin
  Put_Line (Integer'Image (V.A));
end Main;
```

- A. 0
- B. 1
- C. Compilation error
- D. Run-time error

# Quiz

What is the result of building and running this code?

```
procedure Main is
  type Record_T is record
    A, B, C : Integer;
  end record;

  V : Record_T := (A => 1);
begin
  Put_Line (Integer'Image (V.A));
end Main;
```

- A. 0
- B. 1
- C. **Compilation error**
- D. Run-time error

The aggregate is incomplete. The aggregate must specify all components. You could use box notation (A => 1, **others** => <>)

# Quiz

What is the result of building and running this code?

```
procedure Main is
  type My_Integer is new Integer;
  type Record_T is record
    A, B, C : Integer;
    D : My_Integer;
  end record;

  V : Record_T := (others => 1);
begin
  Put_Line (Integer'Image (V.A));
end Main;
```

- A. 0
- B. 1
- C. Compilation error
- D. Run-time error

# Quiz

What is the result of building and running this code?

```
procedure Main is
  type My_Integer is new Integer;
  type Record_T is record
    A, B, C : Integer;
    D : My_Integer;
  end record;

  V : Record_T := (others => 1);
begin
  Put_Line (Integer'Image (V.A));
end Main;
```

- A. 0
- B. 1
- C. *Compilation error*
- D. Run-time error

All components associated to a value using **others** must be of the same **type**.

## Quiz

```
type Nested_T is record
  Component : Integer;
end record;
type Record_T is record
  One      : Integer;
  Two      : Character;
  Three    : Integer;
  Four     : Nested_T;
end record;
X, Y : Record_T;
Z     : constant Nested_T := (others => -1);
```

Which assignment(s) is (are) legal?

- A. X := (1, '2', Three => 3, Four => (6))
- B. X := (Two => '2', Four => Z, others => 5)
- C. X := Y
- D. X := (1, '2', 4, (others => 5))

## Quiz

```
type Nested_T is record
  Component : Integer;
end record;
type Record_T is record
  One      : Integer;
  Two      : Character;
  Three    : Integer;
  Four     : Nested_T;
end record;
X, Y : Record_T;
Z     : constant Nested_T := (others => -1);
```

Which assignment(s) is (are) legal?

- A. `X := (1, '2', Three => 3, Four => (6))`
  - B. `X := (Two => '2', Four => Z, others => 5)`
  - C. `X := Y`
  - D. `X := (1, '2', 4, (others => 5))`
- 
- A. Four **must** use named association
  - B. **others** valid: One and Three are **Integer**
  - C. Valid but Two is not initialized
  - D. Positional for all components

# Delta Aggregates

Ada 2022

- A Record can use a `delta aggregate` just like an array

```
type Coordinate_T is record
  X, Y, Z : Float;
end record;
Location : constant Coordinate_T := (1.0, 2.0, 3.0);
```

- Prior to Ada 2022, you would copy and then modify

```
declare
  New_Location : Coordinate_T := Location;
begin
  New_Location.Z := 0.0;
  -- OR
  New_Location := (Z => 0.0, others => <>);
end;
```

- Now in Ada 2022 we can just specify the change during the copy

```
New_Location : Coordinate_T := (Location with delta Z => 0.0);
```

*Note for record delta aggregates you must use named notation*

## Default Values

## Component Default Values

```
type Complex is
  record
    Real : Float := 0.0;
    Imaginary : Float := 0.0;
  end record;
-- all components use defaults
Phasor : Complex;
-- all components must be specified
I : constant Complex := (0.0, 1.0);
```

## Default Component Value Evaluation

- Occurs when object is elaborated
  - Not when the type is elaborated
- Not evaluated if explicitly overridden

```
type Structure is
  record
    A : Integer;
    R : Time := Clock;
  end record;
-- Clock is called for S1
S1 : Structure;
-- Clock is not called for S2
S2 : Structure := (A => 0, R => Yesterday);
```

## Defaults Within Record Aggregates

- Specified via the `box` notation
- Value for the component is thus taken as for a stand-alone object declaration
  - So there may or may not be a defined default!
- Can only be used with "named association" form
  - But can mix forms, unlike array aggregates

```
type Complex is
  record
    Real : Float := 0.0;
    Imaginary : Float := 0.0;
  end record;
Phase := (42.0, Imaginary => <>);
```

## Default Initialization Via Aspect Clause

- Not definable for entire record type
- Components of scalar types take type's default if no explicit default value specified by record type

```
type Toggle_Switch is (Off, On)
  with Default_Value => Off;
type Controller is record
  -- Off unless specified during object initialization
  Override : Toggle_Switch;
  -- default for this component
  Enable : Toggle_Switch := On;
end record;
C : Controller; -- Override => off, Enable => On
D : Controller := (On, Off); -- All defaults replaced
```

# Quiz

```
function Next return Natural; -- returns next number starting with 1
```

```
type Record_T is record
```

```
  A, B : Integer := Next;
```

```
  C    : Integer := Next;
```

```
end record;
```

```
R : Record_T := (C => 100, others => <>);
```

What is the value of R?

- A. (1, 2, 3)
- B. (1, 1, 100)
- C. (1, 2, 100)
- D. (100, 101, 102)

# Quiz

```
function Next return Natural; -- returns next number starting with 1
```

```
type Record_T is record
```

```
  A, B : Integer := Next;
```

```
  C    : Integer := Next;
```

```
end record;
```

```
R : Record_T := (C => 100, others => <>);
```

What is the value of R?

- A. (1, 2, 3)
- B. (1, 1, 100)
- C. (1, 2, 100)
- D. (100, 101, 102)

Explanations

- A. C => 100
- B. Multiple declaration calls Next twice
- C. Correct
- D. C => 100 has no effect on A and B

## Variant Records

# Variant Record Types

- *Variant record* can use a **discriminant** to specify alternative lists of components
  - Also called *discriminated record* type
  - Different **objects** may have **different** components
  - All objects **still** share the same type
- Kind of *storage overlay*
  - Similar to **union** in C
  - But preserves **type checking**
  - And object size **is related to** discriminant
- Aggregate assignment is allowed

# Immutable Variant Record

- Discriminant must be set at creation time and cannot be modified

```
2 type Person_Group is (Student, Faculty);
3 type Person (Group : Person_Group) is
4 record
5     -- Components common across all discriminants
6     -- (must appear before variant part)
7     Age : Positive;
8     case Group is -- Variant part of record
9         when Student => -- 1st variant
10            Gpa : Float range 0.0 .. 4.0;
11            when Faculty => -- 2nd variant
12                Pubs : Positive;
13     end case;
14 end record;
```

- In a variant record, a discriminant can be used to specify the **variant part** (line 8)
  - Similar to case statements (all values must be covered)
  - Components listed will only be visible if choice matches discriminant
  - Component names need to be unique (even across discriminants)
  - Variant part must be end of record (hence only one variant part allowed)
- Discriminant is treated as any other component
  - But is a constant in an immutable variant record

Note that discriminants can be used for other purposes than the variant part

# Immutable Variant Record Example

- Each object of `Person` has three components, but it depends on `Group`

```
Pat : Person (Student);  
Sam : Person := (Faculty, 33, 5);
```

- Pat has `Group`, `Age`, and `Gpa`
  - Sam has `Group`, `Age`, and `Pubs`
  - Aggregate specifies all components, including the discriminant
- Compiler can detect some problems, but more often clashes are run-time errors

```
procedure Do_Something (Param : in out Person) is  
begin  
  Param.Age := Param.Age + 1;  
  Param.Pubs := Param.Pubs + 1;  
end Do_Something;
```

- `Pat.Pubs := 3;` would generate a compiler warning because compiler knows `Pat` is a `Student`
    - warning: `Constraint_Error` will be raised at run time
  - `Do_Something (Pat);` generates a run-time error, because only at runtime is the discriminant for `Param` known
    - raised `CONSTRAINT_ERROR : discriminant check failed`
- `Pat := Sam;` would be a compiler warning because the constraints do not match

# Mutable Variant Record

- Type will become **mutable** if its discriminant has a *default value* **and** we instantiate the object without specifying a discriminant

```
2 type Person_Group is (Student, Faculty);
3 type Person (Group : Person_Group := Student) is -- default value
4 record
5     Age : Positive;
6     case Group is
7         when Student =>
8             Gpa : Float range 0.0 .. 4.0;
9         when Faculty =>
10            Pubs : Positive;
11     end case;
12 end record;
```

- Pat : Person; is **mutable**
- Sam : Person (Faculty); is **not mutable**
  - Declaring an object with an **explicit** discriminant value (Faculty) makes it immutable

## Mutable Variant Record Example

- Each object of `Person` has three components, but it depends on `Group`

```
Pat : Person := (Student, 19, 3.9);  
Sam : Person (Faculty);
```

- You can only change the discriminant of `Pat`, but only via a whole record assignment, e.g:

```
if Pat.Group = Student then  
  Pat := (Faculty, Pat.Age, 1);  
else  
  Pat := Sam;  
end if;  
Update (Pat);
```

- But you cannot change the discriminant of `Sam`
  - `Sam := Pat;` will give you a run-time error if `Pat.Group` is not `Faculty`
    - And the compiler will not warn about this!

# Quiz

```
type Variant_T (Sign : Integer) is record
  case Sign is
  when Integer'First .. -1 =>
    I : Integer;
    B : Boolean;
  when others =>
    N : Natural;
  end case;
end record;
```

Variant\_Object : Variant\_T (1);

Which component(s) does Variant\_Object contain?

- A. Variant\_Object.I, Variant\_Object.B
- B. Variant\_Object.N
- C. None: Compilation error
- D. None: Run-time error

# Quiz

```
type Variant_T (Sign : Integer) is record
  case Sign is
  when Integer'First .. -1 =>
    I : Integer;
    B : Boolean;
  when others =>
    N : Natural;
  end case;
end record;
```

Variant\_Object : Variant\_T (1);

Which component(s) does Variant\_Object contain?

- A. Variant\_Object.I, Variant\_Object.B
- B. *Variant\_Object.N*
- C. None: Compilation error
- D. None: Run-time error

# Quiz

```
type Variant_T (Floating : Boolean := False) is record
  case Floating is
    when False =>
      I : Integer;
    when True =>
      F : Float;
  end case;
  Flag : Character;
end record;
```

Variant\_Object : Variant\_T (True);

Which component does Variant\_Object contain?

- A. Variant\_Object.F, Variant\_Object.Flag
- B. Variant\_Object.F
- C. None: Compilation error
- D. None: Run-time error

# Quiz

```
type Variant_T (Floating : Boolean := False) is record
  case Floating is
    when False =>
      I : Integer;
    when True =>
      F : Float;
  end case;
  Flag : Character;
end record;
```

Variant\_Object : Variant\_T (True);

Which component does Variant\_Object contain?

- A. Variant\_Object.F, Variant\_Object.Flag
- B. Variant\_Object.F
- C. **None: Compilation error**
- D. None: Run-time error

The variant part cannot be followed by a component declaration (Flag : Character here)

Lab

# Record Types Lab

## ■ Requirements

- Create a simple First-In/First-Out (FIFO) queue record type and object
- Allow the user to:
  - Add ("push") items to the queue
  - Remove ("pop") the next item to be serviced from the queue (Print this item to ensure the order is correct)
- When the user is done manipulating the queue, print out the remaining items in the queue

## ■ Hints

- Queue record should at least contain:
  - Array of items
  - Index into array where next item will be added

## Record Types Lab Solution - Declarations

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure Main is
3
4     type Name_T is array (1 .. 6) of Character;
5     type Index_T is range 0 .. 1_000;
6     type Queue_T is array (Index_T range 1 .. 1_000) of Name_T;
7
8     type Fifo_Queue_T is record
9         Next_Available : Index_T := 1;
10        Last_Served     : Index_T := 0;
11        Queue           : Queue_T := (others => (others => ' '));
12    end record;
13
14    Queue : Fifo_Queue_T;
15    Choice : Integer;
```

# Record Types Lab Solution - Implementation

```
17 begin
18
19   loop
20     Put ("1 = add to queue | 2 = remove from queue | others => done: ");
21     Choice := Integer'Value (Get_Line);
22     if Choice = 1 then
23       Put ("Enter name: ");
24       Queue.Queue (Queue.Next_Available) := Name_T (Get_Line);
25       Queue.Next_Available := Queue.Next_Available + 1;
26     elsif Choice = 2 then
27       if Queue.Next_Available = 1 then
28         Put_Line ("Nobody in line");
29       else
30         Queue.Last_Served := Queue.Last_Served + 1;
31         Put_Line ("Now serving: " & String (Queue.Queue (Queue.Last_Served)));
32       end if;
33     else
34       exit;
35     end if;
36     New_Line;
37   end loop;
38
39   Put_Line ("Remaining in line: ");
40   for Index in Queue.Last_Served + 1 .. Queue.Next_Available - 1 loop
41     Put_Line (" " & String (Queue.Queue (Index)));
42   end loop;
43
44 end Main;
```

## Summary

# Summary

- Heterogeneous types allowed for components
- Default initial values allowed for components
  - Evaluated when each object elaborated, not the type
  - Not evaluated if explicit initial value specified
- Aggregates express literals for composite types
  - Can mix named and positional forms

# Subprograms

## Introduction

# Introduction

- Are syntactically distinguished as **function** and **procedure**
  - Functions represent *values*
  - Procedures represent *actions*

```
function Is_Leaf (T : Tree) return Boolean
procedure Split (T : in out Tree;
                 Left : out Tree;
                 Right : out Tree)
```

- Provide direct syntactic support for separation of specification from implementation

```
function Is_Leaf (T : Tree) return Boolean;
function Is_Leaf (T : Tree) return Boolean is
begin
...
end Is_Leaf;
```

# Recognizing Procedures and Functions

- Functions' results must be treated as values
  - And cannot be ignored
- Procedures cannot be treated as values
- You can always distinguish them via the call context

```
10 Open (Source, "SomeFile.txt");
11 while not End_of_File (Source) loop
12     Get (Next_Char, From => Source);
13     if Found (Next_Char, Within => Buffer) then
14         Display (Next_Char);
15         Increment;
16     end if;
17 end loop;
```

- Note that a subprogram without parameters (Increment on line 15) does not allow an empty set of parentheses

## A Little "Preaching" About Names

- Procedures are abstractions for actions
- Functions are abstractions for values
- Use names that reflect those facts!
  - Imperative verbs for procedure names
  - Nouns for function names, as for mathematical functions
    - Questions work for boolean functions

```
procedure Open (V : in out Valve);  
procedure Close (V : in out Valve);  
function Square_Root (V: Float) return Float;  
function Is_Open (V: Valve) return Boolean;
```

# Syntax

## Specification and Body

- Subprogram specification is the external (user) **interface**
  - **Declaration** and **specification** are used synonymously
- Specification may be required in some cases
  - eg. recursion
- Subprogram body is the **implementation**

## Procedure Specification Syntax (Simplified)

```
procedure Swap (A, B : in out Integer);
```

```
procedure_specification ::=
```

```
  procedure program_unit_name  
    { (parameter_specification  
      ; parameter_specification)};
```

```
parameter_specification ::=
```

```
  identifier_list : mode subtype_mark [ := expression ]
```

```
mode ::= [in] | out | in out
```

## Function Specification Syntax (Simplified)

```
function F (X : Float) return Float;
```

- Close to **procedure** specification syntax
  - With **return**
  - Can be an operator: + - \* / **mod rem** ...

```
function_specification ::=  
  function designator  
  { (parameter_specification  
    ; parameter_specification) }  
  return result_type;
```

```
designator ::= program_unit_name | operator_symbol
```

# Body Syntax

```
subprogram_specification is
    [declarations]
begin
    sequence_of_statements
end [designator];

procedure Hello is
begin
    Ada.Text_IO.Put_Line ("Hello World!");
    Ada.Text_IO.New_Line (2);
end Hello;

function F (X : Float) return Float is
    Y : constant Float := X + 3.0;
begin
    return X * Y;
end F;
```

# Completions

- Bodies **complete** the specification
  - There are **other** ways to complete
- Separate specification is **not required**
  - Body can act as a specification
- A declaration and its body must **fully** conform
  - Mostly **semantic** check
  - But parameters **must** have same name

```
procedure P (J, K : Integer)
procedure P (J : Integer; K : Integer)
procedure P (J, K : in Integer)
-- Invalid
procedure P (A : Integer; B : Integer)
```

# Completion Examples

## ■ Specifications

```
procedure Swap (A, B : in out Integer);  
function Min (X, Y : Person) return Person;
```

## ■ Completions

```
procedure Swap (A, B : in out Integer) is  
  Temp : Integer := A;  
begin  
  A := B;  
  B := Temp;  
end Swap;
```

```
-- Completion as specification
```

```
function Less_Than (X, Y : Person) return Boolean is  
begin  
  return X.Age < Y.Age;  
end Less_Than;
```

```
function Min (X, Y : Person) return Person is  
begin  
  if Less_Than (X, Y) then  
    return X;  
  else  
    return Y;  
  end if;  
end Min;
```

## Direct Recursion - No Declaration Needed

- When **is** is reached, the subprogram becomes **visible**
  - It can call **itself** without a declaration

```
type Vector_T is array (Natural range <>) of Integer;  
Empty_Vector : constant Vector_T (1 .. 0) := (others => 0);
```

```
function Get_Vector return Vector_T is  
  Next : Integer;  
begin  
  Get (Next);  
  
  if Next = 0 then  
    return Empty_Vector;  
  else  
    return Get_Vector & Next;  
  end if;  
end Input;
```

## Indirect Recursion Example

- Elaboration in **linear order**

```
procedure P;
```

```
procedure F is
```

```
begin
```

```
  P;
```

```
end F;
```

```
procedure P is
```

```
begin
```

```
  F;
```

```
end P;
```

# Quiz

Which profile is semantically different from the others?

- A. `procedure P (A : Integer; B : Integer);`
- B. `procedure P (A, B : Integer);`
- C. `procedure P (B : Integer; A : Integer);`
- D. `procedure P (A : in Integer; B : in Integer);`

# Quiz

Which profile is semantically different from the others?

- A. `procedure P (A : Integer; B : Integer);`
- B. `procedure P (A, B : Integer);`
- C. `procedure P (B : Integer; A : Integer);`
- D. `procedure P (A : in Integer; B : in Integer);`

Parameter names are important in Ada. The other selections have the names in the same order with the same mode and type.

## Parameters

# Subprogram Parameter Terminology

- *Actual parameters* are values passed to a call
  - Variables, constants, expressions
- *Formal parameters* are defined by specification
  - Receive the values passed from the actual parameters
  - Specify the types required of the actual parameters
  - Type **cannot** be anonymous

```
procedure Something (Formal1 : in Integer);
```

```
ActualX : Integer;
```

```
...
```

```
Something (ActualX);
```

## Parameter Associations in Calls

- Associate formal parameters with actuals
- Both positional and named association allowed

```
Something (ActualX, Formal2 => ActualY);
```

```
Something (Formal2 => ActualY, Formal1 => ActualX);
```

- Having named **then** positional is forbidden

```
-- Compilation Error
```

```
Something (Formal1 => ActualX, ActualY);
```

# Parameter Modes and Return

## ■ Mode **in**

- Formal parameter is **constant**
  - So actual is not modified either
- Can have **default**, used when **no value** is provided

```
procedure P (N : in Integer := 1; M : in Positive);  
[...]  
P (M => 2);
```

## ■ Mode **out**

- Writing is **expected**
- Reading is **allowed**
- Actual **must** be a writable object

## ■ Mode **in out**

- Actual is expected to be **both** read and written
- Actual **must** be a writable object

## ■ Function **return**

- **Must** always be handled

## Why Read Mode **out** Parameters?

- **Convenience** of writing the body
  - No need for readable temporary variable
- Warning: initial value is **not defined**

```
procedure Compute (Value : out Integer) is
begin
  Value := 0;
  for K in 1 .. 10 loop
    Value := Value + K; -- this is a read AND a write
  end loop;
end Compute;
```

# Parameter Passing Mechanisms

- *By-Copy*
  - The formal denotes a separate object from the actual
  - **in, in out**: actual is copied into the formal **on entry to** the subprogram
  - **out, in out**: formal is copied into the actual **on exit from** the subprogram
- *By-Reference*
  - The formal denotes a view of the actual
  - Reads and updates to the formal directly affect the actual
  - More efficient for large objects
- Parameter **types** control mechanism selection
  - Not the parameter **modes**
  - Compiler determines the mechanism

# By-Copy Vs By-Reference Types

- By-Copy
  - Scalar types
  - **access** types
- By-Reference
  - **tagged** types
  - **task** types and **protected** types
  - **limited** types
- **array, record**
  - By-Reference when they have by-reference **components**
  - By-Reference for **implementation-defined** optimizations
  - By-Copy otherwise
- **private** depends on its full definition
- Note that the parameter mode **aliased** will force pass-by-reference
  - This mode is discussed in the **Access Types** module

# Unconstrained Formal Parameters or Return

- Unconstrained **formals** are allowed
  - Constrained by **actual**
- Unconstrained **return** is allowed too
  - Constrained by the **returned object**

```
type Vector is array (Positive range <>) of Float;  
procedure Print (Formal : Vector);
```

```
Phase : Vector (X .. Y);
```

```
State : Vector (1 .. 4);
```

```
...
```

```
begin
```

```
Print (Phase);           -- Formal'Range is X .. Y
```

```
Print (State);          -- Formal'Range is 1 .. 4
```

```
Print (State (3 .. 4)); -- Formal'Range is 3 .. 4
```

## Unconstrained Parameters Surprise

- Assumptions about formal bounds may be **wrong**

```
type Vector is array (Positive range <>) of Float;  
function Subtract (Left, Right : Vector) return Vector;
```

```
V1 : Vector (1 .. 10); -- length = 10
```

```
V2 : Vector (15 .. 24); -- length = 10
```

```
R : Vector (1 .. 10); -- length = 10
```

```
...
```

```
-- What are the indices returned by Subtract?
```

```
R := Subtract (V2, V1);
```

## Naive Implementation

- **Assumes** bounds are the same everywhere
- Fails when `Left'First /= Right'First`
- Fails when `Left'Length /= Right'Length`
- Fails when `Left'First /= 1`

```
function Subtract (Left, Right : Vector)
  return Vector is
    Result : Vector (1 .. Left'Length);
begin
  ...
  for K in Result'Range loop
    Result (K) := Left (K) - Right (K);
  end loop;
```

## Correct Implementation

- Covers **all** bounds
- **return** indexed by Left'Range

```
function Subtract (Left, Right : Vector) return Vector is
  pragma Assert (Left'Length = Right'Length);

  Result : Vector (Left'Range);
  Offset : constant Integer := Right'First - Result'First;
begin
  for K in Result'Range loop
    Result (K) := Left (K) - Right (K + Offset);
  end loop;

  return Result;
end Subtract;
```

## Quiz

```
function F (P1 : in Integer := 0;
           P2 : in out Integer;
           P3 : in Character := ' ';
           P4 : out Character)
    return Integer;
J1, J2 : Integer;
C : Character;
```

Which call(s) is (are) legal?

- A. J1 := F (P1 => 1, P2 => J2, P3 => '3', P4 => '4');
- B. J1 := F (P1 => 1, P3 => '3', P4 => C);
- C. J1 := F (1, J2, '3', C);
- D. F (J1, J2, '3', C);

## Quiz

```
function F (P1 : in Integer := 0;
           P2 : in out Integer;
           P3 : in Character := ' ';
           P4 : out Character)
    return Integer;
J1, J2 : Integer;
C : Character;
```

Which call(s) is (are) legal?

- A. J1 := F (P1 => 1, P2 => J2, P3 => '3', P4 => '4');
- B. J1 := F (P1 => 1, P3 => '3', P4 => C);
- C. J1 := F (1, J2, '3', C);
- D. F (J1, J2, '3', C);

Explanations

- A. P4 is **out**, it **must** be a variable
- B. P2 has no default value, it **must** be specified
- C. Correct
- D. F is a function, its **return must** be handled

## Null Procedures

## Null Procedure Declarations

- Shorthand for a procedure body that does nothing
- Longhand form

```
procedure NOP is
begin
    null;
end NOP;
```

- Shorthand form

```
procedure NOP is null;
```

- The `null` statement is present in both cases
- Explicitly indicates nothing to be done, rather than an accidental removal of statements

## Null Procedures As Completions

- Completions for a distinct, prior declaration

```
procedure NOP;  
...  
procedure NOP is null;
```

- A declaration and completion together
  - A body is then not required, thus not allowed

```
procedure NOP is null;  
...  
procedure NOP is -- compile error  
begin  
    null;  
end NOP;
```

## Typical Use for Null Procedures: OOP

- When you want a method to be concrete, rather than abstract, but don't have anything for it to do
  - The method is then always callable, including places where an abstract routine would not be callable
  - More convenient than full null-body definition

## Null Procedure Summary

- Allowed where you can have a full body
  - Syntax is then for shorthand for a full null-bodied procedure
- Allowed where you can have a declaration!
  - Example: package declarations
  - Syntax is shorthand for both declaration and completion
    - Thus no body required/allowed
- Formal parameters are allowed

```
procedure Do_Something (P : in Integer) is null;
```

## Nested Subprograms

# Subprograms Within Subprograms

- Subprograms can be placed in any declarative block
  - So they can be nested inside another subprogram
  - Or even within a **declare** block
- Useful for performing sub-operations without passing parameter data

## Nested Subprogram Example

```
1  procedure Main is
2
3      function Read (Prompt : String) return Types.Line_T is
4  begin
5          Put (Prompt & "> ");
6          return Types.Line_T'Value (Get_Line);
7  end Read;
8
9      Lines : Types.Lines_T (1 .. 10);
10 begin
11     for J in Lines'Range loop
12         Lines (J) := Read ("Line " & J'Image);
13     end loop;
```

## Procedure Specifics

## Return Statements in Procedures

- Returns immediately to caller
- Optional
  - Automatic at end of body execution
- Fewer is traditionally considered better

```
procedure P is
begin
    ...
    if Some_Condition then
        return; -- early return
    end if;
    ...
end P; -- automatic return
```

# Main Subprograms

- Must be library subprograms
  - Not nested inside another subprogram
- No special subprogram unit name required
- Can be many per project
- Can always be procedures
- Can be functions if implementation allows it
  - Execution environment must know how to handle result

```
with Ada.Text_IO;  
procedure Hello is  
begin  
  Ada.Text_IO.Put ("Hello World");  
end Hello;
```

## Function Specifics

## Return Statements in Functions

- Must have at least one
  - Compile-time error otherwise
  - Unless doing machine-code insertions
- Returns a value of the specified (sub)type
- Syntax

```
function defining_designator [formal_part]
    return subtype_mark is
    declarative_part
begin
    {statements}
    return expression;
end designator;
```

## No Path Analysis Required by Compiler

- Running to the end of a function without hitting a **return** statement raises `Program_Error`
- Compilers can issue warning if they suspect that a **return** statement will not be hit

```
function Greater (X, Y : Integer) return Boolean is  
begin  
  if X > Y then  
    return True;  
  end if;  
end Greater; -- possible compile warning
```

## Multiple Return Statements

- Allowed
- Sometimes the most clear

```
function Truncated (R : Float) return Integer is
  Converted : Integer := Integer (R);
begin
  if R - Float (Converted) < 0.0 then -- rounded up
    return Converted - 1;
  else -- rounded down
    return Converted;
  end if;
end Truncated;
```

## Multiple Return Statements Versus One

- Many can detract from readability
- Can usually be avoided

```
function Truncated (R : Float) return Integer is
  Result : Integer := Integer (R);
begin
  if R - Float (Result) < 0.0 then -- rounded up
    Result := Result - 1;
  end if;
  return Result;
end Truncated;
```

## Function Dynamic-Size Results

```
function Char_Mult (C : Character; L : Natural)
  return String is
  R : String (1 .. L) := (others => C);
begin
  return R;
end Char_Mult;

X : String := Char_Mult ('x', 4);

begin
  -- OK
  pragma Assert (X'Length = 4 and X = "xxxx");
```

## Expression Functions

# Expression Functions

- Functions whose implementations are pure expressions
  - No other completion is allowed
  - No **return** keyword
- May exist only for sake of pre/postconditions

```
function function_specification is (expression);
```

NB: Parentheses around expression are **required**

- Can complete a prior declaration

```
function Squared (X : Integer) return Integer;  
function Squared (X : Integer) return Integer is  
  (X ** 2);
```

## Expression Functions Example

- Expression function

```
function Square (X : Integer) return Integer is (X ** 2);
```

- Is equivalent to

```
function Square (X : Integer) return Integer is  
begin  
    return X ** 2;  
end Square;
```

# Quiz

Which statement is True?

- A. Expression functions cannot be nested functions.
- B. Expression functions require a specification and a body.
- C. Expression functions must have at least one `return` statement.
- D. Expression functions can have "out" parameters.

# Quiz

Which statement is True?

- A. Expression functions cannot be nested functions.
- B. Expression functions require a specification and a body.
- C. Expression functions must have at least one `return` statement.
- D. *Expression functions can have "out" parameters.*

Explanation

- A. They **can** be nested subprograms (just like any other subprogram)
- B. As in other subprograms, the implementation can serve as the specification
- C. Because they are expressions, the `return` statement is not allowed
- D. An expression function does not allow assignment statements, but it can call another function that is **not** an expression function.

```
function Normal_Fun (Input : Character;
                    Output : out Integer)
    return Boolean is
begin
    Output := Character'Pos (Input);
    return True;
end Normal_Fun;

function Expr_Fun (Input : Character;
                  Output : out Integer)
    return Boolean is
    (Normal_Fun (Character'Succ (Input), Output));
```

## Potential Pitfalls

## Mode **out** Risk for Scalars

- Always assign value to **out** parameters
- Else "By-copy" mechanism will copy something back
  - May be junk
  - `Constraint_Error` or unknown behaviour further down

```
procedure P
  (A, B : in Some_Type; Result : out Scalar_Type) is
begin
  if Some_Condition then
    return;  -- Result not set
  end if;
  ...
  Result := Some_Value;
end P;
```

## "Side Effects"

- Any effect upon external objects or external environment
  - Typically alteration of non-local variables or states
  - Can cause hard-to-debug errors
  - Not legal for **function** in SPARK
- Can be there for historical reasons
  - Or some design patterns

```
Global : Integer := 0;
```

```
function F (X : Integer) return Integer is  
begin  
    Global := Global + X;  
    return Global;  
end F;
```

## Order-Dependent Code and Side Effects

```
Global : Integer := 0;
```

```
function Inc return Integer is  
begin  
  Global := Global + 1;  
  return Global;  
end Inc;
```

```
procedure Assert_Equals (X, Y : in Integer);  
...  
Assert_Equals (Global, Inc);
```

- Language does **not** specify parameters' order of evaluation
- `Assert_Equals` could get called with
  - $X \rightarrow 0, Y \rightarrow 1$  (if `Global` evaluated first)
  - $X \rightarrow 1, Y \rightarrow 1$  (if `Inc` evaluated first)

# Parameter Aliasing

- **Aliasing**: Multiple names for an actual parameter inside a subprogram body
- Possible causes:
  - Global object used is also passed as actual parameter
  - Same actual passed to more than one formal
  - Overlapping **array** slices
  - One actual is a component of another actual
- Can lead to code dependent on parameter-passing mechanism
- Ada detects some cases and raises `Program_Error`

```
procedure Update (Doubled, Tripled : in out Integer);
```

```
...
```

```
Update (Doubled => A, Tripled => A);
```

```
error: writable actual for "Doubled" overlaps with actual for "Tripled"
```

# Functions' Parameter Modes

- Can be mode **in out** and **out** too
- **Note:** operator functions can only have mode **in**
  - Including those you overload
  - Keeps readers sane
- Justification for only mode **in** in earlier versions of the language
  - No side effects: should be like mathematical functions
  - But side effects are still possible via globals
  - So worst possible case: side effects are possible and necessarily hidden!

## Easy Cases Detected and Not Legal

```
procedure Example (A : in out Positive) is
  function Increment (This : Integer) return Integer is
  begin
    A := A + This;
    return A;
  end Increment;
  X : array (1 .. 10) of Integer;
begin
  -- order of evaluating A not specified
  X (A) := Increment (A);
end Example;
```

## Extended Example

# Implementing a Simple "Set"

- We want to indicate which colors of the rainbow are in a **set**
  - If you remember from the *Basic Types* module, a type is made up of values and primitive operations
- Our values will be
  - Type indicating colors of the rainbow
  - Type to group colors
  - Mechanism to indicate which color is in our set
- Our primitive operations will be
  - Create a set
  - Add a color to the set
  - Remove a color from the set
  - Check if color is in set

## Values for the Set

- Colors of the rainbow

```
type Color_T is (Red, Orange, Yellow, Green,  
                Blue, Indigo, Violet,  
                White, Black);
```

- Group of colors

```
type Group_Of_Colors_T is  
    array (Positive range <>) of Color_T;
```

- Mechanism indicating which color is in the set

```
type Set_T is array (Color_T) of Boolean;  
-- if array component at Color is True,  
-- the color is in the set
```

## Primitive Operations for the Set

- Create a set

```
function Make (Colors : Group_Of_Colors_T) return Set_T;
```

- Add a color to the set

```
procedure Add (Set      : in out Set_T;  
              Color   :      Color_T);
```

- Remove a color from the set

```
procedure Remove (Set      : in out Set_T;  
                Color   :      Color_T);
```

- Check if color is in set

```
function Contains (Set   : Set_T;  
                 Color : Color_T)  
              return Boolean;
```

# Implementation of the Primitive Operations

- Implementation of the primitives is easy
  - We could do operations directly on Set\_T, but that's not flexible

```
function Make (Colors : Group_Of_Colors_T) return Set_T is
  Set : Set_T := (others => False);
begin
  for Color of Colors loop
    Set (Color) := True;
  end loop;
  return Set;
end Make;

procedure Add (Set   : in out Set_T;
              Color :      Color_T) is
begin
  Set (Color) := True;
end Add;

procedure Remove (Set   : in out Set_T;
                 Color :      Color_T) is
begin
  Set (Color) := False;
end Remove;

function Contains (Set   : Set_T;
                  Color : Color_T)
  return Boolean is
  (Set (Color));
```

## Using our Set Construct

```
Rgb    : Set_T := Make ((Red, Green, Blue));  
Light : Set_T := Make ((Red, Yellow, Green));  
  
if Contains (Rgb, Black) then  
    Remove (Rgb, Black);  
else  
    Add (Rgb, Black);  
end if;
```

*In addition, because of the operations available to arrays of Boolean, we can easily implement set operations*

```
Union           : Set_T := Rgb or Light;  
Intersection    : Set_T := Rgb and Light;  
Difference      : Set_T := Rgb xor Light;
```

Lab

# Subprograms Lab

## ■ Requirements

- Build a list of sorted unique integers
  - Do not add an integer to the list if it is already there
- Print the list

## ■ Hints

- Subprograms can be nested inside other subprograms
  - Like inside **main**
- Build a Search subprogram to find the correct insertion point in the list

# Subprograms Lab Solution - Search

```
4  type List_T is array (Positive range <>) of Integer;
5
6  function Search
7      (List : List_T;
8       Item : Integer)
9      return Positive is
10     begin
11         if List'Length = 0 then
12             return 1;
13         elsif Item <= List (List'First) then
14             return 1;
15         else
16             for Idx in (List'First + 1) .. List'Length loop
17                 if Item <= List (Idx) then
18                     return Idx;
19                 end if;
20             end loop;
21             return List'Last;
22         end if;
23     end Search;
```

# Subprograms Lab Solution - Main

```
25  procedure Add (Item : Integer) is
26      Place : Natural := Search (List (1..Length), Item);
27  begin
28      if List (Place) /= Item then
29          Length                := Length + 1;
30          List (Place + 1 .. Length) := List (Place .. Length - 1);
31          List (Place)           := Item;
32      end if;
33  end Add;
34
35  begin
36
37      Add (100);
38      Add (50);
39      Add (25);
40      Add (50);
41      Add (90);
42      Add (45);
43      Add (22);
44
45      for Idx in 1 .. Length loop
46          Put_Line (List (Idx)'Image);
47      end loop;
48
49  end Main;
```

## Summary

# Summary

- **procedure** is abstraction for actions
- **function** is abstraction for value computations
- Separate declarations are sometimes necessary
  - Mutual recursion
  - Visibility from packages (i.e., exporting)
- Modes allow spec to define effects on actuals
  - Don't have to see the implementation: abstraction maintained
- Parameter-passing mechanism is based on the type
- Watch those side effects!

