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
  - Subet 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

Examples

```
package Bank is
   type Account_T is private with Type_Invariant => Consistent_Balance (Account_T);
   type Currency_T is delta 0.01 digits 12;
   function Consistent_Balance (This : Account_T) return Boolean;
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T);
private
   type List_T is array (1 .. 100) of Currency_T;
   type Transaction_List_T is record
      Values : List_T;
      Count  : Natural := 0;
   end record;
   type Account_T is record -- initial state MUST satisfy invariant
      Current_Balance : Currency_T := 0.0;
      Withdrawals     : Transaction_List_T;
      Deposits        : Transaction_List_T;
   end record;
end Bank;

package body Bank is
   function Total (This : Transaction_List_T) return Currency_T is
      Result : Currency_T := 0.0;
   begin
      for I in 1 .. This.Count loop -- no iteration if list empty
         Result := Result + This.Values (I);
      end loop;
      return Result;
   end Total;
   function Consistent_Balance (This : Account_T) return Boolean is
      ( Total (This.Deposits) - Total (This.Withdrawals) = This.Current_Balance );
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T) is
   begin
      This.Current_Balance := Initial_Deposit;
      -- if we checked, the invariant would be false here!
      This.Withdrawals.Count   := 0;
      This