# Type Derivation

# Introduction

# Type Derivation

- Type *derivation* allows for reusing code
- Type can be **derived** from a **base type**
- Base type can be substituted by the derived type
- Subprograms defined on the base type are **inherited** on derived type
- This is **not** OOP in Ada
  - Tagged derivation **is** OOP in Ada

## Reminder: What is a Type?

- A type is characterized by two components
  - Its data structure
  - The set of operations that applies to it
- The operations are called **primitive operations** in Ada

```
package Types is
```

```
  type Integer_T is range -(2**63) .. 2**63-1 with Size => 64;
```

```
  procedure Increment_With_Truncation (Val : in out Integer_T);
```

```
  procedure Increment_With_Rounding (Val : in out Integer_T);
```

```
end Types;
```

## Simple Derivation

# Simple Type Derivation

- Any type (except **tagged**) can be derived

```
type Natural_T is new Integer_T range 0 .. Integer_T'Last;
```

- Natural\_T inherits from:

- The data **representation** of the parent

- Integer based, 64 bits

- The **primitives** of the parent

- Increment\_With\_Truncation and Increment\_With\_Rounding

- The types are not the same

```
I_Obj : Integer_T := 0;
```

```
N_Obj : Natural_T := 0;
```

- I\_Obj := N\_Obj; → generates a compile error

expected type "Integer\_T" defined at line 2

- But a child can be converted to the parent

- I\_Obj := Integer\_T (N\_Obj);

# Simple Derivation and Type Structure

- The type "structure" can not change

- `array` cannot become `record`
- Integers cannot become floats

- But can be **constrained** further

- Scalar ranges can be reduced

```
type Positive_T is new Natural_T range 1 .. Natural_T'Last;
```

- Unconstrained types can be constrained

```
type Arr_T is array (Integer range <>) of Integer;
```

```
type Ten_Elem_Arr_T is new Arr_T (1 .. 10);
```

```
type Rec_T (Size : Integer) is record
```

```
    Elem : Arr_T (1 .. Size);
```

```
end record;
```

```
type Ten_Elem_Rec_T is new Rec_T (10);
```

# Primitives

# Primitive Operations

- Primitive Operations are those subprograms associated with a type

```
type Integer_T is range -(2**63) .. 2**63-1 with Size => 64;  
procedure Increment_With_Truncation (Val : in out Integer_T);  
procedure Increment_With_Rounding (Val : in out Integer_T);
```

- Most types have some primitive operations defined by the language
  - e.g. equality operators for most types, numeric operators for integers and floats
- A primitive operation on the parent can receive an object of a child type with no conversion

```
declare  
    N_Obj : Natural_T := 1234;  
begin  
    Increment_With_Truncation (N_Obj);  
end;
```

# General Rule for Defining a Primitive

- Primitives are subprograms
- Subprogram *S* is a primitive of type *T* if and only if:
  - *S* is declared in the scope of *T*
  - *S* uses type *T*
    - As a parameter
    - As its return type (for a **function**)
  - *S* is above **freeze-point** (see next section)
- Standard practice
  - Primitives should be declared **right after** the type itself
  - In a scope, declare at most a **single** type with primitives

```
package P is
  type T is range 1 .. 10;
  procedure P1 (V : T);
  procedure P2 (V1 : Integer; V2 : T);
  function F return T;
end P;
```

## Primitive of Multiple Types

A subprogram can be a primitive of several types

```
package P is
  type Distance_T is range 0 .. 9999;
  type Percentage_T is digits 2 range 0.0 .. 1.0;
  type Units_T is (Meters, Feet, Furlongs);

  procedure Convert (Value  : in out Distance_T;
                    Source  :          Units_T;
                    Result  :          Units_T);
  procedure Shrink (Value   : in out Distance_T;
                   Percent  :          Percentage_T);

end P;
```

- Convert and Shrink are primitives for Distance\_T
- Convert is also a primitive of Units\_T
- Shrink is also a primitive of Percentage\_T

## Creating Primitives for Children

- Just because we can inherit a primitive from our parent doesn't mean we want to
- We can create a new primitive (with the same name as the parent) for the child
  - Very similar to overloaded subprograms
  - But added benefit of visibility to grandchildren
- We can also remove a primitive (see next slide)

```
type Integer_T is range -(2**63) .. 2**63-1;  
procedure Increment_With_Truncation (Val : in out Integer_T);  
procedure Increment_With_Rounding (Val : in out Integer_T);
```

```
type Child_T is new Integer_T range -1000 .. 1000;  
procedure Increment_With_Truncation (Val : in out Child_T);
```

```
type Grandchild_T is new Child_T range -100 .. 100;  
procedure Increment_With_Rounding (Val : in out Grandchild_T);
```

# Overriding Indications

- **Optional** indications

- Checked by compiler

```
type Child_T is new Integer_T range -1000 .. 1000;
procedure Increment_With_Truncation
  (Val : in out Child_T);
procedure Just_For_Child
  (Val : in out Child_T);
```

- **Replacing** a primitive: **overriding** indication

```
overriding procedure Increment_With_Truncation
  (Val : in out Child_T);
```

- **Adding** a primitive: **not overriding** indication

```
not overriding procedure Just_For_Child
  (Val : in out Child_T);
```

- **Removing** a primitive: **overriding** as **abstract**

```
overriding procedure Just_For_Child
  (Val : in out Grandchild_T) is abstract;
```

- Using **overriding** or **not overriding** incorrectly will generate a compile error

## Quiz

```
type T is new Integer;
```

Which operator(s) definition(s) is (are) legal?

- A. function "+" (V : T) return Boolean is (V /= 0)
- B. function "+" (A, B : T) return T is (A + B)
- C. function "=" (A, B : T) return T is (A - B)
- D. function "!=" (A : T) return T is (A)

## Quiz

```
type T is new Integer;
```

Which operator(s) definition(s) is (are) legal?

- A. `function "+" (V : T) return Boolean is (V /= 0)`
  - B. `function "+" (A, B : T) return T is (A + B)`
  - C. `function "=" (A, B : T) return T is (A - B)`
  - D. `function ":=" (A : T) return T is (A)`
- 
- B. Infinite recursion
  - C. Unlike some languages, there is no assignment operator

## Freeze Point

# What is the "Freeze Point"?

- Ada doesn't explicitly identify the end of the "scope" of a type
  - The compiler needs to know it for determining primitive operations
  - Also needed for other situations (described elsewhere)
- This end is the implicit **freeze point** occurring whenever:
  - A **variable** of the type is **declared**
  - The type is **derived**
  - The **end of the scope** is reached
- Subprograms past this "freeze point" are not primitive operations

```
type Parent is Integer;  
procedure Prim (V : Parent);
```

```
type Child is new Parent;
```

```
-- Parent has been derived, so it is frozen.
```

```
-- Prim2 is not a primitive
```

```
procedure Prim2 (V : Parent);
```

```
V : Child;
```

```
-- Child used in an object declaration, so it is frozen
```

```
-- Prim3 is not a primitive
```

```
procedure Prim3 (V : Child);
```

# Debugging Type Freeze

- Freeze → Type **completely** defined
- Compiler does **need** to determine the freeze point
  - To instantiate, derive, get info on the type ('Size)...
  - Freeze rules are a guide to place it
  - Actual choice is more technical
    - May contradict the standard
- `-gnatDG` to get **expanded** source
  - **Pseudo-Ada** debug information

pkg.ads

```
type Up_To_Eleven is range 0 .. 11;
```

<obj>/pkg.ads.dg

```
type example__up_to_eleven_t is range 0 .. 11;           -- type declaration
[type example__Tup_to_eleven_tB is new short_short_integer] -- representation
freeze example__Tup_to_eleven_tB []                     -- freeze representation
freeze example__up_to_eleven_t []                       -- freeze representation
```

# Quiz

```
type Parent is range 1 .. 100;
procedure Proc_A (X : in out Parent);

type Child is new Parent range 2 .. 99;
procedure Proc_B (X : in out Parent);
procedure Proc_B (X : in out Child);

-- Other scope
procedure Proc_C (X : in out Child);

type Grandchild is new Child range 3 .. 98;

procedure Proc_C (X : in out Grandchild);
```

Which are Parent's primitives?

- A. Proc\_A
- B. Proc\_B
- C. Proc\_C
- D. No primitives of Parent

# Quiz

```
type Parent is range 1 .. 100;
procedure Proc_A (X : in out Parent);

type Child is new Parent range 2 .. 99;
procedure Proc_B (X : in out Parent);
procedure Proc_B (X : in out Child);

-- Other scope
procedure Proc_C (X : in out Child);

type Grandchild is new Child range 3 .. 98;

procedure Proc_C (X : in out Grandchild);
```

Which are Parent's primitives?

- A. *Proc\_A*
- B. Proc\_B
- C. Proc\_C
- D. No primitives of Parent

Explanations

- A. Correct
- B. Freeze: Parent has been derived
- C. Freeze: scope change
- D. Incorrect

## Summary

# Summary

- *Primitive* of a type
  - Subprogram above **freeze-point** that takes or returns the type
  - Can be a primitive for **multiple types**
- Freeze point rules can be tricky
- Simple type derivation
  - Types derived from other types can only **add limitations**
    - Constraints, ranges
    - Cannot change underlying structure

# Expressions

# Introduction

## Advanced Expressions

- Different categories of expressions above simple assignment and conditional statements
  - Constraining types to sub-ranges to increase readability and flexibility
    - Allows for simple membership checks of values
  - Embedded conditional assignments
    - Equivalent to C's `A ? B : C` and even more elaborate

## Membership Tests

# "Membership" Operation

## ■ Syntax

```
simple_expression [not] in membership_choice_list
membership_choice_list ::= membership_choice
                           { | membership_choice}
membership_choice ::= expression | range | subtype_mark
```

## ■ Acts like a boolean function

## ■ Usable anywhere a boolean value is allowed

```
X : Integer := ...
B : Boolean := X in 0..5;
C : Boolean := X not in 0..5; -- also "not (X in 0..5)"
```

## Testing Constraints Via Membership

```
type Calendar_Days is
    (Mon, Tues, Wed, Thur, Fri, Sat, Sun);
subtype Weekdays is Calendar_Days range Mon .. Fri;
Day : Calendar_Days := Today;
...
if Day in Mon .. Fri then ...
if Day in Weekdays then ... -- same as above
```

## Testing Non-Contiguous Membership

- We use `in` to indicate membership in a range of values

```
if Color in Red .. Green then
if Index in List'Range then
```

- But what if the values are not contiguous?

- We could use a Boolean conjunction

```
if Index = 1 or Index = 3 or Index = 5 then
```

- Or we could simplify it by specifying a collection (or set)

```
if Index in 1 | 3 | 5 then
```

- `|` is used to separate members
- So `1 | 3 | 5` is the set for which we are verifying membership

# Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
subtype Weekdays_T is Days_T range Mon .. Fri;  
Today : Days_T;
```

Which condition(s) is (are) legal?

- A. if Today = Mon or Wed or Fri then
- B. if Today in Days\_T then
- C. if Today not in Weekdays\_T then
- D. if Today in Tue | Thu then

# Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
subtype Weekdays_T is Days_T range Mon .. Fri;  
Today : Days_T;
```

Which condition(s) is (are) legal?

- A. `if Today = Mon or Wed or Fri then`
- B. `if Today in Days_T then`
- C. `if Today not in Weekdays_T then`
- D. `if Today in Tue | Thu then`

Explanations

- A. Wed and Fri are not Boolean expressions - need to compare each of them to Today
- B. Legal - should always return True
- C. Legal - returns True if Today is Sat or Sun
- D. Legal - returns True if Today is Tue or Thu

## Qualified Names

# Qualification

- Explicitly indicates the subtype of the value
- Syntax

```
qualified_expression ::= subtype_mark'(expression) |  
                        subtype_mark'aggregate
```

- Similar to conversion syntax
  - Mnemonic - "qualification uses quote"
- Various uses shown in course
  - Testing constraints
  - Removing ambiguity of overloading
  - Enhancing readability via explicitness

# Testing Constraints Via Qualification

- Asserts value is compatible with subtype
  - Raises exception `Constraint_Error` if not true

```
subtype Weekdays is Days range Mon .. Fri;
This_Day : Days;
...
case Weekdays'(This_Day) is -- run-time error if out of range
  when Mon =>
    Arrive_Late;
    Leave_Early;
  when Tue .. Thur =>
    Arrive_Early;
    Leave_Late;
  when Fri =>
    Arrive_Early;
    Leave_Early;
end case; -- no 'others' because all subtype values covered
```

## Conditional Expressions

# Conditional Expressions

- Ultimate value depends on a controlling condition
- Allowed wherever an expression is allowed
  - Assignment RHS, formal parameters, aggregates, etc.
- Similar intent as in other languages
  - Java, C/C++ ternary operation **A ? B : C**
  - Python conditional expressions
  - etc.
- Two forms:
  - *If expressions*
  - *Case expressions*

# If Expressions

- Syntax looks like an *if statement* without **end if**

```
if_expression ::=  
    (if condition then dependent_expression  
     {elsif condition then dependent_expression}  
     [else dependent_expression])  
condition ::= boolean_expression
```

- The conditions are always Boolean values

```
(if Today > Wednesday then 1 else 0)
```

## Result Must Be Compatible with Context

- The **dependent\_expression** parts, specifically

```
X : Integer :=  
  (if Day_Of_Week (Clock) > Wednesday then 1 else 0);
```

## "If Expression" Example

```
declare
  Remaining : Natural := 5;  -- arbitrary
begin
  while Remaining > 0 loop
    Put_Line ("Warning! Self-destruct in" &
              Remaining'Image &
              (if Remaining = 1 then " second" else " seconds"));
    delay 1.0;
    Remaining := Remaining - 1;
  end loop;
  Put_Line ("Boom! (goodbye Nostromo)");
```

## Boolean "If Expressions"

- Return a value of either True or False
  - `(if P then Q)` - assuming **P** and **Q** are **Boolean**
  - "If P is True then the result of the *if expression* is the value of Q"
- But what is the overall result if all conditions are False?
- Answer: the default result value is True
  - Why?
    - Consistency with mathematical proving

## The "else" Part When Result Is Boolean

- Redundant because the default result is True

```
(if P then Q else True)
```

- So for convenience and elegance it can be omitted

```
Acceptable : Boolean := (if P1 > 0 then P2 > 0 else True);  
Acceptable : Boolean := (if P1 > 0 then P2 > 0);
```

- Use **else** if you need to return False at the end

## Rationale for Parentheses Requirement

- Prevents ambiguity regarding any enclosing expression

- Problem:

```
X : Integer := if condition then A else B + 1;
```

- Does that mean

- If condition, then  $X := A + 1$ , else  $X := B + 1$  OR
- If condition, then  $X := A$ , else  $X := B + 1$

- But not required if parentheses already present

- Because enclosing construct includes them

```
Subprogram_Call (if A then B else C);
```

# When to Use If Expressions

- When you need computation to be done prior to sequence of statements
  - Allows constants that would otherwise have to be variables
- When an enclosing function would be either heavy or redundant with enclosing context
  - You'd already have written a function if you'd wanted one
- Preconditions and postconditions
  - All the above reasons
  - Puts meaning close to use rather than in package body
- Static named numbers
  - Can be much cleaner than using Boolean'Pos (Condition)

## "If Expression" Example for Constants

- Starting from

```
End_of_Month : array (Months) of Days
:= (Sep | Apr | Jun | Nov => 30,
    Feb => 28,
    others => 31);
begin
  if Leap (Today.Year) then -- adjust for leap year
    End_of_Month (Feb) := 29;
  end if;
  if Today.Day = End_of_Month (Today.Month) then
  ...
```

- Using *if expression* to call Leap (Year) as needed

```
End_of_Month : constant array (Months) of Days
:= (Sep | Apr | Jun | Nov => 30,
    Feb => (if Leap (Today.Year)
            then 29 else 28),
    others => 31);
begin
  if Today.Day /= End_of_Month (Today.Month) then
  ...
```

# Case Expressions

- Syntax similar to *case statements*
  - Lighter: no closing **end case**
  - Commas between choices
- Same general rules as *if expressions*
  - Parentheses required unless already present
  - Type of "result" must match context
- Advantage over *if expressions* is completeness checked by compiler
- Same as with **case** statements (unless **others** is used)

*-- compile error if not all days covered*

```
Hours : constant Integer :=  
  (case Day_of_Week is  
   when Mon .. Thurs => 9,  
   when Fri           => 4,  
   when Sat | Sun     => 0);
```

## "Case Expression" Example

```
Leap : constant Boolean :=
    (Today.Year mod 4 = 0 and Today.Year mod 100 /= 0)
    or else
    (Today.Year mod 400 = 0);
End_Of_Month : array (Months) of Days;
...
-- initialize array
for M in Months loop
    End_Of_Month (M) :=
        (case M is
            when Sep | Apr | Jun | Nov => 30,
            when Feb => (if Leap then 29 else 28),
            when others => 31);
end loop;
```

## Quiz

```
function Sqrt (X : Float) return Float;  
F : Float;  
B : Boolean;
```

Which statement(s) is (are) legal?

- A. `F := if X < 0.0 then Sqrt (-1.0 * X) else Sqrt (X);`
- B. `F := Sqrt (if X < 0.0 then -1.0 * X else X);`
- C. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);`
- D. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);`

## Quiz

```
function Sqrt (X : Float) return Float;  
F : Float;  
B : Boolean;
```

Which statement(s) is (are) legal?

- A.** `F := if X < 0.0 then Sqrt (-1.0 * X) else Sqrt (X);`
- B.** `F := Sqrt (if X < 0.0 then -1.0 * X else X);`
- C.** `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);`
- D.** `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);`

Explanations

- A.** Missing parentheses around expression
- B.** Legal - Expression is already enclosed in parentheses so you don't need to add more
- C.** Legal - `else True` not needed but is allowed
- D.** Legal - B will be True if  $X \geq 0.0$

## Quantified Expressions

# Introduction

- Expressions that have a Boolean value
- The value indicates something about a set of objects
  - In particular, whether something is True about that set
- That "something" is expressed as an arbitrary boolean expression
  - A so-called "predicate"
- "Universal" quantified expressions
  - Indicate whether predicate holds for all components
- "Existential" quantified expressions
  - Indicate whether predicate holds for at least one component

# Semantics Are As If You Wrote This Code

```
function Universal (Set : Components) return Boolean is
begin
  for C of Set loop
    if not Predicate (C) then
      return False; -- Predicate must be true for all
    end if;
  end loop;
  return True;
end Universal;
```

```
function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True; -- Predicate need only be true for one
    end if;
  end loop;
  return False;
end Existential;
```

# Quantified Expressions Syntax

- Four **for** variants
  - Index-based **in** or component-based **of**
  - Existential **some** or universal **all**
- Using arrow => to indicate *predicate* expression

```
(for some Index in Subtype_T => Predicate (Index))
```

```
(for all Index in Subtype_T => Predicate (Index))
```

```
(for some Value of Container_Obj => Predicate (Value))
```

```
(for all Value of Container_Obj => Predicate (Value))
```

## Simple Examples

```
Values : constant array (1 .. 10) of Integer := (...);  
Is_Any_Even : constant Boolean :=  
    (for some V of Values => V mod 2 = 0);  
Are_All_Even : constant Boolean :=  
    (for all V of Values => V mod 2 = 0);
```

# Universal Quantifier

- In logic, denoted by  $\forall$  (inverted 'A', for "all")
- "There is no member of the set for which the predicate does not hold"
  - If predicate is False for any member, the whole is False
- Functional equivalent

```
function Universal (Set : Components) return Boolean is
begin
  for C of Set loop
    if not Predicate (C) then
      return False; -- Predicate must be true for all
    end if;
  end loop;
  return True;
end Universal;
```

## Universal Quantifier Illustration

- "There is no member of the set for which the predicate does not hold"
- Given a set of integer answers to a quiz, there are no answers that are not 42 (i.e., all are 42)

```
Ultimate_Answer : constant := 42; -- to everything...
```

```
Answers : constant array (1 .. 10)  
  of Integer := (...);
```

```
All_Correct_1 : constant Boolean :=  
  (for all Component of Answers =>  
    Component = Ultimate_Answer);
```

```
All_Correct_2 : constant Boolean :=  
  (for all K in Answers'Range =>  
    Answers (K) = Ultimate_Answer);
```

## Universal Quantifier Real-World Example

```
type DMA_Status_Flag is (...);  
function Status_Indicated (  
  Flag : DMA_Status_Flag)  
  return Boolean;  
None_Set : constant Boolean := (  
  for all Flag in DMA_Status_Flag =>  
    not Status_Indicated (Flag));
```

# Existential Quantifier

- In logic, denoted by  $\exists$  (rotated 'E', for "exists")
- "There is at least one member of the set for which the predicate holds"
  - If predicate is True for any member, the whole is True
- Functional equivalent

```
function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True; -- Need only be true for at least one
    end if;
  end loop;
  return False;
end Existential;
```

## Existential Quantifier Illustration

- "There is at least one member of the set for which the predicate holds"
- Given set of Integer answers to a quiz, there is at least one answer that is 42

```
Ultimate_Answer : constant := 42; -- to everything...
```

```
Answers : constant array (1 .. 10)  
  of Integer := (...);
```

```
Any_Correct_1 : constant Boolean :=  
  (for some Component of Answers =>  
    Component = Ultimate_Answer);
```

```
Any_Correct_2 : constant Boolean :=  
  (for some K in Answers'Range =>  
    Answers (K) = Ultimate_Answer);
```

## Index-Based Vs Component-Based Indexing

- Given an array of Integers

```
Values : constant array (1 .. 10) of Integer := (...);
```

- Component-based indexing is useful for checking individual values

```
Contains_Negative_Number : constant Boolean :=  
  (for some N of Values => N < 0);
```

- Index-based indexing is useful for comparing across values

```
Is_Sorted : constant Boolean :=  
  (for all I in Values'Range =>  
    I = Values'First or else  
    Values (I) >= Values (I-1));
```

## "Pop Quiz" for Quantified Expressions

- What will be the value of **Ascending\_Order**?

```
Table : constant array (1 .. 10) of Integer :=  
      (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
Ascending_Order : constant Boolean := (  
  for all K in Table'Range =>  
    K > Table'First and then Table (K - 1) <= Table (K));
```

- Answer: **False**. Predicate fails when **K = Table'First**
  - First subcondition is False!
  - Condition should be

```
Ascending_Order : constant Boolean := (  
  for all K in Table'Range =>  
    K = Table'First or else Table (K - 1) <= Table (K));
```

## When the Set Is Empty...

- Universally quantified expressions are True
  - Definition: there is no member of the set for which the predicate does not hold
  - If the set is empty, there is no such member, so True
  - "All people 12-feet tall will be given free chocolate."
- Existentially quantified expressions are False
  - Definition: there is at least one member of the set for which the predicate holds
- If the set is empty, there is no such member, so False
- Common convention in set theory, arbitrary but settled

## Not Just Arrays: Any "Iterable" Objects

- Those that can be iterated over
- Language-defined, such as the containers
- User-defined too

```
package Characters is new
```

```
  Ada.Containers.Vectors (Positive, Character);
```

```
use Characters;
```

```
Alphabet   : constant Vector :=
```

```
  To_Vector ('A',1) & 'B' & 'C';
```

```
Any_Zed    : constant Boolean :=
```

```
  (for some C of Alphabet => C = 'Z');
```

```
All_Lower  : constant Boolean :=
```

```
  (for all C of Alphabet => Is_Lower (C));
```

## Conditional / Quantified Expression Usage

- Use them when a function would be too heavy
- Don't over-use them!

```
if (for some Component of Answers =>  
    Component = Ultimate_Answer)  
then
```

- Function names enhance readability
  - So put the quantified expression in a function

```
if At_Least_One_Answered (Answers) then
```

- Even in pre/postconditions, use functions containing quantified expressions for abstraction

## Quiz

Which declaration(s) is (are) legal?

- A.** `function F (S : String) return Boolean is  
 (for all C of S => C /= ' ');`
- B.** `function F (S : String) return Boolean is  
 (not for some C of S => C = ' ');`
- C.** `function F (S : String) return String is  
 (for all C of S => C);`
- D.** `function F (S : String) return String is  
 (if (for all C of S => C /= ' ') then "OK"  
 else "NOK");`

## Quiz

Which declaration(s) is (are) legal?

- A. *function F (S : String) return Boolean is (for all C of S => C /= ' ');*
  - B. `function F (S : String) return Boolean is (not for some C of S => C = ' ');`
  - C. `function F (S : String) return String is (for all C of S => C);`
  - D. *function F (S : String) return String is (if (for all C of S => C /= ' ') then "OK" else "NOK");*
- B. Parentheses required around the quantified expression
- C. Must return a **Boolean**

## Quiz

```
type T1 is array (1 .. 3) of Integer;  
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code correctly perform(s) equality check on A and B?

- A.** `function "=" (A : T1; B : T2) return Boolean is  
    (A = T1 (B));`
- B.** `function "=" (A : T1; B : T2) return Boolean is  
    (for all E1 of A => (for all E2 of B => E1 = E2));`
- C.** `function "=" (A : T1; B : T2) return Boolean is  
    (for some E1 of A => (for some E2 of B => E1 =  
    E2));`
- D.** `function "=" (A : T1; B : T2) return Boolean is  
    (for all J in A'Range => A (J) = B (J));`

## Quiz

```
type T1 is array (1 .. 3) of Integer;  
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code correctly perform(s) equality check on A and B?

- A.** `function "=" (A : T1; B : T2) return Boolean is  
    (A = T1 (B));`
- B.** `function "=" (A : T1; B : T2) return Boolean is  
    (for all E1 of A => (for all E2 of B => E1 = E2));`
- C.** `function "=" (A : T1; B : T2) return Boolean is  
    (for some E1 of A => (for some E2 of B => E1 =  
    E2));`
- D.** `function "=" (A : T1; B : T2) return Boolean is  
    (for all J in A'Range => A (J) = B (J));`
- B.** Counterexample: A = B = (0, 1, 0) returns False
- C.** Counterexample: A = (0, 0, 1) and B = (0, 1, 1) returns  
True

# Quiz

```
type Array1_T is array (1 .. 3) of Integer;  
type Array2_T is array (1 .. 3) of Array1_T;  
A : Array2_T;
```

The above describes an array *A* whose components are arrays of three components. Which expression would one use to determine if at least one of *A*'s components are sorted?

- A. (for some *E1* of *A* => (for some *Idx* in 2 .. 3 => *E1* (*Idx*) >= *E1* (*Idx* - 1)));
- B. (for all *E1* of *A* => for all *Idx* in 2 .. 3 => *E1* (*Idx*) >= *E1* (*Idx* - 1)));
- C. (for some *E1* of *A* => (for all *Idx* in 2 .. 3 => *E1* (*Idx*) >= *E1* (*Idx* - 1)));
- D. (for all *E1* of *A* => (for some *Idx* in 2 .. 3 => *E1* (*Idx*) >= *E1* (*Idx* - 1)));

## Quiz

```
type Array1_T is array (1 .. 3) of Integer;  
type Array2_T is array (1 .. 3) of Array1_T;  
A : Array2_T;
```

The above describes an array A whose components are arrays of three components. Which expression would one use to determine if at least one of A's components are sorted?

- A. (for some El of A => (for some Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));
  - B. (for all El of A => for all Idx in 2 .. 3 => El (Idx) >= El (Idx - 1));
  - C. (for some El of A => (for all Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));
  - D. (for all El of A => (for some Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));
- A. Will be True if any component has two consecutive increasing values
  - B. Will be True if every component is sorted
  - C. Correct
  - D. Will be True if every component has two consecutive increasing values

Lab

# Expressions Lab

## ■ Requirements

- Allow the user to fill a list with dates
- After the list is created, use *quantified expressions* to print True/False
  - If any date is not legal (taking into account leap years!)
  - If all dates are in the same calendar year
- Use *expression functions* for all validation routines

## ■ Hints

- Use subtype membership for range validation
- You will need *conditional expressions* in your functions
- You *can* use component-based iterations for some checks
  - But you *must* use indexed-based iterations for others
- This is the same lab as the *Expressions* lab, we're just replacing the validation functions with quantified expressions!
  - So you can just copy that project and update the code!

# Expressions Lab Solution - Checks

```
4  subtype Year_T is Positive range 1_900 .. 2_099;
5  subtype Month_T is Positive range 1 .. 12;
6  subtype Day_T is Positive range 1 .. 31;
7
8  type Date_T is record
9      Year : Positive;
10     Month : Positive;
11     Day : Positive;
12 end record;
13
14 List : array (1 .. 5) of Date_T;
15 Item : Date_T;
16
17 function Is_Leap_Year (Year : Positive)
18     return Boolean is
19     (Year mod 400 = 0 or else (Year mod 4 = 0 and Year mod 100 /= 0));
20
21 function Days_In_Month (Month : Positive;
22     Year : Positive)
23     return Day_T is
24     (case Month is when 4 | 6 | 9 | 11 => 30,
25     when 2 => (if Is_Leap_Year (Year) then 29 else 28), when others => 31);
26
27 function Is_Valid (Date : Date_T)
28     return Boolean is
29     (Date.Year in Year_T and then Date.Month in Month_T
30     and then Date.Day <= Days_In_Month (Date.Month, Date.Year));
31
32 function Any_Invalid return Boolean is
33     (for some Date of List => not Is_Valid (Date));
34
35 function Same_Year return Boolean is
36     (for all I in List'Range => List (I).Year = List (List'First).Year);
```

# Expressions Lab Solution - Main

```
37  function Number (Prompt : String)
38          return Positive is
39  begin
40      Put (Prompt & "> ");
41      return Positive'Value (Get_Line);
42  end Number;
43
44  begin
45
46      for I in List'Range loop
47          Item.Year := Number ("Year");
48          Item.Month := Number ("Month");
49          Item.Day := Number ("Day");
50          List (I) := Item;
51      end loop;
52
53      Put_Line ("Any invalid: " & Boolean'Image (Any_Invalid));
54      Put_Line ("Same Year: " & Boolean'Image (Same_Year));
55
56  end Main;
```

## Summary

# Summary

- Conditional expressions are allowed wherever expressions are allowed, but beware over-use
  - Especially useful when a constant is intended
  - Especially useful when a static expression is required
- Quantified expressions are general purpose but especially useful with pre/postconditions
  - Consider hiding them behind expressive function names

# Overloading

## Introduction

# Introduction

- **Overloading** is the use of an already existing name to define a **new** entity
- Historically, only done as part of the language **implementation**
  - Eg. on operators
  - Float vs Integer vs pointers arithmetic
- Several languages allow **user-defined** overloading
  - C++
  - Python (limited to operators)
  - Haskell

## Visibility and Scope

- Overloading is **not** re-declaration
- Both entities **share** the name
  - No hiding
  - Compiler performs **name resolution**
- Allowed to be declared in the **same scope**
  - Remember this is forbidden for "usual" declarations

## Overloadable Entities in Ada

- Identifiers for subprograms
  - Both procedure and function names
- Identifiers for enumeration values (enumerals)
- Language-defined operators for functions

```
procedure Put (Str : in String);  
procedure Put (C : in Complex);  
function Max (Left, Right : Integer) return Integer;  
function Max (Left, Right : Float) return Float;  
function "+" (Left, Right : Rational) return Rational;  
function "+" (Left, Right : Complex) return Complex;  
function "*" (Left : Natural; Right : Character)  
    return String;
```

## Function Operator Overloading Example

```
-- User-defined overloading
function "+" (L,R : Complex) return Complex is
begin
    return (L.Real_Part + R.Real_Part,
           L.Imaginary + R.Imaginary);
end "+";

A, B, C : Complex;
I, J, K : Integer;

I := J + K; -- overloaded operator (predefined)
A := B + C; -- overloaded operator (user-defined)
```

## Benefits and Risk of Overloading

- Management of the name space
  - Support for abstraction
  - Linker will not simply take the first match and apply it globally
- Safe: compiler will reject ambiguous calls
- Sensible names are the programmer's job

```
function "+" (L, R : Integer) return String is
begin
    return Integer'Image (L - R);
end "+";
```

## Enumerals and Operators

# Overloading Enumerals

- Each is treated as if a function name (identifier)
- Thus same rules as for function identifier overloading

```
type Stop_Light is (Red, Yellow, Green);
```

```
type Colors is (Red, Blue, Green);
```

```
Shade : Colors := Red;
```

```
Current_Value : Stop_Light := Red;
```

# Overloadable Operator Symbols

- Only those defined by the language already
  - Users cannot introduce new operator symbols
- Note that assignment ( $:=$ ) is not an operator
- Operators (in precedence order)

Logicals and, or, xor

Relationals  $<$ ,  $<=$ ,  $=$ ,  $>=$ ,  $>$

Unary  $+$ ,  $-$

Binary  $+$ ,  $-$ ,  $\&$

Multiplying  $*$ ,  $/$ , mod, rem

Highest precedence  $**$ , abs, not

# Parameters for Overloaded Operators

- Must not change syntax of calls
  - Number of parameters must remain same (unary, binary...)
  - No default expressions allowed for operators
- Infix calls use positional parameter associations
  - Left actual goes to first formal, right actual goes to second formal
  - Definition

```
function "*" (Left, Right : Integer) return Integer;
```

- Usage

```
X := 2 * 3;
```

- Named parameter associations allowed but ugly
  - Requires prefix notation for call

```
X := "*" (Left => 2, Right => 3);
```

## Call Resolution

# Call Resolution

- Compilers must reject ambiguous calls
- **Resolution** is based on the calling context
  - Compiler attempts to find a matching **profile**
  - Based on **Parameter** and **Result** Type
- Overloading is not re-definition, or hiding
  - More than one matching profile is ambiguous

```
type Complex is ...  
function "+" (L, R : Complex) return Complex;  
A, B : Complex := some_value;  
C : Complex := A + B;  
D : Float := A + B;  -- illegal!  
E : Float := 1.0 + 2.0;
```

## Profile Components Used

- Significant components appear in the call itself
  - **Number** of parameters
  - **Order** of parameters
  - **Base type** of parameters
  - **Result** type (for functions)
- Insignificant components might not appear at call
  - Formal parameter **names** are optional
  - Formal parameter **modes** never appear
  - Formal parameter **subtypes** never appear
  - **Default** expressions never appear

```
Display (X);
```

```
Display (Foo => X);
```

```
Display (Foo => X, Bar => Y);
```

## Manually Disambiguating Calls

- Qualification can be used
- Named parameter association can be used
  - Unless name is ambiguous

```
type Stop_Light is (Red, Yellow, Green);  
type Colors is (Red, Blue, Green);  
procedure Put (Light : in Stop_Light);  
procedure Put (Shade : in Colors);
```

```
Put (Red); -- ambiguous call
```

```
Put (Yellow); -- not ambiguous: only 1 Yellow
```

```
Put (Colors'(Red)); -- using type to distinguish
```

```
Put (Light => Green); -- using profile to distinguish
```

# Overloading Example

```
function "+" (Left : Position; Right : Offset)
  return Position is
begin
  return Position'(Left.Row + Right.Row, Left.Column + Right.Col);
end "+";
```

```
function Acceptable (P : Position) return Boolean;
type Positions is array (Moves range <>) of Position;
```

```
function Next (Current : Position) return Positions is
  Result : Positions (Moves range 1 .. 4);
  Count  : Moves := 0;
  Test   : Position;
begin
  for K in Offsets'Range loop
    Test := Current + Offsets (K);
    if Acceptable (Test) then
      Count := Count + 1;
      Result (Count) := Test;
    end if;
  end loop;
  return Result (1 .. Count);
end Next;
```

# Quiz

```
type Vertical_T is (Top, Middle, Bottom);  
type Horizontal_T is (Left, Middle, Right);  
function "*" (H : Horizontal_T; V : Vertical_T) return Positive;  
function "*" (V : Vertical_T; H : Horizontal_T) return Positive;  
P : Positive;
```

Which statement(s) is (are) legal?

- A. P := Horizontal\_T'(Middle) \* Middle;
- B. P := Top \* Right;
- C. P := "\*" (Middle, Top);
- D. P := "\*" (H => Middle, V => Top);

# Quiz

```
type Vertical_T is (Top, Middle, Bottom);  
type Horizontal_T is (Left, Middle, Right);  
function "*" (H : Horizontal_T; V : Vertical_T) return Positive;  
function "*" (V : Vertical_T; H : Horizontal_T) return Positive;  
P : Positive;
```

Which statement(s) is (are) legal?

- A. `P := Horizontal_T'(Middle) * Middle;`
- B. `P := Top * Right;`
- C. `P := "*" (Middle, Top);`
- D. `P := "*" (H => Middle, V => Top);`

Explanations

- A. Qualifying one parameter resolves ambiguity
- B. No overloaded names
- C. Use of Top resolves ambiguity
- D. When overloading subprogram names, best to not just switch the order of parameters

## User-Defined Equality

# User-Defined Equality

- Allowed like any other operator
  - Must remain a binary operator
- Typically declared as `return Boolean`
- Hard to do correctly for composed types
  - Especially **user-defined** types
  - Issue of *Composition of equality*

Lab

# Overloading Lab

## ■ Requirements

- Create multiple functions named "Convert" to convert between digits and text representation
  - One routine should take a digit and return the text version (e.g. **3** would return **three**)
  - One routine should take text and return the digit (e.g. **two** would return **2**)
- Query the user to enter text or a digit and print its equivalent
- If the user enters consecutive entries that are equivalent, print a message
  - e.g. **4** followed by **four** should get the message

## ■ Hints

- You can use enumerals for the text representation
  - Then use *'Image* / *'Value* where needed
- Use an equivalence function to compare different types

# Overloading Lab Solution - Conversion Functions

```
4  type Digit_T is range 0 .. 9;
5  type Digit_Name_T is
6      (Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine);
7
8  function Convert (Value : Digit_T) return Digit_Name_T;
9  function Convert (Value : Digit_Name_T) return Digit_T;
10 function Convert (Value : Character) return Digit_Name_T;
11 function Convert (Value : String) return Digit_T;
12
13 function "=" (L : Digit_Name_T; R : Digit_T) return Boolean is (Convert (L) = R);
14
15 function Convert (Value : Digit_T) return Digit_Name_T is
16     (case Value is when 0 => Zero, when 1 => One,
17         when 2 => Two, when 3 => Three,
18         when 4 => Four, when 5 => Five,
19         when 6 => Six, when 7 => Seven,
20         when 8 => Eight, when 9 => Nine);
21
22 function Convert (Value : Digit_Name_T) return Digit_T is
23     (case Value is when Zero => 0, when One => 1,
24         when Two => 2, when Three => 3,
25         when Four => 4, when Five => 5,
26         when Six => 6, when Seven => 7,
27         when Eight => 8, when Nine => 9);
28
29 function Convert (Value : Character) return Digit_Name_T is
30     (case Value is when '0' => Zero, when '1' => One,
31         when '2' => Two, when '3' => Three,
32         when '4' => Four, when '5' => Five,
33         when '6' => Six, when '7' => Seven,
34         when '8' => Eight, when '9' => Nine,
35         when others => Zero);
36
37 function Convert (Value : String) return Digit_T is
38     (Convert (Digit_Name_T'Value (Value)));
```

# Overloading Lab Solution - Main

```
40 Last_Entry : Digit_T := 0;
41
42 begin
43 loop
44 Put ("Input: ");
45 declare
46 Str : constant String := Get_Line;
47 begin
48 exit when Str'Length = 0;
49 if Str (Str'First) in '0' .. '9' then
50 declare
51 Converted : constant Digit_Name_T := Convert (Str (Str'First));
52 begin
53 Put (Digit_Name_T'Image (Converted));
54 if Converted = Last_Entry then
55 Put_Line (" - same as previous");
56 else
57 Last_Entry := Convert (Converted);
58 New_Line;
59 end if;
60 end;
61 else
62 declare
63 Converted : constant Digit_T := Convert (Str);
64 begin
65 Put (Digit_T'Image (Converted));
66 if Converted = Last_Entry then
67 Put_Line (" - same as previous");
68 else
69 Last_Entry := Converted;
70 New_Line;
71 end if;
72 end;
73 end if;
74 end;
75 end loop;
76 end Main;
```

## Summary

# Summary

- Ada allows user-defined overloading
  - Identifiers and operator symbols
- Benefits easily outweigh danger of senseless names
  - Can have nonsensical names without overloading
- Compiler rejects ambiguous calls
- Resolution is based on the calling context
  - *Parameter and Result Type Profile*
- Calling context is those items present at point of call
  - Thus modes etc. don't affect overload resolution
- User-defined equality is allowed
  - But is tricky

# Packages

# Introduction

# Packages

- Enforce separation of client from implementation
  - In terms of compile-time visibility
  - For data
  - For type representation, when combined with `private` types
    - Abstract Data Types
- Provide basic namespace control
- Directly support software engineering principles
  - Especially in combination with `private` types
  - Modularity
  - Information Hiding (Encapsulation)
  - Abstraction
  - Separation of Concerns

# Basic Syntax and Nomenclature

- Spec
  - Basic declarative items **only**
  - e.g. no subprogram bodies

```
package name is  
    {basic_declarative_item}  
end [name];
```

- Body

```
package body name is  
    declarative_part  
end [name];
```

## Separating Interface and Implementation

- *Implementation* and *specification* are textually distinct from each other
  - Typically in separate files
- Clients can compile their code before body exists
  - All they need is the package specification
  - Clients have **no** visibility over the body
  - Full client/interface consistency is guaranteed

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

# Uncontrolled Visibility Problem

- Clients have too much access to representation
  - Data
  - Type representation
- Changes force clients to recompile and retest
- Manual enforcement is not sufficient
- Why fixing bugs introduces new bugs!

## Declarations

# Package Declarations

- Required in all cases
  - Cannot have a package without the declaration
- Describe the client's interface
  - Declarations are exported to clients
  - Effectively the "pin-outs" for the black-box
- When changed, requires clients recompilation
  - The "pin-outs" have changed

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

```
package Data is
  Object : Integer;
end Data;
```

## Compile-Time Visibility Control

- Items in the declaration are visible to users

```
package Some_Package is  
    -- exported declarations of  
    -- types, variables, subprograms ...  
end Some_Package;
```

- Items in the body are never externally visible
  - Compiler prevents external references

```
package body Some_Package is  
    -- hidden declarations of  
    -- types, variables, subprograms ...  
    -- implementations of exported subprograms etc.  
end Some_Package;
```

## Example of Exporting to Clients

- Variables, types, exception, subprograms, etc.
  - The primary reason for separate subprogram declarations

```
package P is
  procedure This_Is_Exported;
end P;

package body P is
  procedure Not_Exported is
    ...
  procedure This_Is_Exported is
    ...
end P;
```

## Referencing Other Packages

## with Clause

- When package Client needs access to package Server, it uses a **with** clause
  - Specify the library units that Client depends upon
  - The "context" in which the unit is compiled
  - Client's code gets **visibility** over Server's specification
- Syntax (simplified)

```
context_clause ::= { context_item }  
context_item  ::= with_clause | use_clause  
with_clause   ::= with library_unit_name  
                { , library_unit_name };
```

```
with Server; -- dependency  
procedure Client is
```

## Referencing Exported Items

- Achieved via "dot notation"
- Package Specification

```
package Float_Stack is
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

- Package Reference

```
with Float_Stack;
procedure Test is
  X : Float;
begin
  Float_Stack.Pop (X);
  Float_Stack.Push (12.0);
```

...

## with Clause Syntax

- A library unit is a package or subprogram that is not nested within another unit
  - Typically in its own file(s)
    - e.g. for package `Test`, GNAT defaults to expect the spec in `test.ads` and body in `test.adb` )
- Only library units may appear in a `with` statement
  - Can be a package or a standalone subprogram
- Due to the `with` syntax, library units cannot be overloaded
  - If overloading allowed, which `P` would `with P;` refer to?

# What To Import

- Need only name direct dependencies
  - Those actually referenced in the corresponding unit
- Will not cause compilation of referenced units
  - Unlike "include directives" of some languages

```
package A is
  type Something is ...
end A;

with A;
package B is
  type Something is record
    Component : A.Something;
  end record;
end B;

with B; -- no "with" of A
procedure Foo is
  X : B.Something;
begin
  X.Component := ...
```

## Bodies

# Package Bodies

- Dependent on corresponding package specification
  - Obsolete if specification changed
- Clients need only to relink if body changed
  - Any code that would require editing would not have compiled in the first place
- Necessary for specifications that require a completion, for example:
  - Subprogram bodies
  - Task bodies
  - Incomplete types in `private` part
  - Others...

## Bodies Are Never Optional

- Either required for a given spec or not allowed at all
  - Based on declarations in that spec
- A change from Ada 83
- A (nasty) justification example will be shown later

## Example Spec That Cannot Have a Body

```
package Graphics_Primitives is
  type Coordinate is digits 12;
  type Device_Coordinates is record
    X, Y : Integer;
  end record;
  type Normalized_Coordinates is record
    X, Y : Coordinate range 0.0 .. 1.0;
  end record;
  type Offset is record
    X, Y : Coordinate range -1.0 .. 1.0;
  end record;
  -- nothing to implement, so no body allowed
end Graphics_Primitives;
```

## Example Spec Requiring a Package Body

```
package VT100 is
  subtype Rows is Integer range 1 .. 24;
  subtype Columns is Integer range 1 .. 80;
  type Position is record
    Row : Rows := Rows'First;
    Col : Columns := Columns'First;
  end record;
  -- The following need to be defined in the body
  procedure Move_Cursor (To : in Position);
  procedure Home;
  procedure Clear_Screen;
  procedure Cursor_Up (Count : in Positive := 1);
end VT100;
```

## Required Body Example

```
package body VT100 is
  -- This function is not visible outside this package
  function Unsigned (Input : Integer) return String is
    Str : constant String := Integer'Image (Input);
  begin
    return Str (2 .. Str'Length);
  end Unsigned;
  procedure Move_Cursor (To : in Position) is
  begin
    Text_IO.Put (ASCII.Esc & 'I' &
                 Unsigned (To.Row) & ';' &
                 Unsigned (To.Col) & 'H');
  end Move_Cursor;
  procedure Home is
  begin
    Text_IO.Put (ASCII.Esc & "iH");
  end Home;
  procedure Cursor_Up (Count : in Positive := 1) is ...
    ...
end VT100;
```

# Quiz

```
package P is
  Object_One : Integer;
  procedure One (V : out Integer);
end P;
```

Which completion(s) is (are) correct for `package P`?

- A No completion is needed
- B 

```
package body P is
  procedure One (V : out Integer) is null;
end P;
```
- C 

```
package body P is
  Object_One : Integer;
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```
- D 

```
package body P is
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```

# Quiz

```
package P is
  Object_One : Integer;
  procedure One (V : out Integer);
end P;
```

Which completion(s) is (are) correct for package P?

- A. No completion is needed
  - B. 

```
package body P is
  procedure One (V : out Integer) is null;
end P;
```
  - C. 

```
package body P is
  Object_One : Integer;
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```
  - D. 

```
package body P is
  procedure One (V : out Integer) is
  begin
    V := Object_One;
  end One;
end P;
```
- A. Procedure One must have a body
- B. Parameter V is `out` but not assigned (legal but not a good idea)
- C. Redeclaration of Object\_One
- D. Correct

## Executable Parts

## Optional Executable Part

```
package_body ::=  
    package body name is  
        declarative_part  
    [ begin  
        handled_sequence_of_statements ]  
end [ name ];
```

## Executable Part Semantics

- Executed only once, when package is elaborated
- Ideal when statements are required for initialization
  - Otherwise initial values in variable declarations would suffice

```
package body Random is
  Seed1, Seed2 : Integer;
  Call_Count : Natural := 0;
  procedure Initialize (Seed1 : out Integer;
                      Seed2 : out Integer) is ...
  function Number return Float is ...
begin -- Random
  Initialize (Seed1, Seed2);
end Random;
```

## Requiring/Rejecting Bodies Justification

- Consider the alternative: an optional package body that becomes obsolete prior to building
- Builder could silently choose not to include the package in executable
  - Package executable part might do critical initialization!

```
package P is
    Data : array (L .. U) of
        Integer;
end P;

package body P is
    ...
begin
    for K in Data'Range loop
        Data (K) := ...
    end loop;
end P;
```

## Forcing a Package Body to Be Required

- Use `pragma Elaborate_Body`
  - Says to elaborate body immediately after spec
  - Hence there must be a body!
- Additional pragmas we will examine later

```
package P is
    pragma Elaborate_Body;
    Data : array (L .. U) of
        Integer;
end P;

package body P is
    ...
begin
    for K in Data'Range loop
        Data (K) := ...
    end loop;
end P;
```

## Idioms

## Named Collection of Declarations

- Exports:
  - Objects (constants and variables)
  - Types
  - Exceptions
- Does not export operations

```
package Physical_Constants is
  Polar_Radius_in_feet      : constant := 20_856_010.51;
  Equatorial_Radius_in_feet : constant := 20_926_469.20;
  Earth_Diameter_in_feet   : constant := 2.0 *
    ((Polar_Radius_in_feet + Equatorial_Radius_in_feet)/2.0);
  Sea_Level_Air_Density    : constant := 0.00239; --slugs/foot**3
  Altitude_Of_Tropopause_in_feet : constant := 36089.0;
  Tropopause_Temperature_in_celsius : constant := -56.5;
end Physical_Constants;
```

## Named Collection of Declarations (2)

- Effectively application global data

```
package Equations_of_Motion is
  Longitudinal_Velocity : Float := 0.0;
  Longitudinal_Acceleration : Float := 0.0;
  Lateral_Velocity : Float := 0.0;
  Lateral_Acceleration : Float := 0.0;
  Vertical_Velocity : Float := 0.0;
  Vertical_Acceleration : Float := 0.0;
  Pitch_Attitude : Float := 0.0;
  Pitch_Rate : Float := 0.0;
  Pitch_Acceleration : Float := 0.0;
end Equations_of_Motion;
```

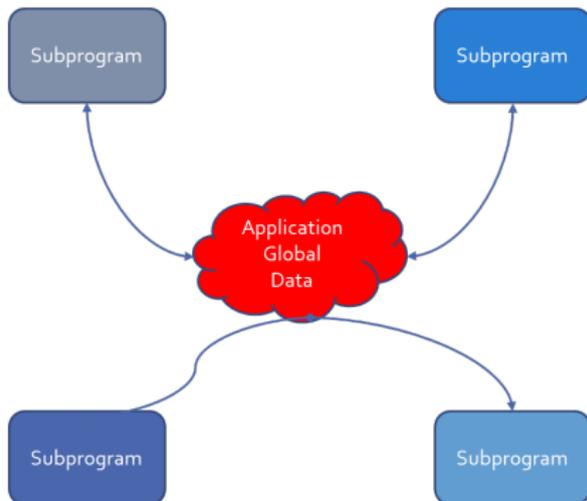
## Group of Related Program Units

- Exports:
  - Objects
  - Types
  - Values
  - Operations
- Users have full access to type representations
  - This visibility may be necessary

```
package Linear_Algebra is
  type Vector is array (Positive range <>) of Float;
  function "+" (L,R : Vector) return Vector;
  function "*" (L,R : Vector) return Vector;
  ...
end Linear_Algebra;
```

# Uncontrolled Data Visibility Problem

- Effects of changes are potentially pervasive so one must understand everything before changing anything

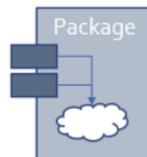
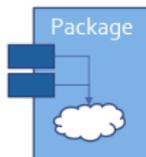
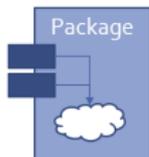


## Packages and "Lifetime"

- Like a subprogram, objects declared directly in a package exist while the package is "in scope"
  - Whether the object is in the package spec or body
- Packages defined at the library level (not inside a subprogram) are always "in scope"
  - Including packages nested inside a package
- So package objects are considered "global data"
  - Putting variables in the spec exposes them to clients
    - Usually - in another module we talk about data hiding in the spec
  - Variables in the body can only be accessed from within the package body

## Controlling Data Visibility Using Packages

- Divides global data into separate package bodies
- Visible only to procedures and functions declared in those same packages
  - Clients can only call these visible routines
- Global change effects are much less likely
  - Direct breakage is impossible



# Abstract Data Machines

- Exports:
  - Operations
  - State information queries (optional)
- No direct user access to data

```
package Float_Stack is
```

```
  Max : constant := 100;
```

```
  procedure Push (X : in Float);
```

```
  procedure Pop (X : out Float);
```

```
end Float_Stack;
```

```
package body Float_Stack is
```

```
  type Contents is array (1 .. Max) of Float;
```

```
  Values : Contents;
```

```
  Top : Integer range 0 .. Max := 0;
```

```
  procedure Push (X : in Float) is ...
```

```
  procedure Pop (X : out Float) is ...
```

```
end Float_Stack;
```

## Controlling Type Representation Visibility

- In other words, support for Abstract Data Types
  - No operations visible to clients based on representation
- The fundamental concept for Ada
- Requires `private` types discussed in coming section...

# Lab

# Packages Lab

## ■ Requirements

- Create a program to add and remove integer values from a list
- Program should allow user to do the following as many times as desired
  - Add an integer in a pre-defined range to the list
  - Remove all occurrences of an integer from the list
  - Print the values in the list

## ■ Hints

- Create (at least) three packages
  - 1 minimum/maximum integer values and maximum number of items in list
  - 2 User input (ensure value is in range)
  - 3 List Abstract Data Machine
- Remember: with `package_name`; gives access to `package_name`

## Creating Packages in GNAT STUDIO

- Right-click on the source directory node
  - If you used a prompt, the directory is probably .
  - If you used the wizard, the directory is probably **src**
- **New** → **Ada Package**
  - Fill in name of Ada package
  - Check the box if you want to create the package body in addition to the package spec

## Packages Lab Solution - Constants

```
1 package Constants is
2
3     Lowest_Value   : constant := 100;
4     Highest_Value  : constant := 999;
5     Maximum_Count  : constant := 10;
6     subtype Integer_T is Integer
7         range Lowest_Value .. Highest_Value;
8
9 end Constants;
```

# Packages Lab Solution - Input

```
1  with Constants;
2  package Input is
3      function Get_Value (Prompt : String) return Constants.Integer_T;
4  end Input;
5
6  with Ada.Text_IO; use Ada.Text_IO;
7  package body Input is
8
9      function Get_Value (Prompt : String) return Constants.Integer_T is
10         Ret_Val : Integer;
11     begin
12         Put (Prompt & "> ");
13         loop
14             Ret_Val := Integer'Value (Get_Line);
15             exit when Ret_Val >= Constants.Lowest_Value
16                 and then Ret_Val <= Constants.Highest_Value;
17             Put ("Invalid. Try Again >");
18         end loop;
19         return Ret_Val;
20     end Get_Value;
21
22 end Input;
```

# Packages Lab Solution - List

```
1 package List is
2   procedure Add (Value : Integer);
3   procedure Remove (Value : Integer);
4   function Length return Natural;
5   procedure Print;
6 end List;
7
8 with Ada.Text_IO; use Ada.Text_IO;
9 with Constants;
10 package body List is
11   Content : array (1 .. Constants.Maximum_Count) of Integer;
12   Last : Natural := 0;
13
14   procedure Add (Value : Integer) is
15   begin
16     if Last < Content'Last then
17       Last := Last + 1;
18       Content (Last) := Value;
19     else
20       Put_Line ("Full");
21     end if;
22   end Add;
23
24   procedure Remove (Value : Integer) is
25   I : Natural := 1;
26   begin
27     while I <= Last loop
28       if Content (I) = Value then
29         Content (I .. Last - 1) := Content (I + 1 .. Last);
30         Last := Last - 1;
31       else
32         I := I + 1;
33       end if;
34     end loop;
35   end Remove;
36
37   procedure Print is
38   begin
39     for I in 1 .. Last loop
40       Put_Line (Integer'Image (Content (I)));
41     end loop;
42   end Print;
43
44   function Length return Natural is (Last);
45 end List;
```

# Packages Lab Solution - Main

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Input;
3 with List;
4 procedure Main is
5
6 begin
7
8     loop
9         Put ("(A)dd | (R)emove | (P)rint | (Q)uit : ");
10        declare
11            Str : constant String := Get_Line;
12        begin
13            exit when Str'Length = 0;
14            case Str (Str'First) is
15                when 'A' =>
16                    List.Add (Input.Get_Value ("Value to add"));
17                when 'R' =>
18                    List.Remove (Input.Get_Value ("Value to remove"));
19                when 'P' =>
20                    List.Print;
21                when 'Q' =>
22                    exit;
23                when others =>
24                    Put_Line ("Illegal entry");
25            end case;
26        end;
27    end loop;
28
29 end Main;
```

## Summary

# Summary

- Emphasizes separations of concerns
- Solves the global visibility problem
  - Only those items in the specification are exported
- Enforces software engineering principles
  - Information hiding
  - Abstraction
- Implementation can't be corrupted by clients
  - Compiler won't let clients compile references to internals
- Bugs must be in the implementation, not clients
  - Only body implementation code has to be understood

# Private Types

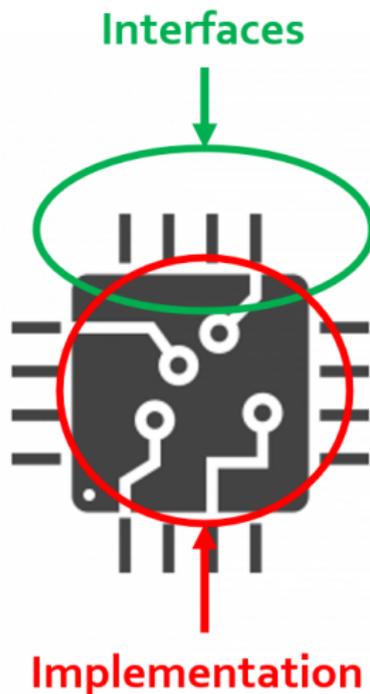
## Introduction

# Introduction

- Why does fixing bugs introduce new ones?
- Control over visibility is a primary factor
  - Changes to an abstraction's internals shouldn't break users
  - Including type representation
- Need tool-enforced rules to isolate dependencies
  - Between implementations of abstractions and their users
  - In other words, "information hiding"

# Information Hiding

- A design technique in which implementation artifacts are made inaccessible to users
- Based on control of visibility to those artifacts
  - A product of "encapsulation"
  - Language support provides rigor
- Concept is "software integrated circuits"



# Views

- Specify legal manipulation for objects of a type
  - Types are characterized by permitted values and operations
- Some views are implicit in language
  - Mode `in` parameters have a view disallowing assignment
- Views may be explicitly specified
  - Disallowing access to representation
  - Disallowing assignment
- Purpose: control usage in accordance with design
  - Adherence to interface
  - Abstract Data Types

## Implementing Abstract Data Types Via Views

# Implementing Abstract Data Types

- A combination of constructs in Ada
- Not based on single "class" construct, for example
- Constituent parts
  - Packages, with "private part" of package spec
  - "Private types" declared in packages
  - Subprograms declared within those packages

## Package Visible and Private Parts for Views

- Declarations in visible part are exported to users
- Declarations in private part are hidden from users
  - No compilable references to type's actual representation

```
package name is
... exported declarations of types, variables, subprograms .
private
... hidden declarations of types, variables, subprograms ...
end name;
```

# Declaring Private Types for Views

- Partial syntax

```
type defining_identifier is private;
```

- Private type declaration must occur in visible part

- *Partial view*

- Only partial information on the type

- Users can reference the type name

- But cannot create an object of that type until after the full type declaration

- Full type declaration must appear in private part

- Completion is the *Full view*

- **Never** visible to users

- **Not** visible to designer until reached

```
package Bounded_Stacks is
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  ...
private
  ...
  type Stack is record
    Top : Positive;
    ...
end Bounded_Stacks;
```

## Partial and Full Views of Types

- Private type declaration defines a *partial view*
  - The type name is visible
  - Only designer's operations and some predefined operations
  - No references to full type representation
- Full type declaration defines the *full view*
  - Fully defined as a record type, scalar, imported type, etc...
  - Just an ordinary type within the package
- Operations available depend upon one's view

# Software Engineering Principles

- Encapsulation and abstraction enforced by views
  - Compiler enforces view effects
- Same protection as hiding in a package body
  - Recall "Abstract Data Machines" idiom
- Additional flexibility of types
  - Unlimited number of objects possible
  - Passed as parameters
  - Components of array and record types
  - Dynamically allocated
  - et cetera

## Users Declare Objects of the Type

- Unlike "abstract data machine" approach
- Hence must specify which stack to manipulate
  - Via parameter

```
X, Y, Z : Bounded_Stacks.Stack;
```

```
...
```

```
Push (42, X);
```

```
...
```

```
if Empty (Y) then
```

```
...
```

```
Pop (Counter, Z);
```

## Compile-Time Visibility Protection

- No type representation details available outside the package
- Therefore users cannot compile code referencing representation
- This does not compile

```
with Bounded_Stacks;  
procedure User is  
  S : Bounded_Stacks.Stack;  
begin  
  S.Top := 1;  -- Top is not visible  
end User;
```

## Benefits of Views

- Users depend only on visible part of specification
  - Impossible for users to compile references to private part
  - Physically seeing private part in source code is irrelevant
- Changes to implementation don't affect users
  - No editing changes necessary for user code
- Implementers can create bullet-proof abstractions
  - If a facility isn't working, you know where to look
- Fixing bugs is less likely to introduce new ones

# Quiz

```
package P is
  type Private_T is private;

  type Record_T is record
```

Which component(s) is (are) legal?

- A. `Component_A : Integer := Private_T'Pos  
(Private_T'First);`
- B. `Component_B : Private_T := null;`
- C. `Component_C : Private_T := 0;`
- D. `Component_D : Integer := Private_T'Size;  
end record;`

# Quiz

```
package P is
  type Private_T is private;

  type Record_T is record
```

Which component(s) is (are) legal?

- A. `Component_A : Integer := Private_T'Pos (Private_T'First);`
- B. `Component_B : Private_T := null;`
- C. `Component_C : Private_T := 0;`
- D. `Component_D : Integer := Private_T'Size;`  
`end record;`

Explanations

- A. Visible part does not know `Private_T` is discrete
- B. Visible part does not know possible values for `Private_T`
- C. Visible part does not know possible values for `Private_T`
- D. Correct - type will have a known size at run-time

## Private Part Construction

## Private Part and Recompile

- Users can compile their code before the package body is compiled or even written
- Private part is part of the specification
  - Compiler needs info from private part for users' code, e.g., storage layouts for private-typed objects
- Thus changes to private part require user recompilation
- Some vendors avoid "unnecessary" recompilation
  - Comment additions or changes
  - Additions which nobody yet references

# Declarative Regions

- Declarative region of the spec extends to the body
  - Anything declared there is visible from that point down
  - Thus anything declared in specification is visible in body

```
package Foo is
  type Private_T is private;
  procedure X (B : in out Private_T);
private
  -- Y and Hidden_T are not visible to users
  procedure Y (B : in out Private_T);
  type Hidden_T is ...;
  type Private_T is array (1 .. 3) of Hidden_T;
end Foo;
```

```
package body Foo is
  -- Z is not visible to users
  procedure Z (B : in out Private_T) is ...
  procedure Y (B : in out Private_T) is ...
  procedure X (B : in out Private_T) is ...
end Foo;
```

# Full Type Declaration

- May be any type
  - Predefined or user-defined
  - Including references to imported types
- Contents of private part are unrestricted
  - Anything a package specification may contain
  - Types, subprograms, variables, etc.

```
package P is
  type T is private;
  ...
private
  type Vector is array (1.. 10)
    of Integer;
  function Initial
    return Vector;
  type T is record
    A, B : Vector := Initial;
  end record;
end P;
```

# Deferred Constants

- Visible constants of a hidden representation
  - Value is "deferred" to private part
  - Value must be provided in private part
- Not just for private types, but usually so

```
package P is
  type Set is private;
  Null_Set : constant Set; -- exported name
  ...
private
  type Index is range ...
  type Set is array (Index) of Boolean;
  Null_Set : constant Set := -- definition
    (others => False);
end P;
```

# Quiz

```
package P is
  type Private_T is private;
  Object_A : Private_T;
  procedure Proc (Param : in out Private_T);
private
  type Private_T is new Integer;
  Object_B : Private_T;
end package P;

package body P is
  Object_C : Private_T;
  procedure Proc (Param : in out Private_T) is null;
end P;
```

Which object definition(s) is (are) legal?

- A. Object\_A
- B. Object\_B
- C. Object\_C
- D. None of the above

# Quiz

```
package P is
  type Private_T is private;
  Object_A : Private_T;
  procedure Proc (Param : in out Private_T);
private
  type Private_T is new Integer;
  Object_B : Private_T;
end package P;

package body P is
  Object_C : Private_T;
  procedure Proc (Param : in out Private_T) is null;
end P;
```

Which object definition(s) is (are) legal?

- A. Object\_A
- B. *Object\_B*
- C. *Object\_C*
- D. None of the above

An object cannot be declared until its type is fully declared. `Object_A` could be declared constant, but then it would have to be finalized in the `private` section.

## View Operations

# View Operations

- Reminder: view is the *interface* you have on the type
- **User** of package has **Partial** view
  - Operations **exported** by package
- **Designer** of package has **Full** view
  - **Once** completion is reached
  - All operations based upon **full definition** of type

## Users Have the Partial View

- Since they are outside package
- Basic operations
- Exported subprograms

```
package Bounded_Stacks is
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  procedure Pop (Item : out Integer; From : in out Stack);
  function Empty (S : Stack) return Boolean;
  procedure Clear (S : in out Stack);
  function Top (S : Stack) return Integer;
private
  ...
end Bounded_Stacks;
```

# User View's Activities

- Declarations of objects
  - Constants and variables
  - Must call designer's functions for values

```
C : Complex.Number := Complex.I;
```

- Assignment, equality and inequality, conversions
- Designer's declared subprograms
- User-declared subprograms
  - Using parameters of the exported private type
  - Dependent on designer's operations

## User View Formal Parameters

- Dependent on designer's operations for manipulation
  - Cannot reference type's representation
- Can have default expressions of private types

*-- external implementation of "Top"*

```
procedure Get_Top (  
    The_Stack : in out Bounded_Stacks.Stack;  
    Value : out Integer) is  
    Local : Integer;  
begin  
    Bounded_Stacks.Pop (Local, The_Stack);  
    Value := Local;  
    Bounded_Stacks.Push (Local, The_Stack);  
end Get_Top;
```

# Limited Private

- `limited` is itself a view
  - Cannot perform assignment, copy, or equality
- `limited private` can restrain user's operation
  - Actual type **does not** need to be `limited`

```
package UART is
  type Instance is limited private;
  function Get_Next_Available return Instance;
  [...]
declare
  A, B : UART.Instance := UART.Get_Next_Available;
begin
  if A = B -- Illegal
  then
    A := B; -- Illegal
  end if;
```

## When to Use or Avoid Private Types

## When to Use Private Types

- Implementation may change
  - Allows users to be unaffected by changes in representation
- Normally available operations do not "make sense"
  - Normally available based upon type's representation
  - Determined by intent of ADT

```
A : Valve;
```

```
B : Valve;
```

```
C : Valve;
```

```
...
```

```
C := A + B;  -- addition not meaningful
```

- Users have no "need to know"
  - Based upon expected usage

## When to Avoid Private Types

- If the abstraction is too simple to justify the effort
  - But that's the thinking that led to Y2K rework
- If normal user interface requires representation-specific operations that cannot be provided
  - Those that cannot be redefined by programmers
  - Would otherwise be hidden by a private type
  - If **Vector** is private, indexing of components is annoying

```
type Vector is array (Positive range <>) of Float;  
V : Vector (1 .. 3);  
...  
V (1) := Alpha; -- Illegal since Vector is private
```

## Idioms

## Effects of Hiding Type Representation

- Makes users independent of representation
  - Changes cannot require users to alter their code
  - Software engineering is all about money...
- Makes users dependent upon exported operations
  - Because operations requiring representation info are not available to users
    - Expression of values (aggregates, etc.)
    - Assignment for limited types
- Common idioms are a result
  - *Constructor*
  - *Selector*

# Constructors

- Create designer's objects from user's values
- Usually functions

```
package Complex is
  type Number is private;
  function Make (Real_Part : Float; Imaginary : Float) return Number;
private
  type Number is record ...
end Complex;
```

```
package body Complex is
  function Make (Real_Part : Float; Imaginary_Part : Float)
    return Number is ...
end Complex:
...
A : Complex.Number :=
  Complex.Make (Real_Part => 2.5, Imaginary => 1.0);
```

# Procedures As Constructors

- Spec

```
package Complex is
  type Number is private;
  procedure Make (This : out Number; Real_Part, Imaginary : in Float) ;
  ...
private
  type Number is record
    Real_Part, Imaginary : Float;
  end record;
end Complex;
```

- Body (partial)

```
package body Complex is
  procedure Make (This : out Number;
                 Real_Part, Imaginary : in Float) is
  begin
    This.Real_Part := Real_Part;
    This.Imaginary := Imaginary;
  end Make;
  ...
```

# Selectors

- Decompose designer's objects into user's values
- Usually functions

```
package Complex is
  type Number is private;
  function Real_Part (This: Number) return Float;
  ...
private
  type Number is record
    Real_Part, Imaginary : Float;
  end record;
end Complex;

package body Complex is
  function Real_Part (This : Number) return Float is
  begin
    return This.Real_Part;
  end Real_Part;
  ...
end Complex;

...
Phase : Complex.Number := Complex.Make (10.0, 5.5);
Object : Float := Complex.Real_Part (Phase);
```

Lab

# Private Types Lab

## ■ Requirements

- Implement a program to create a map such that
  - Map key is a description of a flag
  - Map component content is the set of colors in the flag
- Operations on the map should include: Add, Remove, Modify, Get, Exists, Image
- Main program should print out the entire map before exiting

## ■ Hints

- Should implement a **map** ADT (to keep track of the flags)
  - This **map** will contain all the flags and their color descriptions
- Should implement a **set** ADT (to keep track of the colors)
  - This **set** will be the description of the map component
- Each ADT should be its own package
- At a minimum, the **map** and **set** type should be **private**

# Private Types Lab Solution - Color Set

```
1 package Colors is
2   type Color_T is (Red, Yellow, Green, Blue, Black);
3   type Color_Set_T is private;
4
5   Empty_Set : constant Color_Set_T;
6
7   procedure Add (Set : in out Color_Set_T;
8                 Color : Color_T);
9   procedure Remove (Set : in out Color_Set_T;
10                   Color : Color_T);
11   function Image (Set : Color_Set_T) return String;
12 private
13   type Color_Set_Array_T is array (Color_T) of Boolean;
14   type Color_Set_T is record
15     Values : Color_Set_Array_T := (others => False);
16   end record;
17   Empty_Set : constant Color_Set_T := (Values => (others => False));
18 end Colors;
19
20 package body Colors is
21   procedure Add (Set : in out Color_Set_T;
22                 Color : Color_T) is
23   begin
24     Set.Values (Color) := True;
25   end Add;
26   procedure Remove (Set : in out Color_Set_T;
27                     Color : Color_T) is
28   begin
29     Set.Values (Color) := False;
30   end Remove;
31
32   function Image (Set : Color_Set_T;
33                  First : Color_T;
34                  Last : Color_T)
35     return String is
36     Str : constant String := (if Set.Values (First) then Color_T'Image (First) else "");
37   begin
38     if First = Last then
39       return Str;
40     else
41       return Str & " " & Image (Set, Color_T'Succ (First), Last);
42     end if;
43   end Image;
44   function Image (Set : Color_Set_T) return String is
45     (Image (Set, Color_T'First, Color_T'Last));
46 end Colors;
```

# Private Types Lab Solution - Flag Map (Spec)

```
1 with Colors;
2 package Flags is
3   type Key_T is (USA, England, France, Italy);
4   type Map_Component_T is private;
5   type Map_T is private;
6
7   procedure Add (Map      : in out Map_T;
8                 Key       : Key_T;
9                 Description : Colors.Color_Set_T;
10                Success   : out Boolean);
11  procedure Remove (Map : in out Map_T;
12                  Key   : Key_T;
13                  Success : out Boolean);
14  procedure Modify (Map : in out Map_T;
15                  Key   : Key_T;
16                  Description : Colors.Color_Set_T;
17                  Success   : out Boolean);
18
19  function Exists (Map : Map_T; Key : Key_T) return Boolean;
20  function Get (Map : Map_T; Key : Key_T) return Map_Component_T;
21  function Image (Item : Map_Component_T) return String;
22  function Image (Flag : Map_T) return String;
23 private
24  type Map_Component_T is record
25    Key       : Key_T := Key_T'First;
26    Description : Colors.Color_Set_T := Colors.Empty_Set;
27  end record;
28  type Map_Array_T is array (1 .. 100) of Map_Component_T;
29  type Map_T is record
30    Values : Map_Array_T;
31    Length : Natural := 0;
32  end record;
33 end Flags;
```

## Private Types Lab Solution - Flag Map (Body - 1 of 2)

```
3  function Find (Map : Map_T;  
4      Key : Key_T)  
5      return Integer is  
6  begin  
7      for I in 1 .. Map.Length loop  
8          if Map.Values (I).Key = Key then  
9              return I;  
10         end if;  
11     end loop;  
12     return -1;  
13 end Find;  
14  
15 procedure Add (Map           : in out Map_T;  
16     Key           : Key_T;  
17     Description   : Colors.Color_Set_T;  
18     Success       : out Boolean) is  
19     Index : constant Integer := Find (Map, Key);  
20 begin  
21     Success := False;  
22     if Index not in Map.Values'Range then  
23         declare  
24             New_Item : constant Map_Component_T :=  
25                 (Key           => Key,  
26                  Description => Description);  
27         begin  
28             Map.Length           := Map.Length + 1;  
29             Map.Values (Map.Length) := New_Item;  
30             Success              := True;  
31         end;  
32     end if;  
33 end Add;  
34  
35 procedure Remove (Map           : in out Map_T;  
36     Key           : Key_T;  
37     Success       : out Boolean) is  
38     Index : constant Integer := Find (Map, Key);  
39 begin  
40     Success := False;  
41     if Index in Map.Values'Range then  
42         Map.Values (Index .. Map.Length - 1) :=  
43             Map.Values (Index + 1 .. Map.Length);  
44         Success := True;  
45     end if;  
46 end Remove;
```

## Private Types Lab Solution - Flag Map (Body - 2 of 2)

```

35 procedure Modify (Map           : in out Map_T;
36                 Key            : Key_T;
37                 Description    : Colors.Color_Set_T;
38                 Success        : out Boolean) is
39     Index : constant Integer := Find (Map, Key);
40 begin
41     Success := False;
42     if Index in Map.Values'Range then
43         Map.Values (Index).Description := Description;
44         Success                        := True;
45     end if;
46 end Modify;
47
48 function Exists (Map : Map_T;
49                Key  : Key_T)
50     return Boolean is
51     (Find (Map, Key) in Map.Values'Range);
52
53 function Get (Map : Map_T;
54             Key  : Key_T)
55     return Map_Component_T is
56     Index : constant Integer := Find (Map, Key);
57     Ret_Val : Map_Component_T;
58 begin
59     if Index in Map.Values'Range then
60         Ret_Val := Map.Values (Index);
61     end if;
62     return Ret_Val;
63 end Get;
64
65 function Image (Item : Map_Component_T) return String is
66     (Item.Key'Image & " => " & Colors.Image (Item.Description));
67
68 function Image (Flag : Map_T) return String is
69     Ret_Val : String (1 .. 1_000);
70     Next    : Integer := Ret_Val'First;
71 begin
72     for I in 1 .. Flag.Length loop
73         declares
74             Item : constant Map_Component_T := Flag.Values (I);
75             Str  : constant String         := Image (Item);
76         begin
77             Ret_Val (Next .. Next + Str'Length) := Image (Item) & ASCII.LF;
78             Next                               := Next + Str'Length + 1;
79         end;
80     end loop;
81     return Ret_Val (1 .. Next - 1);
82 end Image;

```

# Private Types Lab Solution - Main

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;
3 with Flags;
4 with Input;
5 procedure Main is
6   Map : Flags.Map_T;
7 begin
8
9   loop
10    Put ("Enter country name (");
11    for Key in Flags.Key_T loop
12      Put (Flags.Key_T'Image (Key) & " ");
13    end loop;
14    Put ("): ");
15    declare
16      Str      : constant String := Get_Line;
17      Key      : Flags.Key_T;
18      Description : Colors.Color_Set_T;
19      Success   : Boolean;
20    begin
21      exit when Str'Length = 0;
22      Key      := Flags.Key_T'Value (Str);
23      Description := Input.Get;
24      if Flags.Exists (Map, Key) then
25        Flags.Modify (Map, Key, Description, Success);
26      else
27        Flags.Add (Map, Key, Description, Success);
28      end if;
29    end;
30  end loop;
31
32  Put_Line (Flags.Image (Map));
33 end Main;
```

## Summary

# Summary

- Tool-enforced support for Abstract Data Types
  - Same protection as Abstract Data Machine idiom
  - Capabilities and flexibility of types
- May also be **limited**
  - Thus additionally no assignment or predefined equality
  - More on this later
- Common interface design idioms have arisen
  - Resulting from representation independence
- Assume private types as initial design choice
  - Change is inevitable

# Program Structure

# Introduction

# Introduction

- Moving to "bigger" issues of overall program composition
- How to compose programs out of program units
- How to control object lifetimes
- How to define subsystems

## Building a System

# What Is a System?

- Also called Application or Program or ...
- Collection of *library units*
  - Which are a collection of packages or subprograms

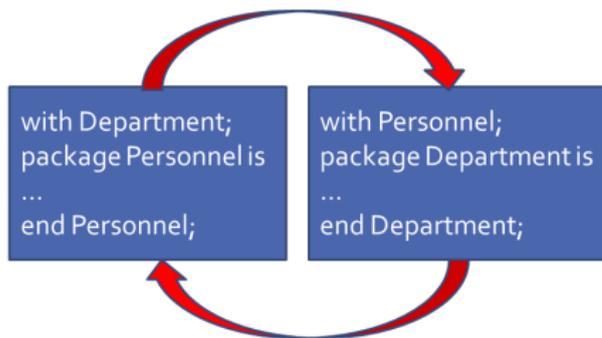
# Library Units Review

- Those units not nested within another program unit
- Candidates
  - Subprograms
  - Packages
  - Generic Units
  - Generic Instantiations
  - Renamings
- Dependencies between library units via **with** clauses
  - What happens when two units need to depend on each other?

## Circular Dependencies

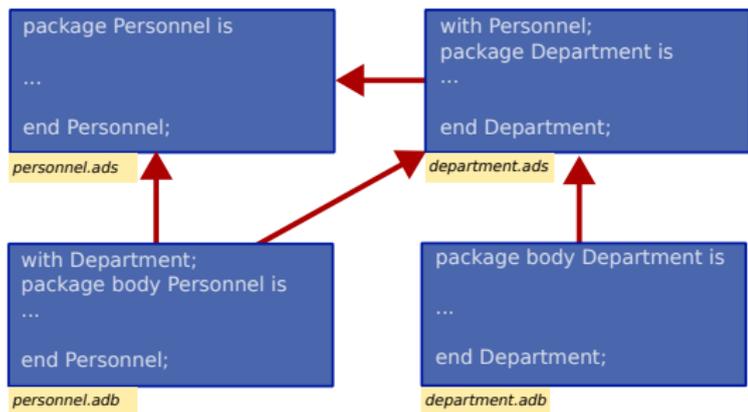
## Handling Cyclic Dependencies

- Elaboration must be linear
- Package declarations cannot depend on each other
  - No linear order is possible
- Which package elaborates first?



## Body-Level Cross Dependencies Are OK

- The bodies only depend on other packages' declarations
- The declarations are already elaborated by the time the bodies are elaborated



## Resulting Design Problem

- Good design dictates that conceptually distinct types appear in distinct package declarations
  - Separation of concerns
  - High level of *cohesion*
- Not possible if they depend on each other
- One solution is to combine them in one package, even though conceptually distinct
  - Poor software engineering
  - May be only choice, depending on language version
    - Best choice would be to implement both parts in a new package

## Circular Dependency in Package Declaration

```
with Department; -- Circular dependency
package Personnel is
  type Employee is private;
  procedure Assign (This : in Employee;
                   To : in out Department.Section);
private
  type Employee is record
    Assigned_To : Department.Section;
  end record;
end Personnel;

with Personnel; -- Circular dependency
package Department is
  type Section is private;
  procedure Choose_Manager (This : in out Section;
                           Who : in Personnel.Employee);
[...]
```

```
end Department;
```

# limited with Clauses

- Solve the cyclic declaration dependency problem
  - Controlled cycles are now permitted
- Provide a *limited view* of the specified package
  - Only type names are visible (including in nested packages)
  - Types are viewed as an *incomplete type*
- Normal view

```
package Personnel is
  type Employee is private;
  procedure Assign ...
private
  type Employee is ...
end Personnel;
```

- Implied limited view

```
package Personnel is
  type Employee;
end Personnel;
```

## Using Incomplete Types

- A type is *incomplete* when its representation is completely unknown
  - Address can still be manipulated through an **access**
  - Can be a formal parameter or function result's type
    - Subprogram's completion needs the complete type
    - Actual parameter needs the complete type
  - Can be a generic formal type parameters
  - If **tagged**, may also use **'Class**

**type** T;

- Can be declared in a **private** part of a package
  - And completed in its body
  - Used to implement opaque pointers
- Thus typically involves some advanced features

# Legal Package Declaration Dependency

```
with Department;
package Personnel is
  type Employee is private;
  procedure Assign (This : in Employee;
                   To : in out Department.Section);
private
  type Employee is record
    Assigned_To : Department.Section;
  end record;
end Personnel;

limited with Personnel;
package Department is
  type Section is private;
  procedure Choose_Manager (This : in out Section;
                           Who : in Personnel.Employee);
private
  type Section is record
    Manager : access Personnel.Employee;
  end record;
end Department;
```

## Full **with** Clause on the Package Body

- Even though declaration has a **limited with** clause
- Typically necessary since body does the work
  - Dereferencing, etc.
- Usual semantics from then on

```
limited with Personnel;  
package Department is
```

```
...
```

```
end Department;
```

```
with Personnel; -- normal view in body  
package body Department is
```

```
...
```

```
end Department;
```

## Hierarchical Library Units

## Problem: Packages Are Not Enough

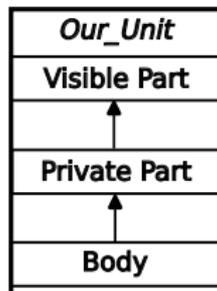
- Extensibility is a problem for private types
  - Provide excellent encapsulation and abstraction
  - But one has either complete visibility or essentially none
  - New functionality must be added to same package for sake of compile-time visibility to representation
  - Thus enhancements require editing/recompilation/retesting
- Should be something "bigger" than packages
  - Subsystems
  - Directly relating library items in one name-space
    - One big package has too many disadvantages
  - Avoiding name clashes among independently-developed code

## Solution: Hierarchical Library Units

- Address extensibility issue
  - Can extend packages with visibility to parent private part
  - Extensions do not require recompilation of parent unit
  - Visibility of parent's private part is protected
- Directly support subsystems
  - Extensions all have the same ancestor *root* name

# Visibility Across a Hierarchy

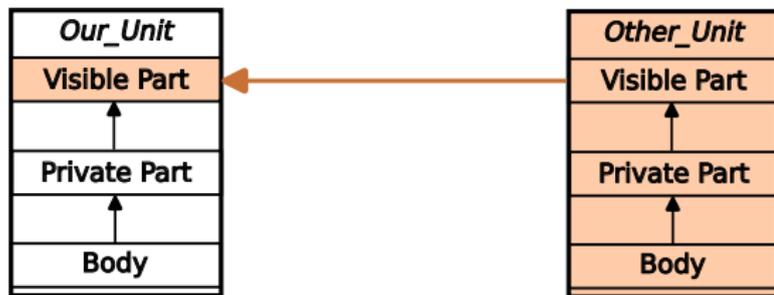
In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.



# Visibility Across a Hierarchy

In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.

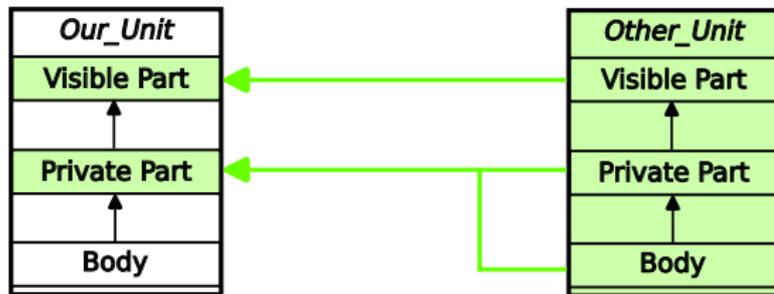
Another **package** can see our **visible part** (depending on where the "with" is), but nothing else.



# Visibility Across a Hierarchy

In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.

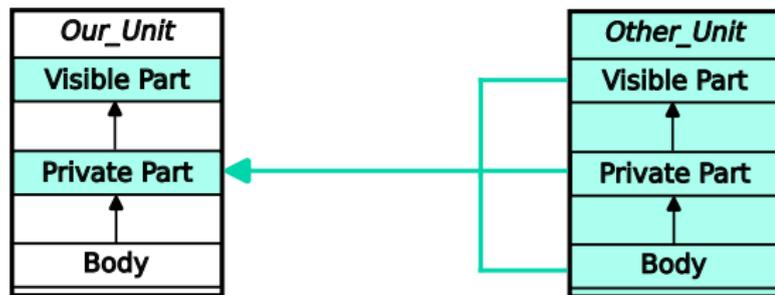
Our **child's visible part** can see our **visible part**, and its **private part** (and **body**) can see our **private part**



# Visibility Across a Hierarchy

In a **package**, the **body** sees everything the **private part** sees, and the **private part** sees everything the **visible part** sees.

Our **private child** can see our private part and **visible part** from anywhere



# Programming by Extension

- *Parent unit*

```
package Complex is
  type Number is private;
  function "*" (Left, Right : Number) return Number;
  function "/" (Left, Right : Number) return Number;
  function "+" (Left, Right : Number) return Number;
  function "-" (Left, Right : Number) return Number;
  ...
private
  type Number is record
    Real_Part, Imaginary_Part : Float;
  end record;
end Complex;
```

- Extension created to work with parent unit

```
package Complex.Utils is
  procedure Put (C : in Number);
  function As_String (C : Number) return String;
  ...
end Complex.Utils;
```

## Extension Can See Private Section

- With certain limitations

```
with Ada.Text_IO;
package body Complex.Utils is
  procedure Put (C : in Number) is
  begin
    Ada.Text_IO.Put (As_String (C));
  end Put;
  function As_String (C : Number) return String is
  begin
    -- Real_Part and Imaginary_Part are
    -- visible to child's body
    return "(" & Float'Image (C.Real_Part) & ", " &
           Float'Image (C.Imaginary_Part) & ")";
  end As_String;
  ...
end Complex.Utils;
```

# Subsystem Approach

```
with Interfaces.C;
package OS is -- Unix and/or POSIX
  type File_Descriptor is new Interfaces.C.int;
  ...
end OS;

package OS.Mem_Mgmt is
  ...
  procedure Dump (File           : File_Descriptor;
                  Requested_Location : System.Address;
                  Requested_Size   : Interfaces.C.Size_T);
  ...
end OS.Mem_Mgmt;

package OS.Files is
  ...
  function Open (Device : Interfaces.C.char_array;
                Permission : Permissions := S_IRWXO)
    return File_Descriptor;
  ...
end OS.Files;
```

# Predefined Hierarchies

- Standard library facilities are children of **Ada**
  - **Ada.Text\_IO**
  - **Ada.Calendar**
  - **Ada.Command\_Line**
  - **Ada.Exceptions**
  - et cetera
- Other root packages are also predefined
  - **Interfaces.C**
  - **Interfaces.Fortran**
  - **System.Storage\_Pools**
  - **System.Storage\_Elements**
  - et cetera

# Hierarchical Visibility

- Children can see ancestors' visible and private parts
  - All the way up to the root library unit
- Siblings have no automatic visibility to each other
- Visibility same as nested
  - As if child library units are nested within parents
    - All child units come after the root parent's specification
    - Grandchildren within children, great-grandchildren within ...



## Example of Visibility As If Nested

```
package Complex is
  type Number is private;
  function "*" (Left, Right : Number) return Number;
  function "/" (Left, Right : Number) return Number;
  function "+" (Left, Right : Number) return Number;
  ...
private
  type Number is record
    Real_Part : Float;
    Imaginary : Float;
  end record;
  package Utils is
    procedure Put (C : in Number);
    function As_String (C : Number) return String;
    ...
  end Utils;
end Complex;
```

## with Clauses for Ancestors Are Implicit

- Because children can reference ancestors' private parts
  - Code is not in executable unless somewhere in the **with** clauses
- Explicit clauses for ancestors are redundant but OK

```
package Parent is
    ...
private
    A : Integer := 10;
end Parent;

-- no "with" of parent needed
package Parent.Child is
    ...
private
    B : Integer := Parent.A;
    -- no dot-notation needed
    C : Integer := A;
end Parent.Child;
```

## with Clauses for Siblings Are Required

- If references are intended

```
with A.Foo; --required  
package body A.Bar is  
  ...  
  -- 'Foo' is directly visible because of the  
  -- implied nesting rule  
  X : Foo.Typemark;  
end A.Bar;
```

# Quiz

```
package Parent is
  Parent_Object : Integer;
end Parent;
```

```
package Parent.Sibling is
  Sibling_Object : Integer;
end Parent.Sibling;
```

```
package Parent.Child is
  Child_Object : Integer := ? ;
end Parent.Child;
```

Which is (are) **NOT** legal initialization(s) of Child\_Object?

- A. Parent.Parent\_Object + Parent.Sibling.Sibling\_Object
- B. Parent\_Object + Sibling.Sibling\_Object
- C. Parent\_Object + Sibling\_Object
- D. None of the above

# Quiz

```
package Parent is
  Parent_Object : Integer;
end Parent;
```

```
package Parent.Sibling is
  Sibling_Object : Integer;
end Parent.Sibling;
```

```
package Parent.Child is
  Child_Object : Integer := ? ;
end Parent.Child;
```

Which is (are) **NOT** legal initialization(s) of Child\_Object?

- A. *Parent.Parent\_Object + Parent.Sibling.Sibling\_Object*
- B. *Parent\_Object + Sibling.Sibling\_Object*
- C. *Parent\_Object + Sibling\_Object*
- D. None of the above

A, B, and C are illegal because there is no reference to package Parent.Sibling (the reference to Parent is implied by the hierarchy). If Parent.Child had "**with** Parent.Sibling;" , then A and B would be legal, but C would still be incorrect because there is no implied reference to a sibling.

## Visibility Limits

# Parents Do Not Know Their Children!

- Children grant themselves access to ancestors' private parts
  - May be created well after parent
  - Parent doesn't know if/when child packages will exist
- Alternatively, language *could have* been designed to grant access when declared
  - Like `friend` units in C++
  - But would have to be prescient!
    - Or else adding children requires modifying parent
  - Hence too restrictive
- Note: Parent body can reference children
  - Typical method of parsing out complex processes

## Correlation to C++ Class Visibility Controls

- Ada private part is visible to child units

```
package P is
  A ...
private
  B ...
end P;
package body P is
  C ...
end P;
```

- Thus private part is like the protected part in C++

```
class C {
public:
  A ...
protected:
  B ...
private:
  C ...
};
```

# Visibility Limits

- Visibility to parent's private part is not open-ended
  - Only visible to private parts and bodies of children
  - As if only private part of child package is nested in parent
- Recall users can only reference exported declarations
  - Child public spec only has access to parent public spec

```
package Parent is
```

```
  ...
```

```
private
```

```
  type Parent_T is ...
```

```
end Parent;
```

```
package Parent.Child is
```

```
  -- Parent_T is not visible here!
```

```
private
```

```
  -- Parent_T is visible here
```

```
end Parent.Child;
```

```
package body Parent.Child is
```

```
  -- Parent_T is visible here
```

```
end Parent.Child;
```

## Children Can Break Abstraction

- Could **break** a parent's abstraction
  - Alter a parent package state
  - Alters an ADT object state
- Useful for reset, testing: fault injections...

```
package Stack is
```

```
  ...
```

```
private
```

```
  Values : array (1 .. N) of Foo;
```

```
  Top : Natural range 0 .. N := 0;
```

```
end Stack;
```

```
package body Stack.Reset is
```

```
  procedure Reset is
```

```
  begin
```

```
    Top := 0;
```

```
  end Reset;
```

```
end Stack.Reset;
```

## Using Children for Debug

- Provide **accessors** to parent's private information
- eg internal metrics...

```
package P is
    ...
private
    Internal_Counter : Integer := 0;
end P;

package P.Child is
    function Count return Integer;
end P.Child;

package body P.Child is
    function Count return Integer is
    begin
        return Internal_Counter;
    end Count;
end P.Child;
```

# Quiz

```
package P is
  Object_A : Integer;
private
  Object_B : Integer;
  procedure Dummy_For_Body;
end P;
```

```
package body P is
  Object_C : Integer;
  procedure Dummy_For_Body is null;
end P;
```

```
package P.Child is
  function X return Integer;
end P.Child;
```

Which return statement would be legal in P.Child.X?

- A. return Object\_A;
- B. return Object\_B;
- C. return Object\_C;
- D. None of the above

# Quiz

```
package P is
  Object_A : Integer;
private
  Object_B : Integer;
  procedure Dummy_For_Body;
end P;

package body P is
  Object_C : Integer;
  procedure Dummy_For_Body is null;
end P;

package P.Child is
  function X return Integer;
end P.Child;
```

Which return statement would be legal in P.Child.X?

- A. `return Object_A;`
- B. `return Object_B;`
- C. `return Object_C;`
- D. None of the above

Explanations

- A. Object\_A is in the public part of P - visible to any unit that `with's` P
- B. Object\_B is in the private part of P - visible in the private part or body of any descendant of P
- C. Object\_C is in the body of P, so it is only visible in the body of P
- D. A and B are both valid completions

## Private Children

# Private Children

- Intended as implementation artifacts
- Only available within subsystem
  - Rules prevent **with** clauses by clients
  - Thus cannot export anything outside subsystem
  - Thus have no parent visibility restrictions
    - Public part of child also has visibility to ancestors' private parts

```
private package Maze.Debug is
  procedure Dump_State;
  . . .
end Maze.Debug;
```

## Rules Preventing Private Child Visibility

- Only available within immediate family
  - Rest of subsystem cannot import them
- Public unit declarations have import restrictions
  - To prevent re-exporting private information
- Public unit bodies have no import restrictions
  - Since can't re-export any imported info
- Private units can import anything
  - Declarations and bodies can import public and private units
  - Cannot be imported outside subsystem so no restrictions

# Import Rules

- Only parent of private unit and its descendants can import a private child
- Public unit declarations import restrictions
  - Not allowed to have **with** clauses for private units
    - Exception explained in a moment
  - Precludes re-exporting private information
- Private units can import anything
  - Declarations and bodies can import private children

## Some Public Children Are Trustworthy

- Would only use a private sibling's exports privately
- But rules disallow `with` clause

```
private package OS.UART is
  type Device is limited private;
  procedure Open (This : out Device; ...);
  ...
end OS.UART;
```

```
-- illegal - private child
with OS.UART;
package OS.Serial is
  type COM_Port is limited private;
  ...
private
  type COM_Port is limited record
    -- but I only need it here!
    COM : OS.UART.Device;
    ...
  end record;
end OS.Serial;
```

## Solution 1: Move Type to Parent Package

```
package OS is
  ...
private
  -- no longer an ADT!
  type Device is limited private;
  ...
end OS;
private package OS.UART is
  procedure Open (This : out Device;
    ...);
  ...
end OS.UART;

package OS.Serial is
  type COM_Port is limited private;
  ...
private
  type COM_Port is limited record
    COM : Device; -- now visible
    ...
  end record;
end OS.Serial;
```

## Solution 2: Partially Import Private Unit

- Via `private with` clause

- Syntax

```
private with package_name {, package_name} ;
```

- Public declarations can then access private siblings
  - But only in their private part
  - Still prevents exporting contents of private unit
- The specified package need not be a private unit
  - But why bother otherwise

## private with Example

```
private package OS.UART is
  type Device is limited private;
  procedure Open (This : out Device;
    ...);
  ...
end OS.UART;

private with OS.UART;
package OS.Serial is
  type COM_Port is limited private;
  ...
private
  type COM_Port is limited record
    COM : OS.UART.Device;
    ...
  end record;
end OS.Serial;
```

## Combining Private and Limited Withs

- Cyclic **limited with** clauses allowed
- A public unit can **with** a private unit
- With-ed unit only visible in the private part

```
limited with Parent.Public_Child;  
private package Parent.Private_Child is  
    type T is ...  
end Parent.Private_Child;
```

```
limited private with Parent.Private_Child;  
package Parent.Public_Child is  
    ...  
private  
    X : access Parent.Private_Child.T;  
end Parent.Public_Child;
```

## Child Subprograms

- Child units can be subprograms
  - Recall syntax
  - Both public and private child subprograms
- Separate declaration required if private
  - Syntax doesn't allow **private** on subprogram bodies
- Only library packages can be parents
  - Only they have necessary scoping

```
private procedure Parent.Child;
```

Lab

# Program Structure Lab

- Requirements
  - Create a message data type
    - Actual message type should be private
    - Need primitives to construct message and query contents
  - Create a child package that allows clients to modify the contents of the message
  - Main program should
    - Build a message
    - Print the contents of the message
    - Modify part of the message
    - Print the new contents of the message
- **Note: There is no prompt for this lab - you need to learn how to build the program structure**

# Program Structure Lab Solution - Messages

```
1 package Messages is
2   type Message_T is private;
3   type Kind_T is (Command, Query);
4   type Request_T is digits 6;
5   type Status_T is mod 255;
6
7   function Create (Kind    : Kind_T;
8                  Request  : Request_T;
9                  Status   : Status_T)
10                  return Message_T;
11
12  function Kind (Message : Message_T) return Kind_T;
13  function Request (Message : Message_T) return Request_T;
14  function Status (Message : Message_T) return Status_T;
15
16 private
17  type Message_T is record
18    Kind    : Kind_T;
19    Request : Request_T;
20    Status  : Status_T;
21  end record;
22 end Messages;
23
24 package body Messages is
25
26  function Create (Kind    : Kind_T;
27                Request  : Request_T;
28                Status   : Status_T)
29                return Message_T is
30    (Kind => Kind, Request => Request, Status => Status);
31
32  function Kind (Message : Message_T) return Kind_T is
33    (Message.Kind);
34  function Request (Message : Message_T) return Request_T is
35    (Message.Request);
36  function Status (Message : Message_T) return Status_T is
37    (Message.Status);
38
39 end Messages;
```

# Program Structure Lab Solution - Message Modification

```
1 package Messages.Modify is
2
3     procedure Kind (Message : in out Message_T;
4                     New_Value : Kind_T);
5     procedure Request (Message : in out Message_T;
6                       New_Value : Request_T);
7     procedure Status (Message : in out Message_T;
8                      New_Value : Status_T);
9
10 end Messages.Modify;
11
12 package body Messages.Modify is
13
14     procedure Kind (Message : in out Message_T;
15                   New_Value : Kind_T) is
16     begin
17         Message.Kind := New_Value;
18     end Kind;
19
20     procedure Request (Message : in out Message_T;
21                      New_Value : Request_T) is
22     begin
23         Message.Request := New_Value;
24     end Request;
25
26     procedure Status (Message : in out Message_T;
27                     New_Value : Status_T) is
28     begin
29         Message.Status := New_Value;
30     end Status;
31
32 end Messages.Modify;
```

# Program Structure Lab Solution - Main

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Messages;
3  with Messages.Modify;
4  procedure Main is
5      Message : Messages.Message_T;
6      procedure Print is
7          begin
8              Put_Line ("Kind => " & Messages.Kind (Message)'Image);
9              Put_Line ("Request => " & Messages.Request (Message)'Image);
10             Put_Line ("Status => " & Messages.Status (Message)'Image);
11             New_Line;
12         end Print;
13     begin
14         Message := Messages.Create (Kind    => Messages.Command,
15                                     Request => 12.34,
16                                     Status  => 56);
17         Print;
18         Messages.Modify.Request (Message    => Message,
19                                   New_Value => 98.76);
20         Print;
21     end Main;
```

## Summary

# Summary

- Hierarchical library units address important issues
  - Direct support for subsystems
  - Extension without recompilation
  - Separation of concerns with controlled sharing of visibility
- Parents should document assumptions for children
  - "These must always be in ascending order!"
- Children cannot misbehave unless imported ("with'ed")
- The writer of a child unit must be trusted
  - As much as if he or she were to modify the parent itself

# Visibility

## Introduction

## Improving Readability

- Descriptive names plus hierarchical packages makes for very long statements

```
Messages.Queue.Diagnostics.Inject_Fault (  
    Fault      => Messages.Queue.Diagnostics.CRC_Failure,  
    Position => Messages.Queue.Front);
```

- Operators treated as functions defeat the purpose of overloading  

```
Complex1 := Complex_Types."+" (Complex2, Complex3);
```
- Ada has mechanisms to simplify hierarchies

# Operators and Primitives

## ■ *Operators*

- Constructs which behave generally like functions but which differ syntactically or semantically
- Typically arithmetic, comparison, and logical

## ■ **Primitive operation**

- Predefined operations such as = and + etc.
- Subprograms declared in the same package as the type and which operate on the type
- Inherited or overridden subprograms
- For **tagged** types, class-wide subprograms
- Enumeration literals

## "use" Clauses

# "use" Clauses

- **use** Pkg; provides direct visibility into public items in Pkg
  - *Direct Visibility* - as if object was referenced from within package being used
  - *Public Items* - any entity defined in package spec public section
- May still use expanded name

```
package Ada.Text_IO is
  procedure Put_Line (...);
  procedure New_Line (...);
  ...
end Ada.Text_IO;

with Ada.Text_IO;
procedure Hello is
  use Ada.Text_IO;
begin
  Put_Line ("Hello World");
  New_Line (3);
  Ada.Text_IO.Put_Line ("Good bye");
end Hello;
```

## "use" Clause Syntax

- May have several, like **with** clauses
- Can refer to any visible package (including nested packages)
- Syntax

```
use_package_clause ::= use package_name {, package_name}
```

- Can only **use** a package
  - Subprograms have no contents to **use**

# "use" Clause Scope

- Applies to end of body, from first occurrence

```
package Pkg_A is
  Constant_A : constant := 123;
end Pkg_A;

package Pkg_B is
  Constant_B : constant := 987;
end Pkg_B;

with Pkg_A;
with Pkg_B;
use Pkg_A; -- everything in Pkg_A is now visible
package P is
  A : Integer := Constant_A; -- legal
  B1 : Integer := Constant_B; -- illegal
  use Pkg_B; -- everything in Pkg_B is now visible
  B2 : Integer := Constant_B; -- legal
  function F return Integer;
end P;

package body P is
  -- all of Pkg_A and Pkg_B is visible here
  function F return Integer is (Constant_A + Constant_B);
end P;
```

# No Meaning Changes

- A new **use** clause won't change a program's meaning!
- Any directly visible names still refer to the original entities

```
package D is
  T : Float;
end D;

with D;
procedure P is
  procedure Q is
    T, X : Float;
  begin
    ...
    declare
      use D;
    begin
      -- With or without the clause, "T" means Q.T
      X := T;
    end;
    ...
  end Q;
end P;
```

# No Ambiguity Introduction

```
package D is
  V : Boolean;
end D;
```

```
package E is
  V : Integer;
end E;
with D, E;
```

```
procedure P is
  procedure Q is
    use D, E;
  begin
    -- to use V here, must specify D.V or E.V
    ...
  end Q;
begin
  ...
end;
```

## "use" Clauses and Child Units

- A clause for a child does **not** imply one for its parent
- A clause for a parent makes the child **directly** visible
  - Since children are 'inside' declarative region of parent

```
package Parent is
```

```
  P1 : Integer;
```

```
end Parent;
```

```
package Parent.Child is
```

```
  PC1 : Integer;
```

```
end Parent.Child;
```

```
with Parent;
```

```
with Parent.Child; use Parent.Child;
```

```
procedure Demo is
```

```
  D1 : Integer := Parent.P1;
```

```
  D2 : Integer := Parent.Child.PC1;
```

```
  use Parent;
```

```
  D3 : Integer := P1; -- illegal
```

```
  D4 : Integer := PC1;
```

```
  ...
```

## "use" Clause and Implicit Declarations

- Visibility rules apply to implicit declarations too

```
package P is
  type Int is range Lower .. Upper;
  -- implicit declarations
  -- function "+"(Left, Right : Int) return Int;
  -- function "="(Left, Right : Int) return Boolean;
end P;

with P;
procedure Test is
  A, B, C : P.Int := some_value;
begin
  C := A + B; -- illegal reference to operator
  C := P."+" (A,B);
  declare
    use P;
  begin
    C := A + B; -- now legal
  end;
end Test;
```

## "use type" and "use all type" Clauses

## "use type" and "use all type"

- **use type** makes **primitive operators** directly visible for specified type
  - Implicit and explicit operator function declarations

```
use type subtype_mark {, subtype_mark};
```
- **use all type** makes primitive operators **and all other operations** directly visible for specified type
  - All **enumerated type values** will also be directly visible

```
use all type subtype_mark {, subtype_mark};
```
- More specific alternatives to **use** clauses
  - Especially useful when multiple **use** clauses introduce ambiguity

## Example Code

```
package Types is
  type Distance_T is range 0 .. Integer'Last;

  -- explicit declaration
  -- (we don't want a negative distance)
  function "-" (Left, Right : Distance_T)
    return Distance_T;

  -- implicit declarations (we get the division operator
  -- for "free", showing it for completeness)
  -- function "/" (Left, Right : Distance_T) return
  --           Distance_T;

  -- primitive operation
  function Min (A, B : Distance_T)
    return Distance_T;

end Types;
```

# "use" Clauses Comparison

Blue = context clause being used

Red = compile errors with the context clause

## No "use" clause

```
with Get_Distance;
with Types;
package Example is
  -- no context clause

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);
end Example;
```

## "use type" clause

```
with Get_Distance;
with Types;
package Example is
  use type Types.Distance;

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);
end Example;
```

## "use" clause

```
with Get_Distance;
with Types;
package Example is
  use Types;

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);
end Example;
```

## "use all type" clause

```
with Get_Distance;
with Types;
package Example is
  use all type Types.Distance;

  Point0 : Distance_T := Get_Distance;
  Point1 : Types.Distance_T := Get_Distance;
  Point2 : Types.Distance_T := Get_Distance;
  Point3 : Types.Distance_T := (Point1 - Point2) / 2;
  Point4 : Types.Distance_T := Min (Point1, Point2);
end Example;
```

## Multiple "use type" Clauses

- May be necessary
- Only those that mention the type in their profile are made visible

```
package P is
  type T1 is range 1 .. 10;
  type T2 is range 1 .. 10;
  -- implicit
  -- function "+"(Left : T2; Right : T2) return T2;
  type T3 is range 1 .. 10;
  -- explicit
  function "+"(Left : T1; Right : T2) return T3;
end P;

with P;
procedure UseType is
  X1 : P.T1;
  X2 : P.T2;
  X3 : P.T3;
  use type P.T1;
begin
  X3 := X1 + X2; -- operator visible because it uses T1
  X2 := X2 + X2; -- operator not visible
end UseType;
```

## Renaming Entities

# Three Positives Make a Negative

- Good Coding Practices ...
  - Descriptive names
  - Modularization
  - Subsystem hierarchies
- Can result in cumbersome references

```
-- use cosine rule to determine distance between two points,  
-- given angle and distances between observer and 2 points  
--  $A^2 = B^2 + C^2 - 2*B*C*\cos(\text{angle})$ 
```

```
Observation.Sides (Viewpoint_Types.Point1_Point2) :=  
  Math_Utilities.Square_Root  
    (Observation.Sides (Viewpoint_Types.Observer_Point1)**2 +  
     Observation.Sides (Viewpoint_Types.Observer_Point2)**2 -  
     2.0 * Observation.Sides (Viewpoint_Types.Observer_Point1) *  
     Observation.Sides (Viewpoint_Types.Observer_Point2) *  
     Math_Utilities.Trigonometry.Cosine  
     (Observation.Vertices (Viewpoint_Types.Observer)));
```

# Writing Readable Code - Part 1

- We could use **use** on package names to remove some dot-notation

```
-- use cosine rule to determine distance between two points, given angle
-- and distances between observer and 2 points A**2 = B**2 + C**2 -
-- 2*B*C*cos(angle)
```

```
Observation.Sides (Point1_Point2) :=
  Square_Root
  (Observation.Sides (Observer_Point1)**2 +
   Observation.Sides (Observer_Point2)**2 -
   2.0 * Observation.Sides (Observer_Point1) *
    Observation.Sides (Observer_Point2) *
    Cosine (Observation.Vertices (Observer)));
```

- But that only shortens the problem, not simplifies it
  - If there are multiple "use" clauses in scope:
    - Reviewer may have hard time finding the correct definition
    - Homographs may cause ambiguous reference errors
- We want the ability to refer to certain entities by another name (like an alias) with full read/write access (unlike temporary variables)

# The "renames" Keyword

- **renames** declaration creates an alias to an entity

- Packages

```
package Trig renames Math.Trigonometry
```

- Objects (or components of objects)

```
Angles : Viewpoint_Types.Vertices_Array_T  
       renames Observation.Vertices;
```

```
Required_Angle : Viewpoint_Types.Vertices_T  
               renames Viewpoint_Types.Observer;
```

- Subprograms

```
function Sqrt (X : Base_Types.Float_T)  
            return Base_Types.Float_T  
            renames Math.Square_Root;
```

## Writing Readable Code - Part 2

- With **renames** our complicated code example is easier to understand
  - Executable code is very close to the specification
  - Declarations as "glue" to the implementation details

**begin**

```
package Math renames Math_Uutilities;  
package Trig renames Math.Trigonometry;
```

```
function Sqrt (X : Base_Types.Float_T) return Base_Types.Float_T  
    renames Math.Square_Root;  
function Cos ...
```

```
B : Base_Types.Float_T  
    renames Observation.Sides (Viewpoint_Types.Observer_Point1);  
-- Rename the others as Side2, Angles, Required_Angle, Desired_Side
```

**begin**

```
...  
-- A**2 = B**2 + C**2 - 2*B*C*cos(angle)  
A := Sqrt (B**2 + C**2 - 2.0 * B * C * Cos (Angle));
```

**end;**

Lab

# Visibility Lab

## ■ Requirements

- Create two types packages for two different shapes. Each package should have the following components:
  - `Number_of_Sides` - indicates how many sides in the shape
  - `Side_T` - numeric value for length
  - `Shape_T` - array of `Side_T` components whose length is `Number_of_Sides`
- Create a main program that will
  - Create an object of each `Shape_T`
  - Set the values for each component in `Shape_T`
  - Add all the components in each object and print the total

## ■ Hints

- There are multiple ways to resolve this!

# Visibility Lab Solution - Types

```
1 package Quads is
2
3     Number_Of_Sides : constant Natural := 4;
4     type Side_T is range 0 .. 1_000;
5     type Shape_T is array (1 .. Number_Of_Sides) of Side_T;
6
7 end Quads;
8
9 package Triangles is
10
11     Number_Of_Sides : constant Natural := 3;
12     type Side_T is range 0 .. 1_000;
13     type Shape_T is array (1 .. Number_Of_Sides) of Side_T;
14
15 end Triangles;
```

# Visibility Lab Solution - Main #1

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Quads;
3 with Triangles;
4 procedure Main1 is
5
6     use type Quads.Side_T;
7     Q_Sides : Natural renames Quads.Number_Of_Sides;
8     Quad    : Quads.Shape_T := (1, 2, 3, 4);
9     Quad_Total : Quads.Side_T := 0;
10
11     use type Triangles.Side_T;
12     T_Sides : Natural renames Triangles.Number_Of_Sides;
13     Triangle : Triangles.Shape_T := (1, 2, 3);
14     Triangle_Total : Triangles.Side_T := 0;
15
16 begin
17
18     for I in 1 .. Q_Sides loop
19         Quad_Total := Quad_Total + Quad (I);
20     end loop;
21     Put_Line ("Quad: " & Quads.Side_T'Image (Quad_Total));
22
23     for I in 1 .. T_Sides loop
24         Triangle_Total := Triangle_Total + Triangle (I);
25     end loop;
26     Put_Line ("Triangle: " & Triangles.Side_T'Image (Triangle_Total));
27
28 end Main1;
```

# Visibility Lab Solution - Main #2

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Quads;      use Quads;
3 with Triangles;  use Triangles;
4 procedure Main2 is
5     function Q_Image (S : Quads.Side_T) return String
6         renames Quads.Side_T'Image;
7     Quad : Quads.Shape_T := (1, 2, 3, 4);
8     Quad_Total : Quads.Side_T := 0;
9
10    function T_Image (S : Triangles.Side_T) return String
11        renames Triangles.Side_T'Image;
12    Triangle : Triangles.Shape_T := (1, 2, 3);
13    Triangle_Total : Triangles.Side_T := 0;
14
15 begin
16
17     for I in Quad'Range loop
18         Quad_Total := Quad_Total + Quad (I);
19     end loop;
20     Put_Line ("Quad: " & Q_Image (Quad_Total));
21
22     for I in Triangle'Range loop
23         Triangle_Total := Triangle_Total + Triangle (I);
24     end loop;
25     Put_Line ("Triangle: " & T_Image (Triangle_Total));
26
27 end Main2;
```

## Summary

# Summary

- **use** clauses are not evil but can be abused
  - Can make it difficult for others to understand code
- **use all type** clauses are more likely in practice than **use type** clauses
- **Renames** allow us to alias entities to make code easier to read
  - Subprogram renaming has many other uses, such as adding / removing default parameter values

# Tagged Derivation

# Introduction

# Object-Oriented Programming with Tagged Types

- For **record** types

```
type T is tagged record
```

```
...
```

- Child types can add new components (*attributes*)
- Object of a child type can be **substituted** for base type
- Primitive (*method*) can **dispatch** **at run-time** depending on the type at call-site
- Types can be **extended** by other packages
  - Conversion and qualification to base type is allowed
- Private data is encapsulated through **privacy**

## Tagged Derivation Ada Vs C++

```
type T1 is tagged record
  Member1 : Integer;
end record;

procedure Attr_F (This : T1);

type T2 is new T1 with record
  Member2 : Integer;
end record;

overriding procedure Attr_F (
  This : T2);
procedure Attr_F2 (This : T2);

class T1 {
  public:
    int Member1;
    virtual void Attr_F(void);
};

class T2 : public T1 {
  public:
    int Member2;
    virtual void Attr_F(void);
    virtual void Attr_F2(void);
};
```

## Tagged Derivation

## Difference with Simple Derivation

- Tagged derivation **can** change the structure of a type
  - Keywords **tagged record** and **with record**

```
type Root is tagged record
```

```
  F1 : Integer;
```

```
end record;
```

```
type Child is new Root with record
```

```
  F2 : Integer;
```

```
end record;
```

- Conversion is only allowed from **child to parent**

```
V1 : Root;
```

```
V2 : Child;
```

```
...
```

```
V1 := Root (V2);
```

```
V2 := Child (V1); -- illegal
```

# Primitives

- Child **cannot remove** a primitive
- Child **can add** new primitives
- *Controlling parameter*
  - Parameters the subprogram is a primitive of
  - For **tagged** types, all should have the **same type**

```
type Root1 is tagged null record;  
type Root2 is tagged null record;
```

```
procedure P1 (V1 : Root1;  
             V2 : Root1);  
procedure P2 (V1 : Root1;  
             V2 : Root2); -- illegal
```

# Freeze Point for Tagged Types

- Freeze point definition does not change
  - A variable of the type is declared
  - The type is derived
  - The end of the scope is reached
- Declaring tagged type primitives past freeze point is **forbidden**

```
type Root is tagged null record;
```

```
procedure Prim (V : Root);
```

```
type Child is new Root with null record; -- freeze root
```

```
procedure Prim2 (V : Root); -- illegal
```

```
V : Child; -- freeze child
```

```
procedure Prim3 (V : Child); -- illegal
```

# Overriding Indicators

- Optional **overriding** and **not overriding** indicators

```
type Shape_T is tagged record
  Name : String (1..10);
end record;

-- primitives of "Shape_T"
function Get_Name (S : Shape_T) return String;
procedure Set_Name (S : in out Shape_T);

-- Derive "Point" from Shape_T
type Point_T is new Shape_T with record
  Origin : Coord_T;
end Point_T;

-- Get_Name is inherited
-- We want to _change_ the behavior of Set_Name
overriding procedure Set_Name (P : in out Point_T);
-- We want to _add_ a new primitive
not overriding procedure Set-Origin (P : in out Point_T);
```

# Prefix Notation

- Tagged types primitives can be called as usual
- The call can use prefixed notation
  - If the first argument is a controlling parameter
  - No need for **use** or **use type** for visibility

```
-- Prim1 visible even without *use Pkg*  
X.Prim1;
```

```
declare  
    use Pkg;  
begin  
    Prim1 (X);  
end;
```

# Quiz

Which declaration(s) will make P a primitive of T1?

- A** type T1 is tagged null record;  
procedure P (O : T1) is null;
- B** type T0 is tagged null record;  
type T1 is new T0 with null record;  
type T2 is new T0 with null record;  
procedure P (O : T1) is null;
- C** type T1 is tagged null record;  
Object : T1;  
procedure P (O : T1) is null;
- D** package Nested is  
type T1 is tagged null record;  
end Nested;  
use Nested;  
procedure P (O : T1) is null;

# Quiz

Which declaration(s) will make P a primitive of T1?

- A.** `type T1 is tagged null record;`  
`procedure P (O : T1) is null;`
  - B.** `type T0 is tagged null record;`  
`type T1 is new T0 with null record;`  
`type T2 is new T0 with null record;`  
`procedure P (O : T1) is null;`
  - C.** `type T1 is tagged null record;`  
`Object : T1;`  
`procedure P (O : T1) is null;`
  - D.** `package Nested is`  
`type T1 is tagged null record;`  
`end Nested;`  
`use Nested;`  
`procedure P (O : T1) is null;`
- A.** Primitive (same scope)
  - B.** Primitive (T1 is not yet frozen)
  - C.** T1 is frozen by the object declaration
  - D.** Primitive must be declared in same scope as type

# Quiz

```
with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;
```

```
procedure Main is
  The_Shape : Shapes.Shape;
  The_Color : Colors.Color;
  The_Weight : Weights.Weight;
```

Which statement(s) is (are) valid?

- A. The\_Shape.P
- B. P (The\_Shape)
- C. P (The\_Color)
- D. P (The\_Weight)

# Quiz

```
with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;
```

```
procedure Main is
  The_Shape : Shapes.Shape;
  The_Color : Colors.Color;
  The_Weight : Weights.Weight;
```

Which statement(s) is (are) valid?

- A. *The\_Shape.P*
  - B. *P (The\_Shape)*
  - C. *P (The\_Color)*
  - D. *P (The\_Weight)*
- D. **use type** only gives visibility to operators; needs to be **use all type**

# Quiz

Which code block(s) is (are) legal?

**A** type A1 is record

```
    Component1 : Integer;  
end record;  
type A2 is new A1 with  
null record;
```

**B** type B1 is tagged

```
record  
    Component2 : Integer;  
end record;  
type B2 is new B1 with  
record  
    Component2b :  
Integer;  
end record;
```

**C** type C1 is tagged

```
record  
    Component3 : Integer;  
end record;  
type C2 is new C1 with  
record  
    Component3 : Integer;  
end record;
```

**D** type D1 is tagged

```
record  
    Component1 : Integer;  
end record;  
type D2 is new D1;
```

# Quiz

Which code block(s) is (are) legal?

- A** type A1 is record  
    Component1 : Integer;  
end record;  
type A2 is new A1 with  
null record;
- B** *type B1 is tagged  
record  
    Component2 : Integer;  
end record;  
type B2 is new B1 with  
record  
    Component2b :  
Integer;  
end record;*
- C** type C1 is tagged  
record  
    Component3 : Integer;  
end record;  
type C2 is new C1 with  
record  
    Component3 : Integer;  
end record;
- D** type D1 is tagged  
record  
    Component1 : Integer;  
end record;  
type D2 is new D1;

Explanations

- A**. Cannot extend a non-tagged type  
**B**. Correct  
**C**. Components must have distinct names  
**D**. Types derived from a tagged type must have an extension

## Extending Tagged Types

# How Do You Extend a Tagged Type?

- Premise of a tagged type is to `extend` an existing type
- In general, that means we want to add more components
  - We can extend a `tagged` type by adding components

```
package Animals is
  type Animal_T is tagged record
    Age : Natural;
  end record;
end Animals;

with Animals; use Animals;
package Mammals is
  type Mammal_T is new Animal_T with record
    Number_Of_Legs : Natural;
  end record;
end Mammals;

with Mammals; use Mammals;
package Canines is
  type Canine_T is new Mammal_T with record
    Domesticated : Boolean;
  end record;
end Canines;
```

# Tagged Aggregate

- At initialization, all components (including **inherited**) must have a **value**

```
Animal : Animal_T := (Age => 1);  
Mammal : Mammal_T := (Age           => 2,  
                      Number_Of_Legs => 2);  
Canine  : Canine_T := (Age           => 2,  
                      Number_Of_Legs => 4,  
                      Domesticated   => True);
```

- But we can also "seed" the aggregate with a parent object

```
Mammal := (Animal with Number_Of_Legs => 4);  
Canine := (Animal with Number_Of_Legs => 4,  
          Domesticated   => False);  
Canine := (Mammal with Domesticated => True);
```

# Private Tagged Types

- But data hiding says types should be private!
- So we can define our base type as private

```
package Animals is
  type Animal_T is tagged private;
  function Get_Age (P : Animal_T) return Natural;
  procedure Set_Age (P : in out Animal_T; A : Natural);
private
  type Animal_T is tagged record
    Age : Natural;
  end record;
end Animals;
```

- And still allow derivation

```
with Animals;
package Mammals is
  type Mammal_T is new Animals.Animal_T with record
    Number_Of_Legs : Natural;
  end record;
```

- But now the only way to get access to Age is with accessor subprograms

## Private Extensions

- In the previous slide, we exposed the components for Mammal\_T!
- Better would be to make the extension itself private

```
package Mammals is
  type Mammal_T is new Animals.Animal_T with private;
private
  type Mammal_T is new Animals.Animal_T with record
    Number_Of_Legs : Natural;
  end record;
end Mammals;
```

## Aggregates with Private Tagged Types

- Remember, an aggregate must specify values for all components
  - But with private types, we can't see all the components!
- So we need to use the "seed" method:

```
procedure Inside_Mammals_Pkg is
  Animal : Animal_T := Animals.Create;
  Mammal  : Mammal_T;
begin
  Mammal := (Animal with Number_Of_Legs => 4);
  Mammal := (Animals.Create with Number_Of_Legs => 4);
end Inside_Mammals_Pkg;
```

- Note that we cannot use `others => <>` for components that are not visible to us

```
Mammal := (Number_Of_Legs => 4,
           others           => <>);  -- Compile Error
```

## Null Extensions

- To create a new type with no additional components
  - We still need to "extend" the record - we just do it with an empty record

```
type Dog_T is new Canine_T with null record;
```

- We still need to specify the "added" components in an aggregate

```
C      : Canine_T := Canines.Create;  
Dog1   : Dog_T := C; -- Compile Error  
Dog2   : Dog_T := (C with null record);
```

# Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
    Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
    Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

Which completion(s) of Create is (are) valid?

- `function Create return Child_T is (Parents.Create with Count => 0);`
- `function Create return Child_T is (others => <>);`
- `function Create return Child_T is (0, 0);`
- `function Create return Child_T is (P with Count => 0);`

# Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
    Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
    Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

Which completion(s) of Create is (are) valid?

- `function Create return Child_T is (Parents.Create with Count => 0);`
- `function Create return Child_T is (others => <>);`
- `function Create return Child_T is (0, 0);`
- `function Create return Child_T is (P with Count => 0);`

Explanations

- Correct - Parents.Create returns Parent\_T
- Cannot use `others` to complete private part of an aggregate
- Aggregate has no visibility to Id component, so cannot assign
- Correct - P is a Parent\_T

Lab

# Tagged Derivation Lab

## ■ Requirements

- Create a type structure that could be used in a business
  - A **person** has some defining characteristics
  - An **employee** is a *person* with some employment information
  - A **staff member** is an *employee* with specific job information
- Create primitive operations to read and print the objects
- Create a main program to test the objects and operations

## ■ Hints

- Use **overriding** and **not overriding** as appropriate (**Ada 2005 and above**)
- Data hiding is important!

# Tagged Derivation Lab Solution - Types (Spec)

```
1 package Employee is
2   type Person_T is tagged private;
3   subtype Name_T is String (1 .. 6);
4   type Date_T is record
5     Year   : Positive;
6     Month  : Positive;
7     Day    : Positive;
8   end record;
9   type Job_T is (Sales, Engineer, Bookkeeping);
10
11  procedure Set_Name (O      : in out Person_T;
12                    Value  : Name_T);
13  function Name (O : Person_T) return Name_T;
14  procedure Set_Birth_Date (O : in out Person_T;
15                           Value  : Date_T);
16  function Birth_Date (O : Person_T) return Date_T;
17  procedure Print (O : Person_T);
18
19  type Employee_T is new Person_T with private;
20  not overriding procedure Set_Start_Date (O : in out Employee_T;
21                                         Value  : Date_T);
22  not overriding function Start_Date (O : Employee_T) return Date_T;
23  overriding procedure Print (O : Employee_T);
24
25  type Position_T is new Employee_T with private;
26  not overriding procedure Set_Job (O : in out Position_T;
27                                   Value  : Job_T);
28  not overriding function Job (O : Position_T) return Job_T;
29  overriding procedure Print (O : Position_T);
30
31 private
32  type Person_T is tagged record
33    The_Name      : Name_T;
34    The_Birth_Date : Date_T;
35  end record;
36
37  type Employee_T is new Person_T with record
38    The_Employee_Id : Positive;
39    The_Start_Date  : Date_T;
40  end record;
41
42  type Position_T is new Employee_T with record
43    The_Job : Job_T;
44  end record;
45 end Employee;
```

## Tagged Derivation Lab Solution - Types (Partial Body)

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 package body Employee is
3
4     function Image (Date : Date_T) return String is
5         (Date.Year'Image & " -" & Date.Month'Image & " -" & Date.Day'Image);
6
7     procedure Set_Name (O : in out Person_T;
8                       Value : Name_T) is
9     begin
10        O.The_Name := Value;
11    end Set_Name;
12    function Name (O : Person_T) return Name_T is (O.The_Name);
13
14    procedure Set_Birth_Date (O : in out Person_T;
15                             Value : Date_T) is
16    begin
17        O.The_Birth_Date := Value;
18    end Set_Birth_Date;
19    function Birth_Date (O : Person_T) return Date_T is (O.The_Birth_Date);
20
21    procedure Print (O : Person_T) is
22    begin
23        Put_Line ("Name: " & O.Name);
24        Put_Line ("Birthdate: " & Image (O.Birth_Date));
25    end Print;
26
27    not overriding procedure Set_Start_Date (O : in out Employee_T;
28                                             Value : Date_T) is
29    begin
30        O.The_Start_Date := Value;
31    end Set_Start_Date;
32    not overriding function Start_Date (O : Employee_T) return Date_T is
33        (O.The_Start_Date);
34
35    overriding procedure Print (O : Employee_T) is
36    begin
37        Print (Person_T (O)); -- Use parent "Print"
38        Put_Line ("Startdate: " & Image (O.Start_Date));
39    end Print;
40
```

# Tagged Derivation Lab Solution - Main

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Employee;
3 procedure Main is
4     Applicant : Employee.Person_T;
5     Employ    : Employee.Employee_T;
6     Staff     : Employee.Position_T;
7
8 begin
9     Applicant.Set_Name ("Wilma ");
10    Applicant.Set_Birth_Date ((Year => 1_234,
11                               Month => 12,
12                               Day  => 1));
13
14    Employ.Set_Name ("Betty ");
15    Employ.Set_Birth_Date ((Year  => 2_345,
16                            Month => 11,
17                            Day   => 2));
18    Employ.Set_Start_Date ((Year => 3_456,
19                            Month => 10,
20                            Day   => 3));
21
22    Staff.Set_Name ("Bambam");
23    Staff.Set_Birth_Date ((Year => 4_567,
24                            Month => 9,
25                            Day   => 4));
26    Staff.Set_Start_Date ((Year => 5_678,
27                            Month => 8,
28                            Day   => 5));
29    Staff.Set_Job (Employee.Engineer);
30
31    Applicant.Print;
32    Employ.Print;
33    Staff.Print;
34 end Main;
```

## Summary

# Summary

- Tagged derivation
  - Building block for OOP types in Ada
- Primitives rules for tagged types are trickier
  - Primitives **forbidden** below freeze point
  - **Unique** controlling parameter
  - Tip: Keep the number of tagged type per package low

# Exceptions

# Introduction

# Rationale for Exceptions

- Textual separation from normal processing
- Rigorous Error Management
  - Cannot be ignored, unlike status codes from routines
  - Example: running out of gasoline in an automobile

```
package Automotive is
  type Vehicle is record
    Fuel_Quantity, Fuel_Minimum : Float;
    Oil_Temperature : Float;
    ...
  end record;
  Fuel_Exhausted : exception;
  procedure Consume_Fuel (Car : in out Vehicle);
  ...
end Automotive;
```

# Semantics Overview

- Exceptions become active by being *raised*
  - Failure of implicit language-defined checks
  - Explicitly by application
- Exceptions occur at run-time
  - A program has no effect until executed
- May be several occurrences active at same time
  - One per task
- Normal execution abandoned when they occur
  - Error processing takes over in response
  - Response specified by *exception handlers*
  - *Handling the exception* means taking action in response
  - Other tasks need not be affected

## Semantics Example: Raising

```
package body Automotive is
  function Current_Consumption return Float is
    ...
  end Current_Consumption;
  procedure Consume_Fuel (Car : in out Vehicle) is
  begin
    if Car.Fuel_Quantity <= Car.Fuel_Minimum then
      raise Fuel_Exhausted;
    else -- decrement quantity
      Car.Fuel_Quantity := Car.Fuel_Quantity -
                           Current_Consumption;
    end if;
  end Consume_Fuel;
  ...
end Automotive;
```

## Semantics Example: Handling

```
procedure Joy_Ride is
  Hot_Rod : Automotive.Vehicle;
  Bored : Boolean := False;
  use Automotive;
begin
  while not Bored loop
    Steer_Aimlessly (Bored);
    -- error situation cannot be ignored
    Consume_Fuel (Hot_Rod);
  end loop;
  Drive_Home;
exception
  when Fuel_Exhausted =>
    Push_Home;
end Joy_Ride;
```

## Handler Part Is Skipped Automatically

- If no exceptions are active, returns normally

```
begin
```

```
...
```

```
-- if we get here, skip to end
```

```
exception
```

```
  when Name1 =>
```

```
    ...
```

```
  when Name2 | Name3 =>
```

```
    ...
```

```
  when Name4 =>
```

```
    ...
```

```
end;
```

## Handlers

## Exception Handler Part

- Contains the exception handlers within a frame
  - Within block statements, subprograms, tasks, etc.
- Separates normal processing code from abnormal
- Starts with the reserved word **exception**
- Optional

```
begin  
    sequence_of_statements  
    [ exception  
        exception_handler  
        { exception handler } ]  
end
```

## Exception Handlers Syntax

- Associates exception names with statements to execute in response
- If used, **others** must appear at the end, by itself
  - Associates statements with all other exceptions
- Syntax

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

## Similarity to Case Statements

- Both structure and meaning
- Exception handler

```
...  
exception  
  when Constraint_Error | Storage_Error | Program_Error =>  
    ...  
  when others =>  
    ...  
end;
```

- Case statement

```
case exception_name is  
  when Constraint_Error | Storage_Error | Program_Error =>  
    ...  
  when others =>  
    ...  
end case;
```

# Handlers Don't "Fall Through"

```
begin
  ...
  raise Name3;
  -- code here is not executed
  ...
exception
  when Name1 =>
    -- not executed
    ...
  when Name2 | Name3 =>
    -- executed
    ...
  when Name4 =>
    -- not executed
    ...
end;
```

## When an Exception Is Raised

- Normal processing is abandoned
- Handler for active exception is executed, if any
- Control then goes to the caller
- If handled, caller continues normally, otherwise repeats the above

- Caller
  - ...
  - Joy\_Ride;
  - Do\_Something\_At\_Home;
  - ...
- Callee

```
procedure Joy_Ride is
    ...
begin
    ...
    Drive_Home;
exception
    when Fuel_Exhausted =>
        Push_Home;
end Joy_Ride;
```

## Handling Specific Statements' Exceptions

```
begin
  loop
    Prompting : loop
      Put (Prompt);
      Get_Line (Filename, Last);
      exit when Last > Filename'First - 1;
    end loop Prompting;
  begin
    Open (F, In_File, Filename (1..Last));
    exit;
  exception
    when Name_Error =>
      Put_Line ("File '" & Filename (1..Last) &
               "' was not found.");
  end;
end loop;
```

## Exception Handler Content

- No restrictions
  - Block statements, subprogram calls, etc.
- Do whatever makes sense

```
begin
    ...
exception
    when Some_Error =>
        declare
            New_Data : Some_Type;
        begin
            P (New_Data);
            ...
        end;
end;
```

## Quiz

```
1  procedure Main is
2      A, B, C, D : Integer range 0 .. 100;
3  begin
4      A := 1; B := 2; C := 3; D := 4;
5      begin
6          D := A - C + B;
7      exception
8          when others => Put_Line ("One");
9              D := 1;
10     end;
11     D := D + 1;
12     begin
13         D := D / (A - C + B);
14     exception
15         when others => Put_Line ("Two");
16             D := -1;
17     end;
18 exception
19     when others =>
20         Put_Line ("Three");
21 end Main;
```

What will get printed?

- A. One, Two, Three
- B. Two, Three
- C. Two
- D. Three

## Quiz

```
1  procedure Main is
2      A, B, C, D : Integer range 0 .. 100;
3  begin
4      A := 1; B := 2; C := 3; D := 4;
5      begin
6          D := A - C + B;
7      exception
8          when others => Put_Line ("One");
9              D := 1;
10     end;
11     D := D + 1;
12     begin
13         D := D / (A - C + B);
14     exception
15         when others => Put_Line ("Two");
16             D := -1;
17     end;
18 exception
19     when others =>
20         Put_Line ("Three");
21 end Main;
```

What will get printed?

- A. One, Two, Three
- B. *Two, Three*
- C. Two
- D. Three

Explanations

- A. Although  $(A - C)$  is not in the range of natural, the range is only checked on assignment, which is after the addition of B, so One is never printed
- B. Correct
- C. If we reach Two, the assignment on line 16 will cause Three to be reached
- D. Divide by 0 on line 13 causes an exception, so Two must be called

## Implicitly and Explicitly Raised Exceptions

## Implicitly-Raised Exceptions

- Correspond to language-defined checks
- Can happen by statement execution

```
K := -10;  -- where K must be greater than zero
```

- Can happen by declaration elaboration

```
Doomed : array (Positive) of Big_Type;
```

## Some Language-Defined Exceptions

- `Constraint_Error`
  - Violations of constraints on range, index, etc.
- `Program_Error`
  - Runtime control structure violated (function with no return ...)
- `Storage_Error`
  - Insufficient storage is available
- For a complete list see RM Q-4

## Explicitly-Raised Exceptions

- Raised by application via **raise** statements
  - Named exception becomes active

- Syntax

```
raise_statement ::= raise; |  
    raise exception_name  
    [with string_expression];
```

*Note "with string\_expression" only available in Ada 2005 and later*

- A **raise** by itself is only allowed in handlers

```
if Unknown (User_ID) then  
    raise Invalid_User;  
end if;
```

```
if Unknown (User_ID) then  
    raise Invalid_User  
    with "Attempt by " &  
        Image (User_ID);  
end if;
```

## User-Defined Exceptions

# User-Defined Exceptions

- Syntax

```
defining_identifier_list : exception;
```

- Behave like predefined exceptions

- Scope and visibility rules apply
- Referencing as usual
- Some minor differences

- Exception identifiers<sup>1</sup> use is restricted

- **raise** statements
- Handlers
- Renaming declarations

## User-Defined Exceptions Example

- An important part of the abstraction
- Designer specifies how component can be used

```
package Stack is
```

```
Underflow, Overflow : exception;
```

```
procedure Push (Item : in Integer);
```

```
...
```

```
end Stack;
```

```
package body Stack is
```

```
procedure Push (Item : in Integer) is
```

```
begin
```

```
  if Top = Index'Last then
```

```
    raise Overflow;
```

```
  end if;
```

```
  Top := Top + 1;
```

```
  Values (Top) := Item;
```

```
end Push;
```

```
...
```

# Propagation

# Propagation

- Control does not return to point of raising
  - Termination Model
- When a handler is not found in a block statement
  - Re-raised immediately after the block
- When a handler is not found in a subprogram
  - Propagated to caller at the point of call
- Propagation is dynamic, back up the call chain
  - Not based on textual layout or order of declarations
- Propagation stops at the main subprogram
  - Main completes abnormally unless handled

## Propagation Demo

```
1  procedure Do_Something is      16  begin -- Do_Something
2      Error : exception;        17      Maybe_Raise (3);
3      procedure Unhandled is    18      Handled;
4      begin                      19      exception
5          Maybe_Raise (1);      20          when Error =>
6      end Unhandled;           21          Print ("Handle 3");
7      procedure Handled is      22  end Do_Something;
8      begin
9          Unhandled;
10         Maybe_Raise (2);
11     exception
12         when Error =>
13             Print ("Handle 1 or 2");
14     end Handled;
```

# Termination Model

- When control goes to handler, it continues from here

```
procedure Joy_Ride is
begin
  loop
    Steer_Aimlessly;

    -- If next line raises Fuel_Exhausted, go to handler
    Consume_Fuel;
  end loop;
exception
  when Fuel_Exhausted => -- Handler
    Push_Home;
    -- Resume from here: loop has been exited
end Joy_Ride;
```

# Quiz

```
2 Main_Problem : exception;
3 I : Integer;
4 function F (P : Integer) return Integer is
5 begin
6   if P > 0 then
7     return P + 1;
8   elsif P = 0 then
9     raise Main_Problem;
10  end if;
11 end F;
12 begin
13   I := F(Input_Value);
14   Put_Line ("Success");
15 exception
16   when Constraint_Error => Put_Line ("Constraint Error");
17   when Program_Error   => Put_Line ("Program Error");
18   when others          => Put_Line ("Unknown problem");
```

What will get printed if Input\_Value on line 13 is Integer'Last?

- A Unknown Problem
- B Success
- C Constraint Error
- D Program Error

# Quiz

```
2 Main_Problem : exception;
3 I : Integer;
4 function F (P : Integer) return Integer is
5 begin
6   if P > 0 then
7     return P + 1;
8   elsif P = 0 then
9     raise Main_Problem;
10  end if;
11 end F;
12 begin
13   I := F(Input_Value);
14   Put_Line ("Success");
15 exception
16   when Constraint_Error => Put_Line ("Constraint Error");
17   when Program_Error   => Put_Line ("Program Error");
18   when others          => Put_Line ("Unknown problem");
```

What will get printed if Input\_Value on line 13 is Integer'Last?

- A Unknown Problem
- B Success
- C Constraint Error
- D Program Error

Explanations

- A "Unknown Problem" is printed by the **when others** due to the raise on line 9 when P is 0
- B "Success" is printed when  $0 < P < \text{Integer}'\text{Last}$
- C Trying to add 1 to P on line 7 generates a Constraint\_Error
- D Program\_Error will be raised by F if  $P < 0$  (no **return** statement found)

## Exceptions As Objects

# Exceptions Are Not Objects

- May not be manipulated
  - May not be components of composite types
  - May not be passed as parameters
- Some differences for scope and visibility
  - May be propagated out of scope

# But You Can Treat Them As Objects

- For raising and handling, and more
- Standard Library

```
package Ada.Exceptions is
  type Exception_Id is private;
  procedure Raise_Exception (E : Exception_Id;
                             Message : String := "");
  ...
  type Exception_Occurrence is limited private;
  function Exception_Name (X : Exception_Occurrence)
    return String;
  function Exception_Message (X : Exception_Occurrence)
    return String;
  function Exception_Information (X : Exception_Occurrence)
    return String;
  procedure Reraise_Occurrence (X : Exception_Occurrence);
  procedure Save_Occurrence (
    Target : out Exception_Occurrence;
    Source : Exception_Occurrence);
  ...
end Ada.Exceptions;
```

## Exception Occurrence

- Syntax associates an object with active exception

```
when defining_identifier : exception_name ... =>
```

- A constant view representing active exception
- Used with operations defined for the type

```
exception
```

```
when Caught_Exception : others =>  
    Put (Exception_Name (Caught_Exception));
```

# Exception\_Occurrence Query Functions

## ■ Exception\_Name

- Returns full expanded name of the exception in string form
  - Simple short name if space-constrained
- Predefined exceptions appear as just simple short name

## ■ Exception\_Message

- Returns string value specified when raised, if any

## ■ Exception\_Information

- Returns implementation-defined string content
- Should include both exception name and message content
- Presumably includes debugging information
  - Location where exception occurred
  - Language-defined check that failed (if such)

# Exception ID

- For an exception identifier, the *identity* of the exception is `<name>'Identity`

```
Mine : exception
use Ada.Exceptions;
...
exception
  when Occurrence : others =>
    if Exception_Identity (Occurrence) = Mine'Identity
    then
      ...
```

## Raise Expressions

## *Raise Expressions*

- **Expression** raising specified exception **at run-time**

```
Foo : constant Integer := (case X is  
    when 1 => 10,  
    when 2 => 20,  
    when others => raise Error);
```

Lab

# Exceptions Lab

## (Simplified) Input Verifier

- Overview
  - Create an application that converts strings to numeric values
- Requirements
  - Create a package to define your numeric type
  - Define a primitive to convert a string to your numeric type
    - The primitive should raise your own exceptions; one for out-of-range and one for illegal string
  - Main program should run multiple tests on the primitive

# Exceptions Lab Solution - Numeric Types

```
1 package Numeric_Types is
2   Illegal_String : exception;
3   Out_Of_Range   : exception;
4
5   Max_Int : constant := 2**15;
6   type Integer_T is range -(Max_Int) .. Max_Int - 1;
7
8   function Value (Str : String) return Integer_T;
9 end Numeric_Types;
10
11 package body Numeric_Types is
12
13   function Legal (C : Character) return Boolean is
14   begin
15     return
16       C in '0' .. '9' or C = '+' or C = '-' or C = '_' or C = 'e' or C = 'E';
17   end Legal;
18
19   function Value (Str : String) return Integer_T is
20   begin
21     for I in Str'Range loop
22       if not Legal (Str (I)) then
23         raise Illegal_String;
24       end if;
25     end loop;
26     return Integer_T'Value (Str);
27   exception
28     when Constraint_Error =>
29       raise Out_Of_Range;
30   end Value;
31
32 end Numeric_Types;
```

# Exceptions Lab Solution - Main

```
1 with Ada.Text_IO;
2 with Numeric_Types;
3 procedure Main is
4
5     procedure Print_Value (Str : String) is
6         Value : Numeric_Types.Integer_T;
7     begin
8         Ada.Text_IO.Put (Str & " => ");
9         Value := Numeric_Types.Value (Str);
10        Ada.Text_IO.Put_Line (Numeric_Types.Integer_T'Image (Value));
11    exception
12        when Numeric_Types.Out_Of_Range =>
13            Ada.Text_IO.Put_Line ("Out of range");
14        when Numeric_Types.Illegal_String =>
15            Ada.Text_IO.Put_Line ("Illegal entry");
16    end Print_Value;
17
18 begin
19     Print_Value ("123");
20     Print_Value ("2_3_4");
21     Print_Value ("-345");
22     Print_Value ("+456");
23     Print_Value ("1234567890");
24     Print_Value ("123abc");
25     Print_Value ("12e3");
26 end Main;
```

## Summary

## Exceptions Are Not Always Appropriate

- What does it mean to have an unexpected error in a safety-critical application?
  - Maybe there's no reasonable response



# Relying on Exception Raising Is Risky

- They may be **suppressed**
  - By runtime environment
  - By build switches
- Not recommended

```
function Tomorrow (Today : Days) return Days is
begin
  return Days'Succ (Today);
exception
  when Constraint_Error =>
    return Days'First;
end Tomorrow;
```

- Recommended

```
function Tomorrow (Today : Days) return Days is
begin
  if Today = Days'Last then
    return Days'First;
  else
    return Days'Succ (Today);
  end if;
end Tomorrow;
```

# Summary

- Should be for unexpected errors
- Give clients the ability to avoid them
- If handled, caller should see normal effect
  - Mode **out** parameters assigned
  - Function return values provided
- Package **Ada.Exceptions** provides views as objects
  - For both raising and special handling
  - Especially useful for debugging
- Checks may be suppressed

# Access Types

## Introduction

# Access Types Design

- A memory-addressed object is called an *access type*
- Objects are associated to *pools* of memory
  - With different allocation / deallocation policies
- Access objects are **guaranteed** to always be meaningful
  - In the absence of `Unchecked_Deallocation`
  - And if pool-specific

## ■ Ada

```
type Integer_Pool_Access  
  is access Integer;  
P_A : Integer_Pool_Access  
  := new Integer;
```

## ■ C++

```
int * P_C = malloc (sizeof (int));  
int * P_CPP = new int;  
int * G_C = &Some_Int;
```

```
type Integer_General_Access  
  is access all Integer;
```

```
G : aliased Integer;
```

```
G_A : Integer_General_Access := G'Access;
```

# Access Types - General vs Pool-Specific

## General Access Types

- Point to any object of designated type
- Useful for creating aliases to existing objects
- Point to existing object via 'Access **or** created by **new**
- No automatic memory management

## Pool-Specific Access Types

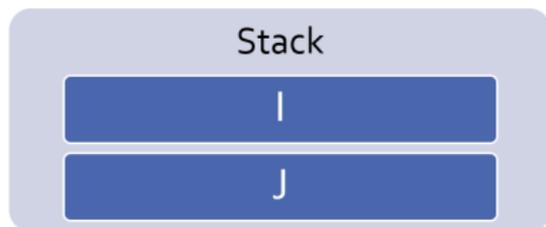
- Tightly coupled to dynamically allocated objects
- Used with Ada's controlled memory management (pools)
- Can only point to object created by **new**
- Memory management tied to specific storage pool

# Access Types Can Be Dangerous

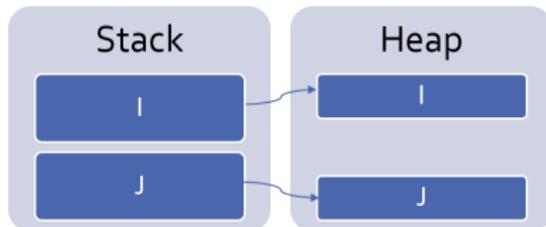
- Multiple memory issues
  - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot more expensive than stacking objects
- Ada avoids using accesses as much as possible
  - Arrays are not pointers
  - Parameters are implicitly passed by reference
- Only use them when needed

# Stack Vs Heap

```
I : Integer := 0;  
J : String := "Some Long String";
```



```
I : Access_Int := new Integer'(0);  
J : Access_Str := new String'("Some Long String");
```



## Access Types

# Declaration Location

- Can be at library level

```
package P is
  type String_Access is access String;
end P;
```

- Can be nested in a procedure

```
package body P is
  procedure Proc is
    type String_Access is access String;
  begin
    ...
  end Proc;
end P;
```

- Nesting adds non-trivial issues

- Creates a nested pool with a nested accessibility
- Don't do that unless you know what you are doing! (see later)

# Null Values

- A pointer that does not point to any actual data has a **null** value
- Access types have a default value of **null**
- **null** can be used in assignments and comparisons

**declare**

```
type Acc is access all Integer;
```

```
V : Acc;
```

**begin**

```
if V = null then
```

```
    -- will go here
```

```
end if;
```

```
V := new Integer'(0);
```

```
V := null; -- semantically correct, but memory leak
```

# Access Types and Primitives

- Subprogram using an access type are primitive of the **access type**
  - **Not** the type of the accessed object

```
type A_T is access all T;  
procedure Proc (V : A_T); -- Primitive of A_T, not T
```

- Primitive of the type can be created with the **access** mode
  - **Anonymous** access type
    - Details elsewhere

```
procedure Proc (V : access T); -- Primitive of T
```

## Dereferencing Access Types

- `.all` does the access dereference
  - Lets you access the object pointed to by the pointer
- `.all` is optional for
  - Access on a component of an array
  - Access on a component of a record

## Dereference Examples

```
type R is record
  F1, F2 : Integer;
end record;
type A_Int is access Integer;
type A_String is access all String;
type A_R is access R;
V_Int      : A_Int := new Integer;
V_String   : A_String := new String("abc");
V_R        : A_R := new R;

V_Int.all := 0;
V_String.all := "cde";
V_String (1) := 'z'; -- similar to V_String.all (1) := 'z';
V_R.all := (0, 0);
V_R.F1 := 1; -- similar to V_R.all.F1 := 1;
```

## Pool-Specific Access Types

## Pool-Specific Access Type

- An access type is a type

```
type T is [...]  
type T_Access is access T;  
V : T_Access := new T;
```

- Conversion is **not** possible between pool-specific access types

# Allocations

- Objects are created with the **new** reserved word
- The created object must be constrained
  - The constraint is given during the allocation

```
V : String_Access := new String (1 .. 10);
```

- The object can be created by copying an existing object - using a qualifier

```
V : String_Access := new String'("This is a String");
```

# Deallocations

- Deallocations are unsafe
  - Multiple deallocations problems
  - Memory corruptions
  - Access to deallocated objects
- As soon as you use them, you lose the safety of your access
- But sometimes, you have to do what you have to do ...
  - There's no simple way of doing it
  - Ada provides **Ada.Unchecked\_Deallocation**
  - Has to be instantiated (it's a generic)
  - Must work on an object, reset to **null** afterwards

## Deallocation Example

```
-- generic used to deallocate memory
with Ada.Unchecked_Deallocation;
procedure P is
  type An_Access is access A_Type;
  -- create instances of deallocation function
  -- (object type, access type)
  procedure Free is new Ada.Unchecked_Deallocation
    (A_Type, An_Access);
  V : An_Access := new A_Type;
begin
  Free (V);
  -- V is now null
end P;
```

## General Access Types

## General Access Types

- Can point to any pool (including stack)

```
type T is [...]  
type T_Access is access all T;  
V : T_Access := new T;
```

- Still distinct type
- Conversions are possible

```
type T_Access_2 is access all T;  
V2 : T_Access_2 := T_Access_2 (V); -- legal
```

## Referencing the Stack

- By default, stack-allocated objects cannot be referenced - and can even be optimized into a register by the compiler
- **aliased** declares an object to be referenceable through an access value

```
V : aliased Integer;
```

- 'Access attribute gives a reference to the object

```
A : Int_Access := V'Access;
```

- 'Unchecked\_Access does it **without checks**

# Aliased Objects Examples

```
type Acc is access all Integer;
V, G : Acc;
I : aliased Integer;
...
V := I'Access;
V.all := 5; -- Same as I := 5
...
procedure P1 is
  I : aliased Integer;
begin
  G := I'Unchecked_Access;
  P2;
  -- Necessary to avoid corruption
  -- Watch out for any of G's copies!
  G := null;
end P1;

procedure P2 is
begin
  G.all := 5;
end P2;
```

# Aliased Parameters

- To ensure a subprogram parameter always has a valid memory address, define it as **aliased**
  - Ensures 'Access and 'Address are valid for the parameter

```
procedure Example (Param : aliased Integer);
```

```
Object1 : aliased Integer;
```

```
Object2 : Integer;
```

```
-- This is OK
```

```
Example (Object1);
```

```
-- Compile error: Object2 could be optimized away
```

```
-- or stored in a register
```

```
Example (Object2);
```

```
-- Compile error: No address available for parameter
```

```
Example (123);
```

# Quiz

```
type One_T is access all Integer;  
type Two_T is access Integer;
```

```
A : aliased Integer;
```

```
B : Integer;
```

```
One : One_T;
```

```
Two : Two_T;
```

Which assignment(s) is (are) legal?

A. One := B'Access;

B. One := A'Access;

C. Two := B'Access;

D. Two := A'Access;

# Quiz

```
type One_T is access all Integer;  
type Two_T is access Integer;
```

```
A : aliased Integer;  
B : Integer;
```

```
One : One_T;  
Two : Two_T;
```

Which assignment(s) is (are) legal?

- A. One := B'Access;
- B. One := A'Access;
- C. Two := B'Access;
- D. Two := A'Access;

'Access is only allowed for general access types (One\_T). To use 'Access on an object, the object must be **aliased**.

## Accessibility Checks

# Introduction to Accessibility Checks (1/2)

- The **depth** of an object depends on its nesting within declarative scopes

```
package body P is
  -- Library level, depth 0
  O0 : aliased Integer;
  procedure Proc is
    -- Library level subprogram, depth 1
    type Acc1 is access all Integer;
    procedure Nested is
      -- Nested subprogram, enclosing + 1, here 2
      O2 : aliased Integer;
```

- Objects can be referenced by access **types** that are at **same depth or deeper**
  - An **access scope** must be  $\leq$  the object scope
- **type** Acc1 (depth 1) can access O0 (depth 0) but not O2 (depth 2)
- The compiler checks it statically
  - Removing checks is a workaround!
- Note: Subprogram library units are at **depth 1** and not 0

## Introduction to Accessibility Checks (2/2)

- Issues with nesting

```
package body P is
  type T0 is access all Integer;
  A0 : T0;
  V0 : aliased Integer;

  procedure Proc is
    type T1 is access all Integer;
    A1 : T1;
    V1 : aliased Integer;
  begin
    A0 := V0'Access;
    -- A0 := V1'Access; -- illegal
    A0 := V1'Unchecked_Access;
    A1 := V0'Access;
    A1 := V1'Access;
    A1 := T1 (A0);
    A1 := new Integer;
    -- A0 := T0 (A1); -- illegal
  end Proc;
end P;
```

- To avoid having to face these issues, avoid nested access types

# Dynamic Accessibility Checks

- Following the same rules
  - Performed dynamically by the runtime
- Lots of possible cases
  - New compiler versions may detect more cases
  - Using access always requires proper debugging and reviewing

```
procedure Main is
  type Acc is access all Integer;
  O : Acc;

  procedure Set_Value (V : access Integer) is
  begin
    O := Acc (V);
  end Set_Value;
begin
  declare
    O2 : aliased Integer := 2;
  begin
    Set_Value (O2'Access);
  end;
end Main;
```

## Getting Around Accessibility Checks

- Sometimes it is OK to use unsafe accesses to data
- 'Unchecked\_Access allows access to a variable of an incompatible accessibility level
- Beware of potential problems!

```
type Acc is access all Integer;  
G : Acc;  
procedure P is  
  V : aliased Integer;  
begin  
  G := V'Unchecked_Access;  
  ...  
  Do_Something (G.all);  
  G := null; -- This is "reasonable"  
end P;
```

## Using Access Types for Recursive Structures

- It is not possible to declare recursive structure
- But there can be an access to the enclosing type

```
type Cell; -- partial declaration
type Cell_Access is access all Cell;
type Cell is record -- full declaration
    Next      : Cell_Access;
    Some_Value : Integer;
end record;
```

# Quiz

```
type Global_Access_T is access all Integer;  
Global_Access : Global_Access_T;  
Global_Object : aliased Integer;  
procedure Proc_Access is  
    type Local_Access_T is access all Integer;  
    Local_Access : Local_Access_T;  
    Local_Object : aliased Integer;  
begin
```

Which assignment(s) is (are) legal?

- A. Global\_Access := Global\_Object'Access;
- B. Global\_Access := Local\_Object'Access;
- C. Local\_Access := Global\_Object'Access;
- D. Local\_Access := Local\_Object'Access;

# Quiz

```
type Global_Access_T is access all Integer;  
Global_Access : Global_Access_T;  
Global_Object : aliased Integer;  
procedure Proc_Access is  
    type Local_Access_T is access all Integer;  
    Local_Access : Local_Access_T;  
    Local_Object : aliased Integer;  
begin
```

Which assignment(s) is (are) legal?

- A. `Global_Access := Global_Object'Access;`
- B. `Global_Access := Local_Object'Access;`
- C. `Local_Access := Global_Object'Access;`
- D. `Local_Access := Local_Object'Access;`

Explanations

- A. Access type has same depth as object
- B. Access type is not allowed to have higher level than accessed object
- C. Access type has lower depth than accessed object
- D. Access type has same depth as object

## Memory Corruption

# Common Memory Problems (1/3)

- Uninitialized pointers

```
declare
  type An_Access is access all Integer;
  V : An_Access;
begin
  V.all := 5; -- constraint error
```

- Double deallocation

```
declare
  type An_Access is access all Integer;
  procedure Free is new
    Ada.Unchecked_Deallocation (Integer, An_Access);
  V1 : An_Access := new Integer;
  V2 : An_Access := V1;
begin
  Free (V1);
  ...
  Free (V2);
```

- May raise `Storage_Error` if memory is still protected (unallocated)
- May deallocate a different object if memory has been reallocated
  - Putting that object in an inconsistent state

## Common Memory Problems (2/3)

- Accessing deallocated memory

```
declare
```

```
  type An_Access is access all Integer;
```

```
  procedure Free is new
```

```
    Ada.Unchecked_Deallocation (Integer, An_Access);
```

```
  V1 : An_Access := new Integer;
```

```
  V2 : An_Access := V1;
```

```
begin
```

```
  Free (V1);
```

```
  ...
```

```
  V2.all := 5;
```

- May raise `Storage_Error` if memory is still protected (unallocated)
- May modify a different object if memory has been reallocated (putting that object in an inconsistent state)

## Common Memory Problems (3/3)

- Memory leaks

```
declare
```

```
  type An_Access is access all Integer;
```

```
  procedure Free is new
```

```
    Ada.Unchecked_Deallocation (Integer, An_Access);
```

```
  V : An_Access := new Integer;
```

```
begin
```

```
  V := null;
```

- Silent problem

- Might raise `Storage_Error` if too many leaks
- Might slow down the program if too many page faults

## How to Fix Memory Problems?

- There is no language-defined solution
- Use the debugger!
- Use additional tools
  - `gnatmem` monitor memory leaks
  - `valgrind` monitor all the dynamic memory
  - **GNAT.Debug\_Pools** gives a pool for an access type, raising explicit exception in case of invalid access
  - Others...

## Anonymous Access Types

# Anonymous Access Parameters

- Parameter modes are of 4 types: **in**, **out**, **in out**, **access**
- The access mode is called *anonymous access type*
  - Anonymous access is implicitly general (no need for **all**)
- When used:
  - Any named access can be passed as parameter
  - Any anonymous access can be passed as parameter

```
type Acc is access all Integer;  
Aliased_Integer : aliased Integer;  
Access_Object   : Acc := Aliased_Integer'Access;  
procedure P1 (Anon_Access : access Integer) is null;  
procedure P2 (Access_Parameter : access Integer) is  
begin  
    P1 (Aliased_Integer'Access);  
    P1 (Access_Object);  
    P1 (Access_Parameter);  
end P2;
```

## Anonymous Access Types

- Other places can declare an anonymous access

```
function F return access Integer;  
V : access Integer;  
type T (V : access Integer) is record  
  C : access Integer;  
end record;  
type A is array (Integer range <>) of access Integer;
```

- Do not use them without a clear understanding of accessibility check rules

# Anonymous Access Constants

- **constant** (instead of **all**) denotes an access type through which the referenced object cannot be modified

```
type CAcc is access constant Integer;  
G1 : aliased Integer;  
G2 : aliased constant Integer := 123;  
V1 : CAcc := G1'Access;  
V2 : CAcc := G2'Access;  
V1.all := 0; -- illegal
```

- **not null** denotes an access type for which null value cannot be accepted
  - Available in Ada 2005 and later

```
type NAcc is not null access Integer;  
V : NAcc := null; -- illegal
```

- Also works for subprogram parameters

```
procedure Bar (V1 : access constant Integer);  
procedure Foo (V1 : not null access Integer); -- Ada 2005
```

Lab

# Access Types Lab

## ■ Overview

- Create a (really simple) Password Manager
  - The Password Manager should store the password and a counter for each of some number of logins
  - As it's a Password Manager, you want to modify the data directly (not pass the information around)

## ■ Requirements

- Create a Password Manager package
  - Create a record to store the password string and the counter
  - Create an array of these records indexed by the login identifier
  - The user should be able to retrieve a pointer to the record, either for modification or for viewing
- Main program should:
  - Set passwords and initial counter values for many logins
  - Print password and counter value for each login

## ■ Hint

- Password is a string of varying length
  - Easiest way to do this is a pointer to a string that gets initialized to the correct length

# Access Types Lab Solution - Password Manager

```
package Password_Manager is

    type Login_T is (Email, Banking, Amazon, Streaming);
    type Password_T is record
        Count      : Natural;
        Password   : access String;
    end record;

    type Modifiable_T is access all Password_T;
    type Viewable_T is access constant Password_T;

    function Update (Login : Login_T) return Modifiable_T;
    function View (Login : Login_T) return Viewable_T;

end Password_Manager;

package body Password_Manager is

    Passwords : array (Login_T) of aliased Password_T;

    function Update (Login : Login_T) return Modifiable_T is
        (Passwords (Login)'Access);
    function View (Login : Login_T) return Viewable_T is
        (Passwords (Login)'Access);

end Password_Manager;
```

# Access Types Lab Solution - Main

```
1  with Ada.Text_IO;          use Ada.Text_IO;
2  with Password_Manager; use Password_Manager;
3  procedure Main is
4
5      procedure Update (Which : Password_Manager.Login_T;
6                       Pw     : String;
7                       Count  : Natural) is
8
9          begin
10             Update (Which).Password := new String'(Pw);
11             Update (Which).Count   := Count;
12         end Update;
13
14     begin
15         Update (Email, "QWE!@#", 1);
16         Update (Banking, "asd123", 22);
17         Update (Amazon, "098poi", 333);
18         Update (Streaming, ")(*LKJ", 444);
19
20         for Login in Login_T'Range loop
21             Put_Line
22                 (Login'Image & " => " & View (Login).Password.all &
23                 View (Login).Count'Image);
24         end loop;
25     end Main;
```

## Summary

# Summary

- Access types are the same as C/C++ pointers
- There are usually better ways of memory management
  - Language has its own ways of dealing with large objects passed as parameters
  - Language has libraries dedicated to memory allocation / deallocation
- At a minimum, create your own generics to do allocation / deallocation
  - Minimize memory leakage and corruption

# Genericity

# Introduction

# The Notion of a Pattern

- Sometimes algorithms can be abstracted from types and subprograms

```
procedure Swap_Int (Left, Right : in out Integer) is
  V : Integer := Left;
begin
  Left := Right;
  Right := V;
end Swap_Int;
```

```
procedure Swap_Bool (Left, Right : in out Boolean) is
  V : Boolean := Left;
begin
  Left := Right;
  Right := V;
end Swap_Bool;
```

- It would be nice to extract these properties in some common pattern, and then just replace the parts that need to be replaced

```
procedure Swap (Left, Right : in out (Integer | Boolean)) is
  V : (Integer | Boolean) := Left;
begin
  Left := Right;
  Right := V;
end Swap;
```

## Solution: Generics

- A *generic unit* is a unit that does not exist
- It is a pattern based on properties
- The instantiation applies the pattern to certain parameters

## Ada Generic Compared to C++ Template

### Ada Generic

```
-- specification
generic
  type T is private;
  procedure Swap (L, R : in out T);

-- implementation
procedure Swap (L, R : in out T) is
  Tmp : T := L;
begin
  L := R;
  R := Tmp;
end Swap;

-- instance
procedure Swap_F is new Swap (Float);
```

### C++ Template

```
// prototype
template <class T>
void Swap (T & L, T & R);

// implementation
template <class T>
void Swap (T & L, T & R) {
  T Tmp = L;
  L = R;
  R = Tmp;
}

// instance
int x, y;
Swap<int>(x,y);
```

## Creating Generics

# Declaration

- Subprograms

```
generic
  type T is private;
procedure Swap (L, R : in out T);
```

- Packages

```
generic
  type T is private;
package Stack is
  procedure Push (Item : T);
end Stack;
```

- Body is required

- Will be specialized and compiled for **each instance**

- Children of generic units have to be generic themselves

```
generic
package Stack.Utilities is
  procedure Print (S : Stack_T);
```

# Usage

- Instantiated with the **new** keyword

```
-- Standard library
```

```
function Convert is new Ada.Unchecked_Conversion  
  (Integer, Array_Of_4_Bytes);
```

```
-- Callbacks
```

```
procedure Parse_Tree is new Tree_Parser  
  (Visitor_Procedure);
```

```
-- Containers, generic data-structures
```

```
package Integer_Stack is new Stack (Integer);
```

- Advanced usages for testing, proof, meta-programming

# Quiz

Which one(s) of the following can be made generic?

**generic**

```
type T is private;
```

<code goes here>

- A. package
- B. record
- C. function
- D. array

# Quiz

Which one(s) of the following can be made generic?

`generic`

```
    type T is private;
```

<code goes here>

- A. *package*
- B. record
- C. *function*
- D. array

Only packages, functions, and procedures, can be made generic.

## Generic Data

## Generic Types Parameters (1/3)

- A generic parameter is a template
- It specifies the properties the generic body can rely on

```
generic
  type T1 is private;
  type T2 (<>) is private;
  type T3 is limited private;
package Parent is
```

- The actual parameter must be no more restrictive than the *generic contract*

## Generic Types Parameters (2/3)

- Generic formal parameter tells generic what it is allowed to do with the type

---

<code>type T1 is (&lt;&gt;);</code>	Discrete type; 'First, 'Succ, etc available
<code>type T2 is range &lt;&gt;;</code>	Signed Integer type; appropriate mathematic operations allowed
<code>type T3 is digits &lt;&gt;;</code>	Floating point type; appropriate mathematic operations allowed
<code>type T4;</code>	Incomplete type; can only be used as target of <code>access</code>
<code>type T5 is tagged private;</code>	<code>tagged</code> type; can extend the type
<code>type T6 is private;</code>	No knowledge about the type other than assignment, comparison, object creation allowed
<code>type T7 (&lt;&gt;) is private;</code>	<code>(&lt;&gt;)</code> indicates type can be unconstrained, so any object has to be initialized

---

## Generic Types Parameters (3/3)

- The usage in the generic has to follow the contract

- Generic Subprogram

```
generic
  type T (<>) is private;
procedure P (V : T);
procedure P (V : T) is
  X1 : T := V; -- OK, can constrain by initialization
  X2 : T;      -- Compilation error, no constraint to this
begin
```

- Instantiations

```
type Limited_T is limited null record;

-- unconstrained types are accepted
procedure P1 is new P (String);

-- type is already constrained
-- (but generic will still always initialize objects)
procedure P2 is new P (Integer);

-- Illegal: the type can't be limited because the generic
-- thinks it can make copies
procedure P3 is new P (Limited_T);
```

# Generic Parameters Can Be Combined

- Consistency is checked at compile-time

**generic**

```
type T (<>) is private;
type Acc is access all T;
type Index is (<>);
type Arr is array (Index range <>) of Acc;
```

```
function Component (Source : Arr;
                   Position : Index)
return T;
```

```
type String_Ptr is access all String;
type String_Array is array (Integer range <>)
of String_Ptr;
```

```
function String_Component is new Component
(T      => String,
 Acc    => String_Ptr,
 Index  => Integer,
 Arr    => String_Array);
```

# Quiz

```
generic
  type T1 is (<>);
  type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is (are) legal instantiation(s)?

- A. procedure A is new G (String, Character);
- B. procedure B is new G (Character, Integer);
- C. procedure C is new G (Integer, Boolean);
- D. procedure D is new G (Boolean, String);

# Quiz

```
generic
  type T1 is (<>);
  type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is (are) legal instantiation(s)?

- A. `procedure A is new G (String, Character);`
- B. `procedure B is new G (Character, Integer);`
- C. `procedure C is new G (Integer, Boolean);`
- D. `procedure D is new G (Boolean, String);`

T1 must be discrete - so an integer or an enumeration. T2 can be any type

## Generic Formal Data

# Generic Constants/Variables As Parameters

- Variables can be specified on the generic contract
- The mode specifies the way the variable can be used:
  - **in** → read only
  - **in out** → read write
- Generic variables can be defined after generic types

- Generic package

```
generic
  type Component_T is private;
  Array_Size      : Positive;
  High_Watermark : in out Component_T;
package Repository is
```

- Generic instance

```
V : Float;
Max : Float;
```

```
procedure My_Repository is new Repository
(Component_T => Float,
 Array_size  => 10,
 High_Watermark => Max);
```

# Generic Subprogram Parameters

- Subprograms can be defined in the generic contract
- Must be introduced by **with** to differ from the generic unit

```
generic
  type T is private;
  with function Less_Than (L, R : T) return Boolean;
function Max (L, R : T) return T;

function Max (L, R : T) return T is
begin
  if Less_Than (L, R) then
    return R;
  else
    return L;
  end if;
end Max;

type Something_T is null record;
function Less_Than (L, R : Something_T) return Boolean;
procedure My_Max is new Max (Something_T, Less_Than);
```

## Generic Subprogram Parameters Defaults

- `is <>` - matching subprogram is taken by default
- `is null` - null procedure is taken by default
  - Only available in Ada 2005 and later

`generic`

```
type T is private;
```

```
with function Is_Valid (P : T) return Boolean is <>;
```

```
with procedure Error_Message (P : T) is null;
```

```
procedure Validate (P : T);
```

```
function Is_Valid_Record (P : Record_T) return Boolean;
```

```
procedure My_Validate is new Validate (Record_T,  
                                     Is_Valid_Record);
```

```
-- Is_Valid maps to Is_Valid_Record
```

```
-- Error_Message maps to a null procedure
```

# Quiz

```
generic
  type Component_T is (<>);
  Last : in out Component_T;
procedure Write (P : Component_T);
```

```
Numeric      : Integer;
Enumerated   : Boolean;
Floating_Point : Float;
```

Which of the following piece(s) of code is (are) legal?

- A. procedure Write\_A is new Write (Integer, Numeric)
- B. procedure Write\_B is new Write (Boolean, Enumerated)
- C. procedure Write\_C is new Write (Integer, Integer'Pos (Numeric))
- D. procedure Write\_D is new Write (Float, Floating\_Point)

# Quiz

```
generic
  type Component_T is (<>);
  Last : in out Component_T;
procedure Write (P : Component_T);
```

```
Numeric      : Integer;
Enumerated   : Boolean;
Floating_Point : Float;
```

Which of the following piece(s) of code is (are) legal?

- A. `procedure Write_A is new Write (Integer, Numeric)`
  - B. `procedure Write_B is new Write (Boolean, Enumerated)`
  - C. `procedure Write_C is new Write (Integer, Integer'Pos (Numeric))`
  - D. `procedure Write_D is new Write (Float, Floating_Point)`
- 
- A. Legal
  - B. Legal
  - C. The second generic parameter has to be a variable
  - D. The first generic parameter has to be discrete

# Quiz

Given the following generic function:

```
generic
  type Some_T is private;
  with function "+" (L : Some_T; R : Integer) return Some_T is <>;
function Incr (Param : Some_T) return Some_T;

function Incr (Param : Some_T) return Some_T is
begin
  return Param + 1;
end Incr;
```

And the following declarations:

```
type Record_T is record
  Component : Integer;
end record;
function Add (L : Record_T; I : Integer) return Record_T is
  ((Component => L.Component + I))
function Weird (L : Integer; R : Integer) return Integer is (0);
```

Which of the following instantiation(s) is/are **not** legal?

- function IncrA is new Incr (Integer, Weird);
- function IncrB is new Incr (Record\_T, Add);
- function IncrC is new Incr (Record\_T);
- function IncrD is new Incr (Integer);

## Quiz

Given the following generic function:

```
generic
  type Some_T is private;
  with function "+" (L : Some_T; R : Integer) return Some_T is <>;
function Incr (Param : Some_T) return Some_T;

function Incr (Param : Some_T) return Some_T is
begin
  return Param + 1;
end Incr;
```

And the following declarations:

```
type Record_T is record
  Component : Integer;
end record;
function Add (L : Record_T; I : Integer) return Record_T is
  ((Component => L.Component + I))
function Weird (L : Integer; R : Integer) return Integer is (0);
```

Which of the following instantiation(s) is/are **not** legal?

- function IncrA is new Incr (Integer, Weird);
- function IncrB is new Incr (Record\_T, Add);
- function IncrC is new Incr (Record\_T);
- function IncrD is new Incr (Integer);

with function "+" (L : Some\_T; R : Integer) return Some\_T is <>;  
indicates that if no function for + is passed in, find (if possible) a matching definition at the point of instantiation.

- Weird matches the subprogram profile, so Incr will use Weird when doing addition for Integer
- Add matches the subprogram profile, so Incr will use Add when doing the addition for Record\_T
- There is no matching + operation for Record\_T, so that instantiation fails to compile
- Because there is no parameter for the generic formal parameter +, the compiler will look for one in the scope of the instantiation. Because the instantiating type is numeric, the inherited + operator is found

## Generic Completion

## Implications at Compile-Time

- The body needs to be visible when compiling the user code
- Therefore, when distributing a component with generics to be instantiated, the code of the generic must come along

## Generic and Freezing Points

- A generic type **freezes** the type and needs the **full view**
- May force separation between its declaration (in spec) and instantiations (in private or body)

```
generic
```

```
  type X is private;
```

```
package Base is
```

```
  V : access X;
```

```
end Base;
```

```
package P is
```

```
  type X is private;
```

```
  -- illegal
```

```
  package B is new Base (X);
```

```
private
```

```
  type X is null record;
```

```
end P;
```

## Generic Incomplete Parameters

- A generic type can be incomplete
- Allows generic instantiations before full type definition
- Restricts the possible usages (only **access**)

```
generic
```

```
  type X; -- incomplete
```

```
package Base is
```

```
  V : access X;
```

```
end Base;
```

```
package P is
```

```
  type X is private;
```

```
  -- legal
```

```
  package B is new Base (X);
```

```
private
```

```
  type X is null record;
```

```
end P;
```

# Quiz

```
generic
  type T1;
  A1 : access T1;
  type T2 is private;
  A2, B2 : T2;
procedure G_P;
procedure G_P is
begin
  -- Complete here
end G_P;
```

Which of the following statement(s) is (are) legal for G\_P's body?

- A. pragma Assert (A1 /= null)
- B. pragma Assert (A1.all'Size > 32)
- C. pragma Assert (A2 = B2)
- D. pragma Assert (A2 - B2 /= 0)

## Quiz

```
generic
  type T1;
  A1 : access T1;
  type T2 is private;
  A2, B2 : T2;
procedure G_P;
procedure G_P is
begin
  -- Complete here
end G_P;
```

Which of the following statement(s) is (are) legal for G\_P's body?

- A. `pragma Assert (A1 /= null)`
- B. `pragma Assert (A1.all'Size > 32)`
- C. `pragma Assert (A2 = B2)`
- D. `pragma Assert (A2 - B2 /= 0)`

# Genericity Lab

## ■ Requirements

- Create a record structure containing multiple components
  - Need subprograms to convert the record to a string, and compare the order of two records
  - Lab prompt package `Data_Type` contains a framework
- Create a generic list implementation
  - Need subprograms to add items to the list, sort the list, and print the list
- The **main** program should:
  - Add many records to the list
  - Sort the list
  - Print the list

## ■ Hints

- Sort routine will need to know how to compare components
- Print routine will need to know how to print one component

# Genericity Lab Solution - Generic (Spec)

```
1  generic
2      type Component_T is private;
3      Max_Size : Natural;
4      with function ">" (L, R : Component_T) return Boolean is <>;
5      with function Image (Component : Component_T) return String;
6  package Generic_List is
7
8      type List_T is private;
9
10     procedure Add (This : in out List_T;
11                   Item : in    Component_T);
12     procedure Sort (This : in out List_T);
13     procedure Print (List : List_T);
14
15 private
16     subtype Index_T is Natural range 0 .. Max_Size;
17     type List_Array_T is array (1 .. Index_T'Last) of Component_T;
18
19     type List_T is record
20         Values : List_Array_T;
21         Length : Index_T := 0;
22     end record;
23 end Generic_List;
```

# Genericity Lab Solution - Generic (Body)

```
1 with Ada.Text_io; use Ada.Text_IO;
2 package body Generic_List is
3
4   procedure Add (This : in out List_T;
5                 Item : in   Component_T) is
6   begin
7     This.Length      := This.Length + 1;
8     This.Values (This.Length) := Item;
9   end Add;
10
11  procedure Sort (This : in out List_T) is
12    Temp : Component_T;
13  begin
14    for I in 1 .. This.Length loop
15      for J in 1 .. This.Length - I loop
16        if This.Values (J) > This.Values (J + 1) then
17          Temp          := This.Values (J);
18          This.Values (J) := This.Values (J + 1);
19          This.Values (J + 1) := Temp;
20        end if;
21      end loop;
22    end loop;
23  end Sort;
24
25  procedure Print (List : List_T) is
26  begin
27    for I in 1 .. List.Length loop
28      Put_Line (Integer'Image (I) & " " & Image (List.Values (I)));
29    end loop;
30  end Print;
31
32 end Generic_List;
```

# Genericity Lab Solution - Main

```
1 with Data_Type;
2 with Generic_List;
3 procedure Main is
4     package List is new Generic_List (Component_T => Data_Type.Record_T,
5                                       Max_Size => 20,
6                                       ">" => Data_Type.">",
7                                       Image => Data_Type.Image);
8
9     My_List : List.List_T;
10    Component : Data_Type.Record_T;
11
12 begin
13     List.Add (My_List, (Integer_Component => 111,
14                       Character_Component => 'a'));
15     List.Add (My_List, (Integer_Component => 111,
16                       Character_Component => 'z'));
17     List.Add (My_List, (Integer_Component => 111,
18                       Character_Component => 'A'));
19     List.Add (My_List, (Integer_Component => 999,
20                       Character_Component => 'B'));
21     List.Add (My_List, (Integer_Component => 999,
22                       Character_Component => 'Y'));
23     List.Add (My_List, (Integer_Component => 999,
24                       Character_Component => 'b'));
25     List.Add (My_List, (Integer_Component => 112,
26                       Character_Component => 'a'));
27     List.Add (My_List, (Integer_Component => 998,
28                       Character_Component => 'z'));
29
30     List.Sort (My_List);
31     List.Print (My_List);
32 end Main;
```

## Summary

# Generic Routines Vs Common Routines

```
package Helper is
  type Float_T is digits 6;
  generic
    type Type_T is digits <>;
    Min : Type_T;
    Max : Type_T;
  function In_Range_Generic (X : Type_T) return Boolean;
  function In_Range_Common (X : Float_T;
                           Min : Float_T;
                           Max : Float_T)
    return Boolean;
end Helper;

procedure User is
  type Speed_T is new Float_T range 0.0 .. 100.0;
  B : Boolean;
  function Valid_Speed is new In_Range_Generic
    (Speed_T, Speed_T'First, Speed_T'Last);
begin
  B := Valid_Speed (12.3);
  B := In_Range_Common (12.3, Speed_T'First, Speed_T'Last);
```

# Summary

- Generics are useful for copying code that works the same just for different types
  - Sorting, containers, etc
- Properly written generics only need to be tested once
  - But testing / debugging can be more difficult
- Generic instantiations are best done at compile time
  - At the package level
  - Can be run time expensive when done in subprogram scope

# Tasking

# Introduction

# Concurrency Mechanisms

- Task
  - **Active**
  - Rendezvous: **Client / Server** model
  - Server **entries**
  - Client **entry calls**
  - Typically maps to OS threads
- Protected object
  - **Passive**
  - *Monitors* protected data
  - **Restricted** set of operations
  - Concurrency-safe **semantics**
  - No thread overhead
  - Very portable
- Object-Oriented
  - **Synchronized** interfaces
  - Protected objects inheritance

# A Simple Task

- Concurrent code execution via **task**
- **limited** types (No copies allowed)

```
procedure Main is
  task type Simple_Task_T;
  task body Simple_Task_T is
  begin
    loop
      delay 1.0;
      Put_Line ("T");
    end loop;
  end Simple_Task_T;
  Simple_Task : Simple_Task_T;
  -- This task starts when Simple_Task is elaborated
begin
  loop
    delay 1.0;
    Put_Line ("Main");
  end loop;
end;
```

- A task is started when its declaration scope is **elaborated**
- Its enclosing scope exits when **all tasks** have finished

# Tasks

# Rendezvous Definitions

- **Server** declares several **entry**
- Client calls entries like subprograms
- Server **accept** the client calls
- At each standalone **accept**, server task **blocks**
  - **Until** a client calls the related **entry**

```
task type Msg_Box_T is
  entry Start;
  entry Receive_Message (S : String);
end Msg_Box_T;

task body Msg_Box_T is
begin
  loop
    accept Start;
    Put_Line ("start");

    accept Receive_Message (S : String) do
      Put_Line ("receive " & S);
    end Receive_Message;
  end loop;
end Msg_Box_T;
```

```
T : Msg_Box_T;
```

## Rendezvous Entry Calls

- Upon calling an **entry**, client **blocks**
  - **Until** server reaches **end** of its **accept** block

```
Put_Line ("calling start");
T.Start;
Put_Line ("calling receive 1");
T.Receive_Message ("1");
Put_Line ("calling receive 2");
T.Receive_Message ("2");
```

- May be executed as follows:

```
calling start
start           -- May switch place with line below
calling receive 1 -- May switch place with line above
receive 1
calling receive 2
-- Blocked until another task calls Start
```

# Rendezvous with a Task

- **accept** statement
  - Wait on single entry
  - If entry call waiting: Server handles it
  - Else: Server **waits** for an entry call
- **select** statement
  - **Several** entries accepted at the **same time**
  - Can **time-out** on the wait
  - Can be **not blocking** if no entry call waiting
  - Can **terminate** if no clients can **possibly** make entry call
  - Can **conditionally** accept a rendezvous based on a **guard expression**

## Protected Objects

# Protected Objects

- **Multitask-safe** accessors to get and set state
- **No** direct state manipulation
- **No** concurrent modifications
- **limited** types (No copies allowed)

## Protected: Functions and Procedures

- A **function** can **get** the state
  - **Multiple-Readers**
  - Protected data is **read-only**
  - Concurrent call to **function** is **allowed**
  - **No** concurrent call to **procedure**
- A **procedure** can **set** the state
  - **Single-Writer**
  - **No** concurrent call to either **procedure** or **function**
  - In case of concurrency, other callers get **blocked**
    - Until call finishes

# Example

```
protected type Protected_Value is
  procedure Set (V : Integer);
  function Get return Integer;
private
  Value : Integer;
end Protected_Value;
```

```
protected body Protected_Value is
  procedure Set (V : Integer) is
  begin
    Value := V;
  end Set;

  function Get return Integer is
  begin
    return Value;
  end Get;
end Protected_Value;
```

## Delays

## Delay Keyword

- **delay** keyword part of tasking
- Blocks for a time
- Relative: Blocks for at least Duration
- Absolute: Blocks until no earlier than Calendar.Time or Real\_Time.Time

```
with Calendar;
```

```
procedure Main is
```

```
    Relative : Duration := 1.0;
```

```
    Absolute : Calendar.Time
```

```
        := Calendar.Time_Of (2030, 10, 01);
```

```
begin
```

```
    delay Relative;
```

```
    delay until Absolute;
```

```
end Main;
```

## Task and Protected Types

# Task Activation

- Instantiated tasks start running when **activated**
- On the **stack**
  - When **enclosing** declarative part finishes **elaborating**
- On the **heap**
  - **Immediately** at instantiation

```
task type First_T is ...
type First_T_A is access all First_T;

task body First_T is ...
...
declare
    V1 : First_T;
    V2 : First_T_A;
begin -- V1 is activated
    V2 := new First_T; -- V2 is activated immediately
```

# Single Declaration

- Instantiate an **anonymous** task (or protected) type
- Declares an object of that type

```
task type Task_T is
    entry Start;
end Task_T;
```

```
type Task_Ptr_T is access all Task_T;
```

```
task body Task_T is
begin
    accept Start;
end Task_T;
```

```
...
```

```
V1 : Task_T;
```

```
V2 : Task_Ptr_T;
```

```
begin
```

```
V1.Start;
```

```
V2 := new Task_T;
```

```
V2.all.Start;
```

# Task Scope

- Nesting is possible in **any** declarative block
- Scope has to **wait** for tasks to finish before ending
- At library level: program ends only when **all tasks** finish

```
package P is
  task type T;
end P;
```

```
package body P is
  task body T is
    loop
      delay 1.0;
      Put_Line ("tick");
    end loop;
  end T;
```

```
Task_Instance : T;
end P;
```

## Some Advanced Concepts

# Waiting on Multiple Entries

- **select** can wait on multiple entries
  - With **equal** priority, regardless of declaration order

```
loop
  select
    accept Receive_Message (V : String)
    do
      Put_Line ("Message : " & V);
    end Receive_Message;
  or
    accept Stop;
    exit;
  end select;
end loop;

...
T.Receive_Message ("A");
T.Receive_Message ("B");
T.Stop;
```

# Waiting with a Delay

- A **select** statement may **time-out** using **delay** or **delay until**
  - Resume execution at next statement
- Multiple **delay** allowed
  - Useful when the value is not hard-coded

```
loop
  select
    accept Receive_Message (V : String) do
      Put_Line ("Message : " & V);
    end Receive_Message;
  or
    delay 50.0;
    Put_Line ("Don't wait any longer");
    exit;
  end select;
end loop;
```

*Task will wait up to 50 seconds for Receive\_Message. If no message is received, it will write to the console, and then restart the loop. (If the **exit** wasn't there, the loop would exit the first time no message was received.)*

## Calling an Entry with a Delay Protection

- A call to **entry** **blocks** the task until the entry is **accept**'ed
- Wait for a **given amount of time** with **select ... delay**
- Only **one** entry call is allowed
- No **accept** statement is allowed

```
task Msg_Box is
    entry Receive_Message (V : String);
end Msg_Box;
```

```
procedure Main is
begin
    select
        Msg_Box.Receive_Message ("A");
    or
        delay 50.0;
    end select;
end Main;
```

*Procedure will wait up to 50 seconds for Receive\_Message to be accepted before it gives up*

# Non-blocking Accept or Entry

- Using **else**
  - Task **skips** the **accept** or **entry** call if they are **not ready** to be entered
- **delay** is **not** allowed in this case

```
select
  accept Receive_Message (V : String) do
    Put_Line ("Received : " & V);
  end Receive_Message;
else
  Put_Line ("Nothing to receive");
end select;
```

[...]

```
select
  T.Receive_Message ("A");
else
  Put_Line ("Receive message not called");
end select;
```

# Queue

- Protected **entry** or **procedure** and tasks **entry** are activated by **one** task at a time
- **Mutual exclusion** section
- Other tasks trying to enter are **queued**
  - In **First-In First-Out** (FIFO) by default
- When the server task **terminates**, tasks still queued receive `Tasking_Error`

# Advanced Tasking

Other constructions are available

- **Guard condition** on **accept**
- **requeue** to **defer** handling of an **entry** call
- **terminate** the task when no **entry** call can happen anymore
- **abort** to stop a task immediately
- **select ... then abort** some other task

Lab

# Tasking Lab

- Requirements
  - Create multiple tasks with the following attributes
    - Startup entry receives some identifying information and a delay length
    - Stop entry will end the task
    - Until stopped, the task will send it's identifying information to a monitor periodically based on the delay length
  - Create a protected object that stores the identifying information of task that called it
  - Main program should periodically check the protected object, and print when it detects a task switch
    - I.e. If the current task is different than the last printed task, print the identifying information for the current task

# Tasking Lab Solution - Protected Object

```
1  with Task_Type;
2  package Protected_Object is
3      protected Monitor is
4          procedure Set (Id : Task_Type.Task_Id_T);
5          function Get return Task_Type.Task_Id_T;
6      private
7          Value : Task_Type.Task_Id_T;
8      end Monitor;
9  end Protected_Object;
10
11 package body Protected_Object is
12     protected body Monitor is
13         procedure Set (Id : Task_Type.Task_Id_T) is
14             begin
15                 Value := Id;
16             end Set;
17         function Get return Task_Type.Task_Id_T is (Value);
18     end Monitor;
19 end Protected_Object;
```

# Tasking Lab Solution - Task Type

```
1 package Task_Type is
2   type Task_Id_T is range 1_000 .. 9_999;
3   task type Task_T is
4     entry Start_Task (Task_Id      : Task_Id_T;
5                      Delay_Duration : Duration);
6     entry Stop_Task;
7   end Task_T;
8 end Task_Type;
9
10 with Protected_Object;
11 package body Task_Type is
12   task body Task_T is
13     Wait_Time : Duration;
14     Id        : Task_Id_T;
15   begin
16     accept Start_Task (Task_Id      : Task_Id_T;
17                      Delay_Duration : Duration) do
18       Wait_Time := Delay_Duration;
19       Id        := Task_Id;
20     end Start_Task;
21     loop
22       select
23         accept Stop_Task;
24         exit;
25       or
26         delay Wait_Time;
27         Protected_Object.Monitor.Set (Id);
28       end select;
29     end loop;
30   end Task_T;
31 end Task_Type;
```

# Tasking Lab Solution - Main

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Protected_Object;
3  with Task_Type;
4  procedure Main is
5      T1, T2, T3      : Task_Type.Task_T;
6      Last_Id, This_Id : Task_Type.Task_Id_T := Task_Type.Task_Id_T'Last;
7      use type Task_Type.Task_Id_T;
8  begin
9
10     T1.Start_Task (1_111, 0.3);
11     T2.Start_Task (2_222, 0.5);
12     T3.Start_Task (3_333, 0.7);
13
14     for Count in 1 .. 20 loop
15         This_Id := Protected_Object.Monitor.Get;
16         if Last_Id /= This_Id then
17             Last_Id := This_Id;
18             Put_Line (Count'Image & "> " & Last_Id'Image);
19         end if;
20         delay 0.2;
21     end loop;
22
23     T1.Stop_Task;
24     T2.Stop_Task;
25     T3.Stop_Task;
26
27 end Main;
```

## Summary

# Summary

- Tasks are **language-based** concurrency mechanisms
  - Typically implemented as threads
  - Not necessarily for **truly** parallel operations
  - Originally for task-switching / time-slicing
- Multiple mechanisms to **synchronize** tasks
  - Delay
  - Rendezvous
  - Queues
  - Protected Objects

# Interfacing with C

# Introduction

# Introduction

- Lots of C code out there already
  - Maybe even a lot of reusable code in your own repositories
- Need a way to interface Ada code with existing C libraries
  - Built-in mechanism to define ability to import objects from C or export Ada objects
- Passing data between languages can cause issues
  - Sizing requirements
  - Passing mechanisms (by reference, by copy)

## Import / Export

## Import / Export Aspects (1/2)

- Aspects `Import` and `Export` allow Ada and C to interact
  - `Import` indicates a subprogram imported into Ada
  - `Export` indicates a subprogram exported from Ada
- Need aspects defining calling convention and external name
  - `Convention => C` tells linker to use C-style calling convention
  - `External_Name => "<name>"` defines object name for linker
- Ada implementation

```
procedure Imported_From_C with
  Import,
  Convention    => C,
  External_Name => "SomeProcedureInC";

procedure Exported_To_C with
  Export,
  Convention    => C,
  External_Name => "some_ada_procedure";
```

- C implementation

```
void SomeProcedureInC (void) {
    // some code
}

extern void ada_some_procedure (void);
```

## Import / Export Aspects (2/2)

- You can also import/export variables
  - Variables imported won't be initialized
  - Ada view

```
My_Var : Integer_Type with
  Import,
  Convention      => C,
  External_Name  => "my_var";
Pragma Import (C, My_Var, "my_var");
```

- C implementation

```
int my_var;
```

## Import / Export with Pragmas

- You can also use **pragma** to import/export entities

```
procedure C_Some_Procedure;
```

```
pragma Import (C, C_Some_Procedure, "SomeProcedure");
```

```
procedure Some_Procedure;
```

```
pragma Export (C, Some_Procedure, "ada_some_procedure");
```

## Parameter Passing

## Parameter Passing to/from C

- The mechanism used to pass formal subprogram parameters and function results depends on:
  - The type of the parameter
  - The mode of the parameter
  - The Convention applied on the Ada side of the subprogram declaration
- The exact meaning of *Convention C*, for example, is documented in *LRM* B.1 - B.3, and in the *GNAT User's Guide* section 3.11.

## Passing Scalar Data As Parameters

- C types are defined by the Standard
- Ada types are implementation-defined
- GNAT standard types are compatible with C types
  - Implementation choice, use carefully
- At the interface level, scalar types must be either constrained with representation clauses, or coming from Interfaces.C
- Ada view

```
with Interfaces.C;  
function C_Proc (I : Interfaces.C.Int)  
    return Interfaces.C.Int;  
pragma Import (C, C_Proc, "c_proc");
```

- C view

```
int c_proc (int i) {  
    /* some code */  
}
```

## Passing Structures As Parameters

- An Ada record that is mapping on a C struct must:
  - Be marked as convention C to enforce a C-like memory layout
  - Contain only C-compatible types

- C View

```
enum Enum {E1, E2, E3};  
struct Rec {  
    int A, B;  
    Enum C;  
};
```

- Ada View

- This can also be done with pragmas

```
type Enum is (E1, E2, E3);  
Pragma Convention (C, Enum);  
type Rec is record  
    A, B : int;  
    C : Enum;  
end record;  
Pragma Convention (C, Rec);
```

# Parameter Modes

- **in** scalar parameters passed by copy
- **out** and **in out** scalars passed using temporary pointer on C side
- By default, composite types passed by reference on all modes except when the type is marked `C_Pass_By_Copy`
  - Be very careful with records - some C ABI pass small structures by copy!
- Ada View

```
Type R1 is record
  V : int;
end record
with Convention => C;
```

```
type R2 is record
  V : int;
end record
with Convention => C_Pass_By_Copy;
```

- C View

```
struct R1{
  int V;
};
struct R2 {
  int V;
};
void f1 (R1 p);
void f2 (R2 p);
```

## Complex Data Types

# Unions

- C `union`

```
union Rec {  
    int A;  
    float B;  
};
```

- C unions can be bound using the `Unchecked_Union` aspect
- These types must have a mutable discriminant for convention purpose, which doesn't exist at run-time
  - All checks based on its value are removed - safety loss
  - It cannot be manually accessed
- Ada implementation of a C `union`

```
type Rec (Flag : Boolean := False) is  
record  
    case Flag is  
        when True =>  
            A : int;  
        when False =>  
            B : float;  
    end case;  
end record  
with Unchecked_Union,  
Convention => C;
```

# Arrays Interfacing

- In Ada, arrays are of two kinds:
  - Constrained arrays
  - Unconstrained arrays
- Unconstrained arrays are associated with
  - Components
  - Bounds
- In C, an array is just a memory location pointing (hopefully) to a structured memory location
  - C does not have the notion of unconstrained arrays
- Bounds must be managed manually
  - By convention (null at the end of string)
  - By storing them on the side
- Only Ada constrained arrays can be interfaced with C

## Arrays From Ada to C

- An Ada array is a composite data structure containing 2 parts: Bounds and Components
  - **Fat pointers**
- When arrays can be sent from Ada to C, C will only receive an access to the components of the array
- Ada View

```
type Arr is array (Integer range <>) of int;  
procedure P (V : Arr; Size : int);  
pragma Import (C, P, "p");
```

- C View

```
void p (int * v, int size) {  
}
```

# Arrays From C to Ada

- There are no boundaries to C types, the only Ada arrays that can be bound must have static bounds
- Additional information will probably need to be passed
- Ada View

```
-- DO NOT DECLARE OBJECTS OF THIS TYPE
type Arr is array (0 .. Integer'Last) of int;
```

```
procedure P (V : Arr; Size : int);
pragma Export (C, P, "p");
```

```
procedure P (V : Arr; Size : int) is
begin
  for J in 0 .. Size - 1 loop
    -- code;
  end loop;
end P;
```

- C View

```
extern void p (int * v, int size);
int x [100];
p (x, 100);
```

# Strings

- Importing a `String` from C is like importing an array - has to be done through a constrained array
- `Interfaces.C.Strings` gives a standard way of doing that
- Unfortunately, C strings have to end by a null character
- Exporting an Ada string to C needs a copy!

```
Ada_Str : String := "Hello World";  
C_Str : chars_ptr := New_String (Ada_Str);
```

- Alternatively, a knowledgeable Ada programmer can manually create Ada strings with correct ending and manage them directly

```
Ada_Str : String := "Hello World" & ASCII.NUL;
```

- Back to the unsafe world - it really has to be worth it speed-wise!

## Interfaces.C

## Interfaces.C Hierarchy

- Ada supplies a subsystem to deal with Ada/C interactions
- `Interfaces.C` - contains typical C types and constants, plus some simple Ada string to/from C character array conversion routines
  - `Interfaces.C.Extensions` - some additional C/C++ types
  - `Interfaces.C.Pointers` - generic package to simulate C pointers (pointer as an unconstrained array, pointer arithmetic, etc)
  - `Interfaces.C.Strings` - types / functions to deal with C "char \*"

## Interfaces.C

```

package Interfaces.C is

  -- Declaration's based on C's <limits.h>
  CHAR_BIT  : constant := 8;
  SCHAR_MIN : constant := -128;
  SCHAR_MAX : constant := 127;
  UCHAR_MAX : constant := 255;

  type int    is new Integer;
  type short  is new Short_Integer;
  type long   is range -(2 ** (System.Parameters.long_bits - Integer'(1))) ..
    .. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;

  type signed_char is range SCHAR_MIN .. SCHAR_MAX;
  for signed_char'Size use CHAR_BIT;

  type unsigned      is mod 2 ** int'Size;
  type unsigned_short is mod 2 ** short'Size;
  type unsigned_long  is mod 2 ** long'Size;

  type unsigned_char is mod (UCHAR_MAX + 1);
  for unsigned_char'Size use CHAR_BIT;

  type ptrdiff_t is range -(2 ** (System.Parameters.ptr_bits - Integer'(1))) ..
    .. +(2 ** (System.Parameters.ptr_bits - Integer'(1))) - 1;

  type size_t is mod 2 ** System.Parameters.ptr_bits;

  -- Floating-Point
  type C_float  is new Float;
  type double   is new Standard.Long_Float;
  type long_double is new Standard.Long_Long_Float;

  type char is new Character;
  nul : constant char := char'First;

  function To_C (Item : Character) return char;
  function To_Ada (Item : char) return Character;

  type char_array is array (size_t range <>) of aliased char;
  for char_array'Component_Size use CHAR_BIT;

  function Is_Nul_Terminated (Item : char_array) return Boolean;

  -- (more not specified here)

end Interfaces.C;

```

# Interfaces.C.Extensions

```
package Interfaces.C.Extensions is

  -- Definitions for C "void" and "void *" types
  subtype void      is System.Address;
  subtype void_ptr is System.Address;

  -- Definitions for C incomplete/unknown structs
  subtype opaque_structure_def is System.Address;
  type opaque_structure_def_ptr is access opaque_structure_def;

  -- Definitions for C++ incomplete/unknown classes
  subtype incomplete_class_def is System.Address;
  type incomplete_class_def_ptr is access incomplete_class_def;

  -- C bool
  type bool is new Boolean;
  pragma Convention (C, bool);

  -- 64-bit integer types
  subtype long_long is Long_Long_Integer;
  type unsigned_long_long is mod 2 ** 64;

  -- (more not specified here)

end Interfaces.C.Extensions;
```

## Interfaces.C.Pointers

```
generic
  type Index is (<>);
  type Component is private;
  type Component_Array is array (Index range <>) of aliased Component;
  Default_Terminator : Component;

package Interfaces.C.Pointers is

  type Pointer is access all Component;
  for Pointer'Size use System.Parameters.ptr_bits;

  function Value (Ref          : Pointer;
                 Terminator   : Component := Default_Terminator)
    return Component_Array;

  function Value (Ref      : Pointer;
                 Length   : ptrdiff_t)
    return Component_Array;

  Pointer_Error : exception;

  function "+" (Left : Pointer;   Right : ptrdiff_t) return Pointer;
  function "+" (Left : ptrdiff_t; Right : Pointer)   return Pointer;
  function "-" (Left : Pointer;   Right : ptrdiff_t) return Pointer;
  function "-" (Left : Pointer;   Right : Pointer)   return ptrdiff_t;

  procedure Increment (Ref : in out Pointer);
  procedure Decrement (Ref : in out Pointer);

  -- (more not specified here)

end Interfaces.C.Pointers;
```

# Interfaces.C.Strings

```
package Interfaces.C.Strings is

  type char_array_access is access all char_array;
  for char_array_access'Size use System.Parameters.ptr_bits;

  type chars_ptr is private;

  type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;

  Null_Ptr : constant chars_ptr;

  function To_Chars_Ptr (Item      : char_array_access;
                       Nul_Check : Boolean := False) return chars_ptr;

  function New_Char_Array (Chars : char_array) return chars_ptr;

  function New_String (Str : String) return chars_ptr;

  procedure Free (Item : in out chars_ptr);

  function Value (Item : chars_ptr) return char_array;
  function Value (Item  : chars_ptr;
                 Length : size_t)
    return char_array;
  function Value (Item : chars_ptr) return String;
  function Value (Item  : chars_ptr;
                 Length : size_t)
    return String;

  function Strlen (Item : chars_ptr) return size_t;

  -- (more not specified here)

end Interfaces.C.Strings;
```

Lab

# Interfacing with C Lab

## ■ Requirements

- Given a C function that calculates speed in MPH from some information, your application should
  - Ask user for distance and time
  - Populate the structure appropriately
  - Call C function to return speed
  - Print speed to console

## ■ Hints

- Structure contains the following components
  - Distance (floating point)
  - Distance Type (enumeral)
  - Seconds (floating point)

## Interfacing with C Lab - GNAT Studio

To compile/link the C file into the Ada executable:

- 1 Make sure the C file is in the same directory as the Ada source files
- 2 **Edit** → **Project Properties**
- 3 **Sources** → **Languages** → Check the "C" box
- 4 Build and execute as normal

# Interfacing with C Lab Solution - Ada

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Interfaces.C;
3 procedure Main is
4
5     package Float_IO is new Ada.Text_IO.Float_IO (Interfaces.C.C_float);
6
7     One_Minute_In_Seconds : constant := 60.0;
8     One_Hour_In_Seconds   : constant := 60.0 * One_Minute_In_Seconds;
9
10    type Distance_T is (Feet, Meters, Miles) with Convention => C;
11    type Data_T is record
12        Distance       : Interfaces.C.C_float;
13        Distance_Type  : Distance_T;
14        Seconds        : Interfaces.C.C_float;
15    end record with Convention => C;
16    function C_Miles_Per_Hour (Data : Data_T) return Interfaces.C.C_float
17        with Import, Convention => C, External_Name => "miles_per_hour";
18
19    Object_Feet : constant Data_T :=
20        (Distance => 6_000.0,
21         Distance_Type => Feet,
22         Seconds => One_Minute_In_Seconds);
23    Object_Meters : constant Data_T :=
24        (Distance => 3_000.0,
25         Distance_Type => Meters,
26         Seconds => One_Hour_In_Seconds);
27    Object_Miles : constant Data_T :=
28        (Distance => 1.0,
29         Distance_Type =>
30             Miles, Seconds => 1.0);
31
32    procedure Run (Object : Data_T) is
33    begin
34        Float_IO.Put (Object.Distance);
35        Put (" " & Distance_T'Image (Object.Distance_Type) & " in ");
36        Float_IO.Put (Object.Seconds);
37        Put (" seconds = ");
38        Float_IO.Put (C_Miles_Per_Hour (Object));
39        Put_Line (" mph");
40    end Run;
41
42    begin
43        Run (Object_Feet);
44        Run (Object_Meters);
45        Run (Object_Miles);
46    end Main;
```

# Interfacing with C Lab Solution - C

```
enum DistanceT { FEET, METERS, MILES };
struct DataT {
    float distance;
    enum DistanceT distanceType;
    float seconds;
};

float miles_per_hour (struct DataT data) {
    float miles = data.distance;
    switch (data.distanceType) {
        case METERS:
            miles = data.distance / 1609.344;
            break;
        case FEET:
            miles = data.distance / 5280.0;
            break;
    };
    return miles / (data.seconds / (60.0 * 60.0));
}
```

## Summary

# Summary

- Possible to interface with other languages (typically C)
- Ada provides some built-in support to make interfacing simpler
- Crossing languages can be made safer
  - But it still increases complexity of design / implementation

# Subprogram Contracts

# Introduction

## Design-By-Contract

- Source code acting in roles of **client** and **supplier** under a binding **contract**
  - **Contract** specifies *requirements* or *guarantees*
    - "A specification of a software element that affects its use by potential clients." (Bertrand Meyer)
  - **Supplier** provides services
    - Guarantees specific functional behavior
    - Has requirements for guarantees to hold
  - **Client** utilizes services
    - Guarantees supplier's conditions are met
    - Requires result to follow the subprogram's guarantees

# Ada Contracts

- Ada contracts include enforcement
  - At compile-time: specific constructs, features, and rules
  - At run-time: language-defined and user-defined exceptions
- Facilities as part of the language definition
  - Range specifications
  - Parameter modes
  - Generic contracts
  - OOP **interface** types
  - Work well, but on a restricted set of use-cases
- Contract aspects to be more expressive
  - Carried by subprograms
  - ... or by types (seen later)
  - Can have **arbitrary** conditions, more **versatile**

# Assertion

- Boolean expression expected to be True
- Said *to hold* when True
- Language-defined **pragma**

```
pragma Assert (not Full (Stack));  
-- stack is not full  
pragma Assert (Stack_Length = 0,  
              Message => "stack was not empty");  
-- stack is empty
```

- Raises language-defined `Assertion_Error` exception if expression does not hold
- The `Ada.Assertions.Assert` subprogram wraps it

```
package Ada.Assertions is  
  Assertion_Error : exception;  
  procedure Assert (Check : in Boolean);  
  procedure Assert (Check : in Boolean; Message : in String);  
end Ada.Assertions;
```

# Quiz

Which of the following statements is (are) correct?

- A. Contract principles apply only to newer versions of the language
- B. Contract should hold even for unique conditions and corner cases
- C. Contract principles were first implemented in Ada
- D. You cannot be both supplier and client

# Quiz

Which of the following statements is (are) correct?

- A. Contract principles apply only to newer versions of the language
- B. *Contract should hold even for unique conditions and corner cases*
- C. Contract principles were first implemented in Ada
- D. You cannot be both supplier and client

## Explanations

- A. No, but design-by-contract **aspects** were fully integrated into Ada 2012
- B. Yes, special case should be included in the contract
- C. No, in eiffel, in 1986!
- D. No, in fact you are always **both**, even the `Main` has a caller!

# Quiz

Which of the following statements is (are) correct?

- A. Assertions can be used in declarations
- B. Assertions can be used in expressions
- C. Any corrective action should happen before contract checks
- D. Assertions must be checked using `pragma Assert`

# Quiz

Which of the following statements is (are) correct?

- A. **Assertions can be used in declarations**
- B. Assertions can be used in expressions
- C. **Any corrective action should happen before contract checks**
- D. Assertions must be checked using `pragma Assert`

## Explanations

- A. Will be checked at elaboration
- B. No assertion expression, but `raise` expression exists
- C. Exceptions as flow-control adds complexity, prefer a proactive `if` to a (reactive) `exception` handler
- D. You can call `Ada.Assertions.Assert`, or even directly `raise Assertion_Error`

# Quiz

Which of the following statements is (are) correct?

- A. Defensive coding is a good practice
- B. Contracts can replace all defensive code
- C. Contracts are executable constructs
- D. Having exhaustive contracts will prevent run-time errors

# Quiz

Which of the following statements is (are) correct?

- A. *Defensive coding is a good practice*
- B. Contracts can replace all defensive code
- C. Contracts are executable constructs
- D. Having exhaustive contracts will prevent run-time errors

## Explanations

- A. Principles are sane, contracts extend those
- B. See previous slide example
- C. e.g. generic contracts are resolved at compile-time
- D. A failing contract **will cause** a run-time error, only extensive (dynamic / static) analysis of contracted code may provide confidence in the absence of runtime errors (AoRTE)

## Preconditions and Postconditions

# Subprogram-based Assertions

- **Explicit** part of a subprogram's **specification**
  - Unlike defensive code
- *Precondition*
  - Assertion expected to hold **prior to** subprogram call
- *Postcondition*
  - Assertion expected to hold **after** subprogram return
- Requirements and guarantees on both supplier and client
- Syntax uses **aspects**

```
procedure Push (This : in out Stack_T;  
               Value : Content_T)  
with Pre => not Full (This),  
     Post => not Empty (This)  
and Top (This) = Value;
```

# Requirements / Guarantees: Quiz

- Given the following piece of code

```

procedure Start is
begin
    ...
    Turn_On;
    ...

procedure Turn_On
  with Pre => Has_Power,
        Post => Is_On;
  
```

- Complete the table in terms of requirements and guarantees

	Client (Start)	Supplier (Turn_On)
Pre (Has_Power)		
Post (Is_On)		

# Requirements / Guarantees: Quiz

- Given the following piece of code

```

procedure Start is
begin
    ...
    Turn_On;
    ...

procedure Turn_On
with Pre => Has_Power,
      Post => Is_On;
  
```

- Complete the table in terms of requirements and guarantees

	Client (Start)	Supplier (Turn_On)
Pre (Has_Power)	Requirement	Guarantee
Post (Is_On)	Guarantee	Requirement

# Defensive Programming

- Should be replaced by subprogram contracts when possible

```
procedure Push (The_Stack : Stack) is
  Entry_Length : constant Positive := Length (The_Stack);
begin
  pragma Assert (not Is_Full (The_Stack)); -- entry condition
  [...]
  pragma Assert (Length (The_Stack) = Entry_Length + 1); -- exit condition
end Push;
```

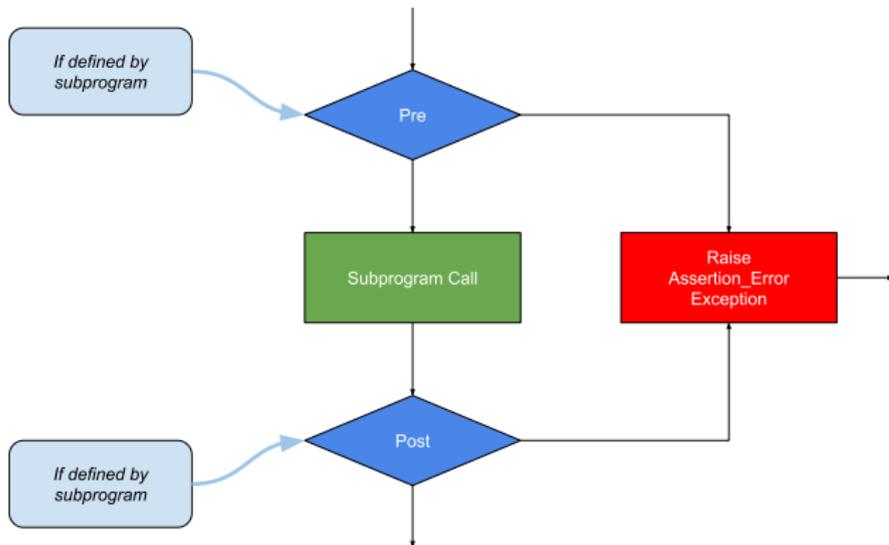
- Subprogram contracts are an **assertion** mechanism
  - **Not** a drop-in replacement for all defensive code

```
procedure Force_Acquire (Resource : Peripheral) is
begin
  if not Available (Resource) then
    -- Corrective action
    Force_Release (Resource);
    pragma Assert (Available (Resource));
  end if;

  Acquire (Resource);
end;
```

# Pre/Postcondition Semantics

- Calls inserted automatically by compiler



## Contract with Quantified Expression

- Pre- and post-conditions can be **arbitrary Boolean** expressions

```
type Status_Flag is (Power, Locked, Running);
```

```
procedure Clear_All_Status (  
    Unit : in out Controller)  
    -- guarantees no flags remain set after call  
with Post => (for all Flag in Status_Flag =>  
    not Status_Indicated (Unit, Flag));
```

```
function Status_Indicated (  
    Unit : Controller;  
    Flag : Status_Flag)  
return Boolean;
```

# Visibility for Subprogram Contracts

- **Any** visible name
  - All of the subprogram's **parameters**
  - Can refer to functions **not yet specified**
    - Must be declared in same scope
    - Different elaboration rules for expression functions

```
function Top (This : Stack) return Content  
  with Pre => not Empty (This);  
function Empty (This : Stack) return Boolean;
```

- Post has access to special attributes
  - See later

## Preconditions and Postconditions Example

- Multiple aspects separated by commas

```
procedure Push (This : in out Stack;  
               Value : Content)  
with Pre  => not Full (This),  
     Post => not Empty (This) and Top (This) = Value;
```

## (Sub)Types Allow Simpler Contracts

### ■ Pre-condition

```
procedure Compute_Square_Root (Input : Integer;  
                               Result : out Natural)  
  with Pre => Input >= 0,  
       Post => (Result * Result) <= Input and  
              (Result + 1) * (Result + 1) > Input;
```

### ■ Subtype

```
procedure Compute_Square_Root (Input  : Natural;  
                               Result : out Natural)  
  with  
    -- "Pre => Input >= 0" not needed  
    -- (Input can't be < 0)  
    Post => (Result * Result) <= Input and  
           (Result + 1) * (Result + 1) > Input;
```

# Quiz

```
-- Convert string to Integer
function To_Integer ( S : String ) return Integer
  with Pre => S'Length > 0;

procedure Print_Something is
  I : Integer := To_Integer ("");
begin
  Put_Line (I'Image);
end Print_Something;
```

Assuming `To_Integer` is defined somewhere, what happens when `Print_Something` is run?

- A. "0" is printed
- B. Constraint Error exception
- C. Assertion Error exception
- D. Undefined behavior

# Quiz

```
-- Convert string to Integer
function To_Integer ( S : String ) return Integer
  with Pre => S'Length > 0;

procedure Print_Something is
  I : Integer := To_Integer ("");
begin
  Put_Line (I'Image);
end Print_Something;
```

Assuming `To_Integer` is defined somewhere, what happens when `Print_Something` is run?

- A. "0" is printed
- B. Constraint Error exception
- C. **Assertion Error exception**
- D. Undefined behavior

## Explanations

The call to `To_Integer` will fail its precondition, which is considered an `Assertion_Error` exception.

# Quiz

```
function Area (Length : Positive; Height : Positive) return Positive is
  (Length * Height)
with Pre => ?
```

Which pre-condition is necessary for Area to calculate the correct result for all values L and H

- A. Length > 0 and Height > 0
- B. Length < Positive'Last and Height < Positive'Last
- C. Length \* Height in Positive
- D. None of the above

# Quiz

```
function Area (Length : Positive; Height : Positive) return Positive is
  (Length * Height)
with Pre => ?
```

Which pre-condition is necessary for Area to calculate the correct result for all values L and H

- A. Length > 0 and Height > 0
- B. Length < Positive'Last and Height < Positive'Last
- C. Length \* Height in Positive
- D. **None of the above**

## Explanations

- A. Parameters are Positive, so this is unnecessary
- B.

Length = Positive'Last-1 and Height = Positive'Last-1  
will still cause an overflow

- C. Classic trap: the check itself may cause an overflow!

Preventing an overflow requires using the expression  
Integer'Last / Length <= Height

## Special Attributes

# Evaluate an Expression on Subprogram Entry

- Post-conditions may require knowledge of a subprogram's **entry context**

```
procedure Increment (This : in out Integer)
  with Post => ??? -- how to assert incrementation of `This`?
```

- Language-defined attribute 'Old
- Expression is **evaluated** at subprogram entry
  - After pre-conditions check
  - Makes a copy
    - **limited** types are forbidden
    - May be expensive
  - Expression can be **arbitrary**
    - Typically **in out** parameters and globals

```
procedure Increment (This : in out Integer) with
  Pre  => This < Integer'Last,
  Post => This = This'Old + 1;
```

## Example for Attribute 'Old

```

Global : String := Init_Global;
...
-- In Global, move character at Index to the left one position,
-- and then increment the Index
procedure Shift_And_Advance (Index : in out Integer) is
begin
  Global (Index) := Global (Index + 1);
  Index          := Index + 1;
end Shift_And_Advance;

```

- Note the different uses of 'Old in the postcondition

```

procedure Shift_And_Advance (Index : in out Integer) with Post =>
  -- Global (Index) before call (so Global and Index are original)
  Global (Index)'Old
  -- Original Global and Original Index
  = Global'Old (Index'Old)
and
  -- Global after call and Index before call
  Global (Index)'Old
  -- Global and Index after call
  = Global (Index);

```

## Error on Conditional Evaluation of 'Old

- This code is **incorrect**

```
procedure Clear_Character (In_String : in out String;  
                          At_Position : Positive)  
with Post => (if At_Position in In_String'Range  
             then In_String (At_Position)'Old = ' ');
```

- Copies In\_String (At\_Position) on entry
  - Will raise an exception on entry if  
At\_Position not in In\_String'Range
  - The postcondition's if check is not sufficient

- Solution requires a full copy of In\_String

```
procedure Clear_Character (In_String : in out String;  
                          At_Position : Positive)  
with Post => (if At_Position in In_String'Range  
             then In_String'Old (At_Position) = ' ');
```

## Postcondition Usage of Function Results

- **function** result can be read with 'Result

```
function Greatest_Common_Denominator (A, B : Positive)
return Positive with
    Post => Is_GCD (A, B,
                    Greatest_Common_Denominator'Result);
```



# Quiz

```

Database : String (1 .. 10) := "ABCDEFGHJIJ";
-- Set the value for the component at position Index in
-- array Database to Value and then increment Index by 1
function Set_And_Move (Value :      Character;
                      Index : in out Index_T)
                      return Boolean

  with Post => ...

```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move ('X', 4)`?

Legend  
 Value on call entry  
 Value on call return

■ Database'Old (Index)

```

Database'Old (Index)
Database before the call: ABCDEFGHJIJ
Index after the call   : 5
Value                  : E

```

■ Database (Index'Old)

■ Database (Index)'Old

# Quiz

```

Database : String (1 .. 10) := "ABCDEFGHJIJ";
-- Set the value for the component at position Index in
-- array Database to Value and then increment Index by 1
function Set_And_Move (Value :      Character;
                      Index : in out Index_T)
                      return Boolean

  with Post => ...

```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move ('X', 4)`?

Legend  
 Value on call entry  
 Value on call return

■ Database'Old (Index)

```

Database'Old (Index)
Database before the call: ABCDEFGHJIJ
Index after the call   : 5
Value                  : E

```

■ Database (Index'Old)

```

Database (Index'Old)
Database after the call : ABCDEFGHJIJ
Index before the call  : 4
Value                  : X

```

■ Database (Index)'Old

## Quiz

```

Database : String (1 .. 10) := "ABCDEFGHJIJ";
-- Set the value for the component at position Index in
-- array Database to Value and then increment Index by 1
function Set_And_Move (Value :      Character;
                      Index : in out Index_T)
                      return Boolean

  with Post => ...

```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move ('X', 4)`?

## Legend

Value on call entry  
Value on call return

## ■ Database'Old (Index)

```

Database'Old (Index)
Database before the call: ABCDEFGHJIJ
Index after the call   : 5
Value                  : E

```

## ■ Database (Index'Old)

```

Database (Index'Old)
Database after the call : ABCDEFGHJIJ
Index before the call  : 4
Value                  : X

```

## ■ Database (Index)'Old

```

Database (Index)'Old
Database before the call: ABCDEFGHJIJ
Index before the call  : 4
Value                  : D

```

# Stack Example (Spec with Contracts)

```

package Stack_Pkg is
  procedure Push (Item : in Integer) with
    Pre => not Full,
    Post => not Empty and then Top = Item;
  procedure Pop (Item : out Integer) with
    Pre => not Empty,
    Post => not Full and Item = Top'Old;
  function Pop return Integer with
    Pre => not Empty,
    Post => not Full and Pop'Result = Top'Old;
  function Top return Integer with
    Pre => not Empty;
  function Empty return Boolean;
  function Full return Boolean;
end Stack_Pkg;

```

```

package body Stack_Pkg is
  Values : array (1 .. 100) of Integer;
  Current : Natural := 0;
  -- Preconditions prevent Push/Pop failure
  procedure Push (Item : in Integer) is
  begin
    Current := Current + 1;
    Values (Current) := Item;
  end Push;
  procedure Pop (Item : out Integer) is
  begin
    Item := Values (Current);
    Current := Current - 1;
  end Pop;
  function Pop return Integer is
    Item : constant Integer := Values (Current);
  begin
    Current := Current - 1;
    return Item;
  end Pop;
  function Top return Integer is
    (Values (Current));
  function Empty return Boolean is
    (Current not in Values'Range);
  function Full return Boolean is
    (Current >= Values'Length);
end Stack_Pkg;

```

## In Practice

# Pre/Postconditions: to Be or Not to Be

- **Preconditions** are reasonable **default** for run-time checks
- **Postconditions** advantages can be **comparatively** low
  - Use of 'Old and 'Result with (maybe deep) copy
  - Very useful in **static analysis** contexts (Hoare triplets)
- For **trusted** library, enabling **preconditions only** makes sense
  - Catch **user's errors**
  - Library is trusted, so Post => True is a reasonable expectation
- Typically contracts are used for **validation**
- Enabling subprogram contracts in production may be a valid trade-off depending on...
  - Exception failure **trace availability** in production
  - Overall **timing constraints** of the final application
  - Consequences of violations **propagation**
  - Time and space **cost** of the contracts
- Typically production settings favor telemetry and off-line analysis

## No Secret Precondition Requirements

- Client should be able to **guarantee** them
- Enforced by the compiler

```
package Some_Package is
  function Foo return Bar
    with Pre => Hidden; -- illegal private reference
private
  function Hidden return Boolean;
end Some_Package;
```

## Postconditions Are Good Documentation

```
procedure Reset
```

```
  (Unit : in out DMA_Controller;
```

```
   Stream : DMA_Stream_Selector)
```

```
with Post =>
```

```
  not Enabled (Unit, Stream) and
```

```
  Operating_Mode (Unit, Stream) = Normal_Mode and
```

```
  Selected_Channel (Unit, Stream) = Channel_0 and
```

```
  not Double_Buffered (Unit, Stream) and
```

```
  Priority (Unit, Stream) = Priority_Low and
```

```
  (for all Interrupt in DMA_Interrupt =>
```

```
    not Interrupt_Enabled (Unit, Stream, Interrupt));
```

## Postcondition Compared to Their Body

- Specifying relevant properties may "repeat" the body
  - Unlike preconditions
  - Typically **simpler** than the body
  - Closer to a **re-phrasing** than a tautology
- Good fit for *hard to solve and easy to check* problems
  - Solvers: `Solve (Find_Root'Result, Equation) = 0`
  - Search: `Can_Exit (Path_To_Exit'Result, Maze)`
  - Cryptography:  
`Match (Signer (Sign_Certificate'Result), Key.Public_Part)`
- Bad fit for poorly-defined or self-defining subprograms

```
function Get_Magic_Number return Integer
with Post => Get_Magic_Number'Result = 42
-- Useless post-condition, simply repeating the body
is (42);
```

## Postcondition Compared to Their Body: Example

```
function Greatest_Common_Denominator (Num1, Num2 : Natural)
  return Integer with
  Post => Is_GCD (Num1,
                  Num2,
                  Greatest_Common_Denominator'Result);
```

```
function Is_GCD (Num1, Num2, Candidate : Integer)
  return Boolean is
  (Num1 rem Candidate = 0 and
   Num2 rem Candidate = 0 and
   (for all K in 1 .. Integer'Min (Num1,Num2) =>
    (if (Num1 rem K = 0 and Num2 rem K = 0)
     then K <= Candidate)))));
```

# Contracts Code Reuse

- Contracts are about **usage** and **behaviour**
  - Not optimization
  - Not implementation details
  - **Abstraction** level is typically high
- Extracting them to **function** is a good idea
  - *Code as documentation, executable specification*
  - Completes the **interface** that the client has access to
  - Allows for **code reuse**

```
procedure Withdraw (This : in out Account;
                   Amount : Currency) with
  Pre => Open (This) and then Funds_Available (This, Amount),
  Post => Balance (This) = Balance (This)'Old - Amount;
...
function Funds_Available (This : Account;
                          Amount : Currency)
  return Boolean is
  (Amount > 0.0 and then Balance (This) >= Amount)
with Pre => Open (This);
```

- A **function** may be unavoidable
  - Referencing private type components

# Subprogram Contracts on Private Types

```
package Bank is
  type Account is private;
  procedure Process_Transaction (This : Account) with
    Pre => This.Balance > 0; -- not legal
    ...
  function Current_Balance (This : Account) return Integer;
    ...
  procedure R (This : Account) with
    Pre => Current_Balance (This) > 0; -- legal
    ...
private
  type Account is record
    Balance : Natural;
    ...
  end record;
  function Current_Balance (This : Account) return Integer is
    (This.Balance);
end Bank;
```

# Preconditions or Explicit Checks?

- Any requirement from the spec should be a pre-condition
  - If clients need to know the body, abstraction is **broken**

- With pre-conditions

```
type Stack (Capacity : Positive) is tagged private;  
procedure Push (This : in out Stack;  
               Value : Content) with  
  Pre => not Full (This);
```

- With defensive code, comments, and return values

```
-- returns True iff push is successful  
function Try_Push (This : in out Stack;  
                 Value : Content) return Boolean  
begin  
  if Full (This) then  
    return False;  
  end if;  
  ...  
end;
```

- But not both
  - For the implementation, preconditions are a **guarantee**
  - A subprogram body should **never** test them

## Raising Specific Exceptions

- In the Exceptions module, we show how user-defined exceptions are better than pre-defined
  - Stack Push raising `Overflow_Error` rather than `Constraint_Error`
- *Default* behavior for a precondition failure is `Assertion_Error`
  - But it doesn't have to be!
- Use *raise expression* in a precondition to get a different exception

```
procedure Push (This : in out Stack;  
               Value : Content) with  
  Pre => not Full (This) or else Overflow_Error;
```

- *Note: Postcondition failure only ever makes sense as an `Assertion_Error`*
  - It's the supplier's fault, not the client's

# Assertion Policy

- Pre/postconditions can be controlled with

```
pragma Assertion_Policy
```

```
pragma Assertion_Policy
```

```
(Pre => Check,
```

```
Post => Ignore);
```

- Fine **granularity** over assertion kinds and policy identifiers

[https://docs.adacore.com/gnat\\_rm-docs/html/gnat\\_rm/gnat\\_rm/implementation\\_defined\\_pragmas.html#pragma-assertion-policy](https://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm/implementation_defined_pragmas.html#pragma-assertion-policy)

- Certain advantage over explicit checks which are **harder** to disable

- Conditional compilation via global **constant** `Boolean`

```
procedure Push (This : in out Stack; Value : Content) is
```

```
begin
```

```
  if Debugging then
```

```
    if Full (This) then
```

```
      raise Overflow;
```

```
    end if;
```

```
  end if;
```

Lab

# Subprogram Contracts Lab

## ■ Overview

### ■ Create a priority-based queue ADT

- Higher priority items come off queue first
- When priorities are same, process entries in order received

## ■ Requirements

### ■ Main program should verify pre-condition failure(s)

- At least one pre-condition should raise something other than assertion error

### ■ Post-condition should ensure queue is correctly ordered

## ■ Hints

### ■ Basically a stack, except insertion doesn't necessarily happen at "top"

### ■ To enable assertions in the runtime from GNAT STUDIO

- **Edit** → **Project Properties**
- **Build** → **Switches** → **Ada**
- Click on *Enable assertions*

## Subprogram Contracts Lab Solution - Queue (Spec)

```

1 package Priority_Queue is
2   Overflow : exception;
3   type Priority_T is (Low, Medium, High);
4   type Queue_T is tagged private;
5   subtype String_T is String (1 .. 20);
6
7   procedure Push (Queue : in out Queue_T;
8                 Priority : Priority_T;
9                 Value : String) with
10    Pre => (not Full (Queue) and then Value'Length > 0) or else raise Overflow,
11    Post => Valid (Queue);
12   procedure Pop (Queue : in out Queue_T;
13                Value : out String_T) with
14    Pre => not Empty (Queue), Post => Valid (Queue);
15
16   function Full (Queue : Queue_T) return Boolean;
17   function Empty (Queue : Queue_T) return Boolean;
18   function Valid (Queue : Queue_T) return Boolean;
19 private
20   Max_Queue_Size : constant := 10;
21   type Entries_T is record
22     Priority : Priority_T;
23     Value : String_T;
24   end record;
25   type Size_T is range 0 .. Max_Queue_Size;
26   type Queue_Array_T is array (1 .. Size_T'Last) of Entries_T;
27   type Queue_T is tagged record
28     Size : Size_T := 0;
29     Entries : Queue_Array_T;
30   end record;
31
32   function Full (Queue : Queue_T) return Boolean is (Queue.Size = Size_T'Last);
33   function Empty (Queue : Queue_T) return Boolean is (Queue.Size = 0);
34
35   function Valid (Queue : Queue_T) return Boolean is
36     (if Queue.Size <= 1 then True
37      else
38       (for all Index in 2 .. Queue.Size =>
39        Queue.Entries (Index).Priority >=
40         Queue.Entries (Index - 1).Priority));
41 end Priority_Queue;

```

## Subprogram Contracts Lab Solution - Queue (Body)

```

1 package body Priority_Queue is
2
3 function Pad (Str : String) return String_T is
4   Retval : String_T := (others => ' ');
5 begin
6   if Str'Length > Retval'Length then
7     Retval := Str (Str'First .. Str'First + Retval'Length - 1);
8   else
9     Retval (1 .. Str'Length) := Str;
10  end if;
11  return Retval;
12 end Pad;
13
14 procedure Push (Queue : in out Queue_T;
15               Priority : Priority_T;
16               Value : String) is
17   Last : Size_T renames Queue.Size;
18   New_Entry : constant Entries_T := (Priority, Pad (Value));
19 begin
20   if Queue.Size = 0 then
21     Queue.Entries (Last + 1) := New_Entry;
22   elsif Priority < Queue.Entries (1).Priority then
23     Queue.Entries (2 .. Last + 1) := Queue.Entries (1 .. Last);
24     Queue.Entries (1) := New_Entry;
25   elsif Priority > Queue.Entries (Last).Priority then
26     Queue.Entries (Last + 1) := New_Entry;
27   else
28     for Index in 1 .. Last loop
29       if Priority <= Queue.Entries (Index).Priority then
30         Queue.Entries (Index + 1 .. Last + 1) :=
31           Queue.Entries (Index .. Last);
32         Queue.Entries (Index) := New_Entry;
33         exit;
34       end if;
35     end loop;
36   end if;
37   Last := Last + 1;
38 end Push;
39
40 procedure Pop (Queue : in out Queue_T;
41               Value : out String_T) is
42 begin
43   Value := Queue.Entries (Queue.Size).Value;
44   Queue.Size := Queue.Size - 1;
45 end Pop;
46
47 end Priority_Queue;

```

# Subprograms Contracts Lab Solution - Main

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Priority_Queue;
3  procedure Main is
4      Queue : Priority_Queue.Queue_T;
5      Value : Priority_Queue.String_T;
6  begin
7
8      Ada.Text_IO.Put_Line ("Normal processing");
9      for Count in 1 .. 3 loop
10         for Priority in Priority_Queue.Priority_T'Range loop
11             Queue.Push (Priority, Priority'Image & Count'Image);
12         end loop;
13     end loop;
14
15     while not Queue.Empty loop
16         Queue.Pop (Value);
17         Put_Line (Value);
18     end loop;
19
20     Ada.Text_IO.Put_Line ("Test overflow");
21     for Count in 1 .. 4 loop
22         for Priority in Priority_Queue.Priority_T'Range loop
23             Queue.Push (Priority, Priority'Image & Count'Image);
24         end loop;
25     end loop;
26
27 end Main;
```

## Summary

# Contract-Based Programming Benefits

- Facilitates building software with reliability built-in
  - Software cannot work well unless "well" is carefully defined
  - Clarifies design by defining obligations/benefits
- Enhances readability and understandability
  - Specification contains explicitly expressed properties of code
- Improves testability but also likelihood of passing!
- Aids in debugging
- Facilitates tool-based analysis
  - Compiler checks conformance to obligations
  - Static analyzers (e.g., SPARK, GNAT Static Analysis Suite) can verify explicit precondition and postconditions

# Summary

- Based on viewing source code as clients and suppliers with enforced obligations and guarantees
- No run-time penalties unless enforced
- OOP introduces the tricky issues
  - Inheritance of preconditions and postconditions, for example
- Note that pre/postconditions can be used on concurrency constructs too

	Clients	Suppliers
Preconditions	Obligation	Guarantee
Postconditions	Guarantee	Obligation

# Type Contracts

# Introduction

# Strong Typing

- We know Ada supports strong typing

```
type Small_Integer_T is range -1_000 .. 1_000;  
type Enumerated_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
type Array_T is array (1 .. 3) of Boolean;
```

- But what if we need stronger enforcement?

- Number must be even
- Subset of non-consecutive enumerals
- Array should always be sorted

## ■ Type Invariant

- Property of type that is always true on external reference
- *Guarantee* to client, similar to subprogram postcondition

## ■ Subtype Predicate

- Add more complicated constraints to a type
- Always enforced, just like other constraints

## Type Invariants

# Type Invariants

- There may be conditions that must hold over entire lifetime of objects
  - Pre/postconditions apply only to subprogram calls

- Sometimes low-level facilities can express it

```
subtype Weekdays is Days range Mon .. Fri;
```

```
-- Guaranteed (absent unchecked conversion)
```

```
Workday : Weekdays := Mon;
```

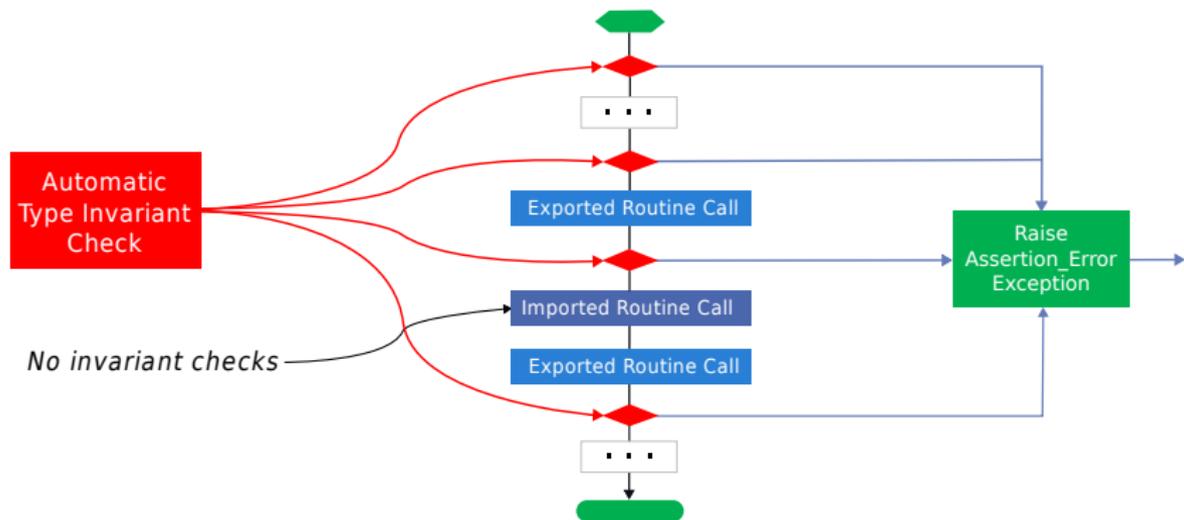
- Type invariants apply across entire lifetime for complex abstract data types
- Part of ADT concept, so only for private types

# Type Invariant Verifications

- Automatically inserted by compiler
- Evaluated as postcondition of creation, evaluation, or return object
  - When objects first created
  - Assignment by clients
  - Type conversions
    - Creates new instances
- Not evaluated on internal state changes
  - Internal routine calls
  - Internal assignments
- Remember - these are abstract data types



## Invariant Over Object Lifetime (Calls)



## Example Type Invariant

- A bank account balance must always be consistent
  - Consistent Balance:  $\text{Total Deposits} - \text{Total Withdrawals} = \text{Balance}$

```
package Bank is
  type Account is private with
    Type_Invariant => Consistent_Balance (Account);
  ...
  -- Called automatically for all Account objects
  function Consistent_Balance (This : Account)
    return Boolean;
  ...
private
  ...
end Bank;
```

# Example Type Invariant Implementation

```
package body Bank is
...
  function Total (This : Transaction_Vector)
    return Currency is
    Result : Currency := 0.0;
  begin
    for Value of This loop
      Result := Result + Value;
    end loop;
    return Result;
  end Total;
  function Consistent_Balance (This : Account)
    return Boolean is
  begin
    return Total (This.Deposits) - Total (This.Withdrawals)
      = This.Current_Balance;
  end Consistent_Balance;
end Bank;
```

## Invariants Don't Apply Internally

- No checking within supplier package
  - Otherwise there would be no way to implement anything!
- Only matters when clients can observe state

```
procedure Open (This : in out Account;  
               Name : in String;  
               Initial_Deposit : in Currency) is  
begin  
  This.Owner := To_Unbounded_String (Name);  
  This.Current_Balance := Initial_Deposit;  
  -- invariant would be false here!  
  This.Withdrawals := Transactions.Empty_Vector;  
  This.Deposits := Transactions.Empty_Vector;  
  This.Deposits.Append (Initial_Deposit);  
  -- invariant is now true  
end Open;
```

## Default Type Initialization for Invariants

- Invariant must hold for initial value
- May need default type initialization to satisfy requirement

```
package Operations is
  -- Type is private, so we can't use Default_Value here
  type Private_T is private with Type_Invariant => Zero (Private_T);
  procedure Op (This : in out Private_T);
  function Zero (This : Private_T) return Boolean;
private
  -- Type is not a record, so we need to use aspect
  -- (A record could use default values for its components)
  type Private_T is new Integer with Default_Value => 0;
  function Zero (This : Private_T) return Boolean is
  begin
    return (This = 0);
  end Zero;
end Operations;
```

## Type Invariant Clause Placement

- Can move aspect clause to private part

```
package Operations is
  type Private_T is private;
  procedure Op (This : in out Private_T);
private
  type Private_T is new Integer with
    Type_Invariant => Private_T = 0,
    Default_Value => 0;
end Operations;
```

- It is really an implementation aspect
  - Client shouldn't care!

## Invariants Are Not Foolproof

- Access to ADT representation via pointer could allow back door manipulation
- These are private types, so access to internals must be granted by the private type's code
- Granting internal representation access for an ADT is a highly questionable design!

# Quiz

```
package Counter_Package is
  type Counter_T is private;
  procedure Increment (Val : in out Counter_T);
private
  function Check_Threshold (Value : Integer)
    return Boolean;
  type Counter_T is new Integer with
    Type_Invariant => Check_Threshold
      (Integer (Counter_T));
end Counter_Package;

package body Counter_Package is
  function Increment_Helper (Helper_Val : Counter_T)
    return Counter_T is
    Next_Value : Counter_T := Helper_Val + 1;
  begin
    return Next_Value;
  end Increment_Helper;
  procedure Increment (Val : in out Counter_T) is
  begin
    Val := Val + 1;
    Val := Increment_Helper (Val);
  end Increment;
  function Check_Threshold (Value : Integer)
    return Boolean is
    (Value <= 100); -- check against constraint
end Counter_Package;
```

If **Increment** is called from outside of **Counter\_Package**, how many times is **Check\_Threshold** called?

- A. 1
- B. 2
- C. 3
- D. 4

# Quiz

```
package Counter_Package is
  type Counter_T is private;
  procedure Increment (Val : in out Counter_T);
private
  function Check_Threshold (Value : Integer)
    return Boolean;
  type Counter_T is new Integer with
    Type_Invariant => Check_Threshold
      (Integer (Counter_T));
end Counter_Package;

package body Counter_Package is
  function Increment_Helper (Helper_Val : Counter_T)
    return Counter_T is
    Next_Value : Counter_T := Helper_Val + 1;
  begin
    return Next_Value;
  end Increment_Helper;
  procedure Increment (Val : in out Counter_T) is
  begin
    Val := Val + 1;
    Val := Increment_Helper (Val);
  end Increment;
  function Check_Threshold (Value : Integer)
    return Boolean is
    (Value <= 100); -- check against constraint
end Counter_Package;
```

If **Increment** is called from outside of **Counter\_Package**, how many times is **Check\_Threshold** called?

- A. 1
- B. 2
- C. 3
- D. 4

Type Invariants are only evaluated on entry into/exit from externally visible subprograms. So **Check\_Threshold** is called when entering/exiting **Increment** - not **Increment\_Helper**

## Subtype Predicates

# Subtype Predicates Concept

- Ada defines support for various kinds of constraints
  - Range constraints
  - Index constraints
  - Others...
- Language defines rules for these constraints
  - All range constraints are contiguous
  - Matter of efficiency
- **Subtype predicates** generalize possibilities
  - Define new kinds of constraints

# Predicates

- Something asserted to be true about some subject
  - When true, said to "hold"
- Expressed as any legal Boolean expression in Ada
  - Quantified and conditional expressions
  - Boolean function calls
- Two forms in Ada
  - **Static Predicates**
    - Specified via aspect named **Static\_Predicate**
  - **Dynamic Predicates**
    - Specified via aspect named **Dynamic\_Predicate**

## Really, type and subtype Predicates

- Applicable to both
- Applied via aspect clauses in both cases
- Syntax

```
type name is type_definition
    with aspect_mark [ => expression ] { ,
        aspect_mark [ => expression ] }
subtype defining_identifier is subtype_indication
    with aspect_mark [ => expression ] { ,
        aspect_mark [ => expression ] }
```

## Why Two Predicate Forms?

	Static	Dynamic
Content	More Restricted	Less Restricted
Placement	Less Restricted	More Restricted

- Static predicates can be used in more contexts
  - More restrictions on content
  - Can be used in places Dynamic Predicates cannot
- Dynamic predicates have more expressive power
  - Fewer restrictions on content
  - Not as widely available

## (Sub)Type Predicate Examples

- Dynamic Predicate

```
subtype Even is Integer with Dynamic_Predicate =>  
    Even mod 2 = 0; -- Boolean expression  
    -- (Even indicates "current instance")
```

- Static Predicate

```
type Serial_Baud_Rate is range 110 .. 115200  
    with Static_Predicate => Serial_Baud_Rate in  
    -- Non-contiguous range  
    110 | 300 | 600 | 1200 | 2400 | 4800 |  
    9600 | 14400 | 19200 | 28800 | 38400 | 56000 |  
    57600 | 115200;
```

# Predicate Checking

- Calls inserted automatically by compiler
- Violations raise exception `Assertion_Error`
  - When predicate does not hold (evaluates to `False`)
- Checks are done before value change
  - Same as language-defined constraint checks
  - Associated variable is unchanged when violation is detected

# Predicate Checks Placement

- Anywhere value assigned that may violate target constraint
- Assignment statements
- Explicit initialization as part of object declaration
- Subtype conversion
- Parameter passing
  - All modes when passed by copy
  - Modes **in out** and **out** when passed by reference
- Implicit default initialization for record components
- On default type initialization values, when taken

# References Are Not Checked

```
with Ada.Text_IO;   use Ada.Text_IO;
procedure Even_Number_Test is
  subtype Even is Integer with Dynamic_Predicate => Even mod 2 = 0;
  Current_Value, Next_Value : Even;
begin
  -- predicates are not checked here
  Put_Line ("Current_Value is" & Current_Value'Image);
  Put_Line ("Next_Value is" & Next_Value'Image);
  -- predicate is checked here
  Current_Value := Next_Value; -- assertion failure here
  Put_Line ("Current_Value is" & Current_Value'Image);
  Put_Line ("Next_Value is" & Next_Value'Image);
end Even_Number_Test;
```

- Output would look like

```
Current_Value is 1969492223
Next_Value is 4220029
```

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE:
Dynamic_Predicate failed at even_number_test.adb:9
```

## Predicate Expression Content

- Reference to value of type itself, i.e., "current instance"

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
Current_Value, Next_Value : Even := 42;
```

- Any visible object or function in scope
  - Does not have to be defined before use
  - Relaxation of "declared before referenced" rule of linear elaboration
  - Intended especially for (expression) functions declared in same package spec

# Static Predicates

- *Static* means known at compile-time, informally
  - Language defines meaning formally (RM 3.2.4)
- Allowed in contexts in which compiler must be able to verify properties
- Content restrictions on predicate are necessary

## Allowed Static Predicate Content (1)

- Ordinary Ada static expressions
- Static membership test selected by current instance
- Example 1

```
type Serial_Baud_Rate is range 110 .. 115200
  with Static_Predicate => Serial_Baud_Rate in
    -- Non-contiguous range
    110   | 300   | 600   | 1200  | 2400  | 4800  | 9600  |
    14400 | 19200 | 28800 | 38400 | 56000 | 57600 | 115200;
```

- Example 2

```
type Days is (Sun, Mon, Tues, We, Thu, Fri, Sat);
  -- only way to create subtype of non-contiguous values
subtype Weekend is Days
  with Static_Predicate => Weekend in Sat | Sun;
```

## Allowed Static Predicate Content (2)

- Case expressions in which dependent expressions are static and selected by current instance

```
type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
subtype Weekend is Days with Static_Predicate =>  
  (case Weekend is  
   when Sat | Sun => True,  
   when Mon .. Fri => False);
```

- Note: if-expressions are disallowed, and not needed

```
subtype Drudge is Days with Static_Predicate =>  
  -- not legal  
  (if Drudge in Mon .. Fri then True else False);  
-- should be  
subtype Drudge is Days with Static_Predicate =>  
  Drudge in Mon .. Fri;
```

## Allowed Static Predicate Content (3)

- A call to `=`, `/=`, `<`, `<=`, `>`, or `>=` where one operand is the current instance (and the other is static)
- Calls to operators `and`, `or`, `xor`, `not`
  - Only for pre-defined type **Boolean**
  - Only with operands of the above
- Short-circuit controls with operands of above
- Any of above in parentheses

## Dynamic Predicate Expression Content

- Any arbitrary Boolean expression
  - Hence all allowed static predicates' content
- Plus additional operators, etc.

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
subtype Vowel is Character with Dynamic_Predicate =>
  (case Vowel is
   when 'A' | 'E' | 'I' | 'O' | 'U' => True,
   when others => False); -- evaluated at run-time
```

- Plus calls to functions
  - User-defined
  - Language-defined

## Types Controlling For-Loops

- Types with dynamic predicates cannot be used

- Too expensive to implement

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
...
-- not legal - how many iterations?
for A_Number in Even loop
  ...
end loop;
```

- Types with static predicates can be used

```
type Days is (Sun, Mon, Tues, We, Thu, Fri, Sat);
subtype Weekend is Days
  with Static_Predicate => Weekend in Sat | Sun;
-- Loop uses "Days", and only enters loop when in Weekend
-- So "Sun" is first value for A_Day
for A_Day in Weekend loop
  ...
end loop;
```

## Why Allow Types with Static Predicates?

- Efficient code can be generated for usage

```
type Days is (Sun, Mon, Tues, We, Thu, Fri, Sat);  
subtype Weekend is Days with Static_Predicate => Weekend in Sat | Sun;  
...  
for A_Day in Weekend loop  
  GNAT.IO.Put_Line (A_Day'Image);  
end loop;
```

- for loop generates code like

```
declare  
  a_day : weekend := sun;  
begin  
  loop  
    gnat_io_put_line__2 (a_day'Image);  
    case a_day is  
      when sun =>  
        a_day := sat;  
      when sat =>  
        exit;  
      when others =>  
        a_day := weekend'succ (a_day);  
    end case;  
  end loop;  
end;
```

## In Some Cases Neither Kind Is Allowed

- No predicates can be used in cases where contiguous layout required
  - Efficient access and representation would be impossible
- Hence no array index or slice specification usage

```
type Play is array (Weekend) of Integer; -- illegal
type Vector is array (Days range <>) of Integer;
Not_Legal : Vector (Weekend); -- not legal
```

## Special Attributes for Predicated Types

- Attributes **'First\_Valid** and **'Last\_Valid**
  - Can be used for any static subtype
  - Especially useful with static predicates
  - **'First\_Valid** returns smallest valid value, taking any range or predicate into account
  - **'Last\_Valid** returns largest valid value, taking any range or predicate into account
- Attributes **'Range**, **'First** and **'Last** are not allowed
  - Reflect non-predicate constraints so not valid
  - **'Range** is just a shorthand for **'First .. 'Last**
- **'Succ** and **'Pred** are allowed since work on underlying type

## Initial Values Can Be Problematic

- Users might not initialize when declaring objects
  - Most predefined types do not define automatic initialization
  - No language guarantee of any specific value (random bits)
  - Example

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
Some_Number : Even;  -- unknown (invalid?) initial value
```

- The predicate is not checked on a declaration when no initial value is given
- So can reference such junk values before assigned
  - This is not illegal (but is a bounded error)

## Subtype Predicates Aren't Bullet-Proof

- For composite types, predicate checks apply to whole object values, not individual components

```
procedure Demo is
  type Table is array (1 .. 5) of Integer
    -- array should always be sorted
  with Dynamic_Predicate =>
    (for all Idx in Table'Range =>
      (Idx = Table'First or else Table (Idx-1) <= Table (Idx)));
  Values : Table := (1, 3, 5, 7, 9);
begin
  ...
  Values (3) := 0; -- does not generate an exception!
  ...
  Values := (1, 3, 0, 7, 9); -- does generate an exception
  ...
end Demo;
```

## Beware Accidental Recursion in Predicate

- Involves functions because predicates are expressions
- Caused by checks on function arguments
- Infinitely recursive example

```
type Sorted_Table is array (1 .. N) of Integer with
  Dynamic_Predicate => Sorted (Sorted_Table);
-- on call, predicate is checked!
function Sorted (T : Sorted_Table) return Boolean;
```

- Non-recursive example

```
type Sorted_Table is array (1 .. N) of Integer with
  Dynamic_Predicate =>
    (for all Index in Sorted_Table'Range =>
      (Index = Sorted_Table'First
       or else Sorted_Table (Index - 1) <= Sorted_Table (Index)));
```

- Type-based example

```
type Table is array (1 .. N) of Integer;
subtype Sorted_Table is Table with
  Dynamic_Predicate => Sorted (Sorted_Table);
function Sorted (T : Table) return Boolean;
```

## GNAT-Specific Aspect Name *Predicate*

- Conflates two language-defined names
- Takes on kind with widest applicability possible
  - Static if possible, based on predicate expression content
  - Dynamic if cannot be static
- Remember: static predicates allowed anywhere that dynamic predicates allowed
  - But not inverse
- Slight disadvantage: you don't find out if your predicate is not actually static
  - Until you use it where only static predicates are allowed

## Enabling/Disabling Contract Verification

- Corresponds to controlling specific run-time checks
  - Syntax

```
pragma Assertion_Policy (policy_name);  
pragma Assertion_Policy (  
    assertion_name => policy_name  
    {, assertion_name => policy_name});
```

- Vendors may define additional policies (GNAT does)
- Default, without pragma, is implementation-defined
- Vendors almost certainly offer compiler switch
  - GNAT uses same switch as for pragma Assert: `-gnata`

# Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
function Is_Weekday (Day : Days_T) return Boolean is  
  (Day /= Sun and then Day /= Sat);
```

Which of the following is a valid subtype predicate?

- A** subtype Sub\_Day is Days\_T with  
 Static\_Predicate => Sub\_Day in Sun | Sat;
- B** subtype Sub\_Day is Days\_T with Static\_Predicate =>  
 (if Sub\_Day = Sun or else Sub\_Day = Sat then True  
 else False);
- C** subtype Sub\_Day is Days\_T with  
 Static\_Predicate => not Is\_Weekday (Sub\_Day);
- D** subtype Sub\_Day is Days\_T with  
 Static\_Predicate =>  
 case Sub\_Day is when Sat | Sun => True,  
 when others => False;

# Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
function Is_Weekday (Day : Days_T) return Boolean is  
  (Day /= Sun and then Day /= Sat);
```

Which of the following is a valid subtype predicate?

- A** `subtype Sub_Day is Days_T with  
 Static_Predicate => Sub_Day in Sun | Sat;`
- B** `subtype Sub_Day is Days_T with Static_Predicate =>  
 (if Sub_Day = Sun or else Sub_Day = Sat then True  
 else False);`
- C** `subtype Sub_Day is Days_T with  
 Static_Predicate => not Is_Weekday (Sub_Day);`
- D** `subtype Sub_Day is Days_T with  
 Static_Predicate =>  
 case Sub_Day is when Sat | Sun => True,  
 when others => False;`

Explanations

- A** Correct
- B** `If` statement not allowed in a predicate
- C** Function call not allowed in `Static_Predicate` (this would be OK for `Dynamic_Predicate`)
- D** Missing parentheses around `case` expression

Lab

# Type Contracts Lab

## ■ Overview

- Create simplistic class scheduling system
  - Client will specify name, day of week, start time, end time
  - Supplier will add class to schedule
  - Supplier must also be able to print schedule

## ■ Requirements

- Monday, Wednesday, and/or Friday classes can only be 1 hour long
- Tuesday and/or Thursday classes can only be 1.5 hours long
- Classes without a set day meet for any non-negative length of time

## ■ Hints

- *Subtype Predicate* to create subtypes of day of week
- *Type Invariant* to ensure that every class meets for correct length of time
- To enable assertions in the runtime from GNAT STUDIO
  - **Edit** → **Project Properties**
  - **Build** → **Switches** → **Ada**
  - Click on *Enable assertions*

# Type Contracts Lab Solution - Schedule (Spec)

```

1 package Schedule is
2   Maximum_Classes : constant := 24;
3   subtype Name_T is String (1 .. 10);
4   type Days_T is (Mon, Tue, Wed, Thu, Fri, None);
5   type Time_T is delta 0.5 range 0.0 .. 23.5;
6   type Classes_T is tagged private;
7   procedure Add_Class (Classes : in out Classes_T;
8                       Name      : Name_T;
9                       Day       : Days_T;
10                      Start_Time : Time_T;
11                      End_Time  : Time_T) with
12                      Pre => Count (Classes) < Maximum_Classes;
13   procedure Print (Classes : Classes_T);
14   function Count (Classes : Classes_T) return Natural;
15 private
16   subtype Short_Class_T is Days_T with Static_Predicate => Short_Class_T in Mon | Wed | Fri;
17   subtype Long_Class_T is Days_T with Static_Predicate => Long_Class_T in Tue | Thu;
18   type Class_T is tagged record
19     Name      : Name_T := (others => ' ');
20     Day       : Days_T := None;
21     Start_Time : Time_T := 0.0;
22     End_Time  : Time_T := 0.0;
23   end record;
24   subtype Class_Size_T is Natural range 0 .. Maximum_Classes;
25   subtype Class_Index_T is Class_Size_T range 1 .. Class_Size_T'Last;
26   type Class_Array_T is array (Class_Index_T range <>) of Class_T;
27   type Classes_T is tagged record
28     Size : Class_Size_T := 0;
29     List : Class_Array_T (Class_Index_T);
30   end record with Type_Invariant =>
31     (for all Index in 1 .. Size => Valid_Times (Classes_T.List (Index)));
32
33   function Valid_Times (Class : Class_T) return Boolean is
34     (if Class.Day in Short_Class_T then Class.End_Time - Class.Start_Time = 1.0
35      elsif Class.Day in Long_Class_T then Class.End_Time - Class.Start_Time = 1.5
36      else Class.End_Time >= Class.Start_Time);
37
38   function Count (Classes : Classes_T) return Natural is (Classes.Size);
39 end Schedule;

```

# Type Contracts Lab Solution - Schedule (Body)

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 package body Schedule is
3
4   procedure Add_Class
5     (Classes : in out Classes_T;
6      Name     : Name_T;
7      Day      : Days_T;
8      Start_Time : Time_T;
9      End_Time  : Time_T) is
10  begin
11    Classes.Size := Classes.Size + 1;
12    Classes.List (Classes.Size) :=
13      (Name => Name, Day => Day,
14       Start_Time => Start_Time, End_Time => End_Time);
15  end Add_Class;
16
17  procedure Print (Classes : Classes_T) is
18  begin
19    for Index in 1 .. Classes.Size loop
20      Put_Line
21        (Days_T'Image (Classes.List (Index).Day) & " : " &
22         Classes.List (Index).Name & " (" &
23         Time_T'Image (Classes.List (Index).Start_Time) & " -" &
24         Time_T'Image (Classes.List (Index).End_Time) & " )");
25    end loop;
26  end Print;
27
28 end Schedule;
```

# Type Contracts Lab Solution - Main

```
1 with Ada.Exceptions; use Ada.Exceptions;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with Schedule; use Schedule;
4 procedure Main is
5   Classes : Classes_T;
6 begin
7   Classes.Add_Class (Name => "Calculus ",
8                     Day => Mon,
9                     Start_Time => 10.0,
10                    End_Time => 11.0);
11  Classes.Add_Class (Name => "History ",
12                    Day => Tue,
13                    Start_Time => 11.0,
14                    End_Time => 12.5);
15  Classes.Add_Class (Name => "Biology ",
16                    Day => Wed,
17                    Start_Time => 13.0,
18                    End_Time => 14.0);
19  Classes.Print;
20  begin
21    Classes.Add_Class (Name => "Chemistry ",
22                      Day => Thu,
23                      Start_Time => 13.0,
24                      End_Time => 14.0);
25  exception
26    when The_Err : others =>
27      Put_Line (Exception_Information (The_Err));
28  end;
29 end Main;
```

## Summary

## Working with Type Invariants

- They are not fully foolproof
  - External corruption is possible
  - Requires dubious usage
- Violations are intended to be supplier bugs
  - But not necessarily so, since not always bullet-proof
- However, reasonable designs will be foolproof

# Type Invariants Vs Predicates

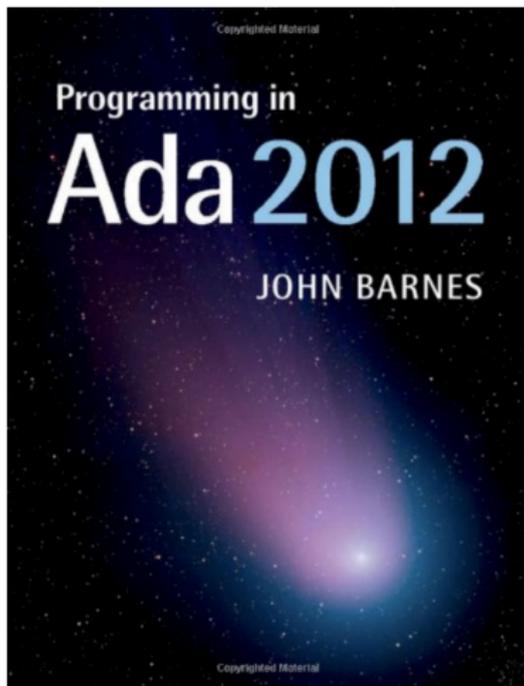
- Type Invariants are valid at external boundary
  - Useful for complex types - type may not be consistent during an operation
- Predicates are like other constraint checks
  - Checked on declaration, assignment, calls, etc

## Annex - Reference Materials

## General Ada Information

# Learning the Ada Language

- Written as a tutorial for those new to Ada



# Reference Manual

- **LRM** - Language Reference Manual (or just **RM**)
  - Always on-line (including all previous versions) at [www.adaic.org](http://www.adaic.org)
- Finding stuff in the RM
  - You will often see the RM cited like this **RM 4.5.3(10)**
  - This means *Section 4.5.3, paragraph 10*
  - Have a look at the table of contents
    - Knowing that chapter 5 is *Statements* is useful
  - Index is very long, but very good!

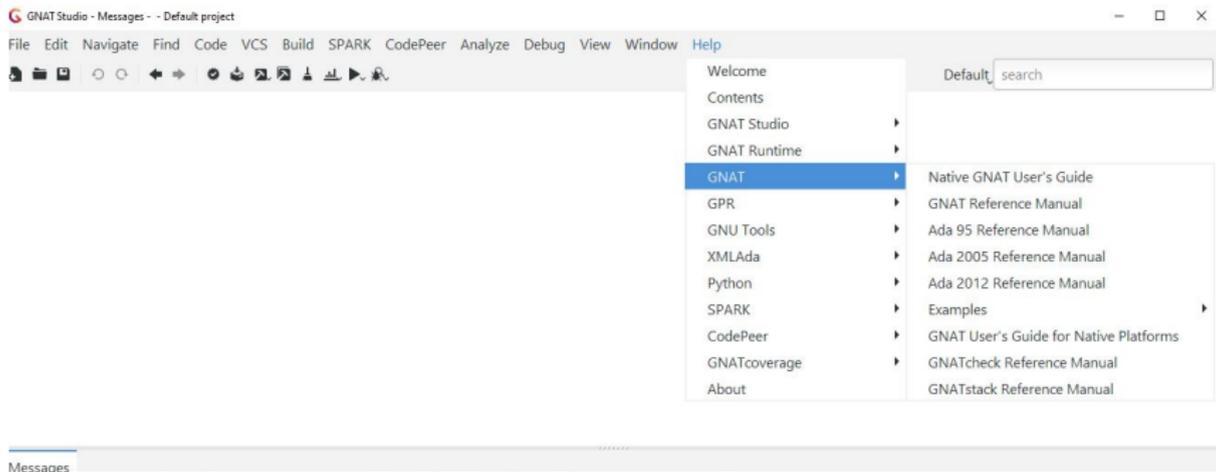
## Current Ada Standard

- "ISO/IEC 8652(E) with Technical Corrigendum 1"
- Useful as a Reference Text but not intended to be read from beginning to end

## GNAT-Specific Help

# Reference Manual

## ■ Reference Manual(s) available from GNAT STUDIO Help



# GNAT Tools

- GNAT User's Guide
  - LOTS of info about the main tools: the GNAT compiler, binder, linker etc.
- GNAT Reference Manual
  - How GNAT implements Ada, pragmas, aspects, attributes etc. etc.
- GNAT STUDIO (the IDE)
  - Tutorial
  - User's Guide
  - Release notes
- Many other tools

## AdaCore Support

## Need More Help?

- If you have an AdaCore subscription:
  - Find out your customer number #XXXX
- Open a "Case" via the GNATtracker web interface and/or email
  - GNATtracker
    - Select "Create A New Case" from the main landing page
  - Email
    - Send to: support@adacore.com
    - Subject should read: #XXXX - (descriptive text)
- Not just for "bug reports"
  - Ask questions, make suggestions, etc.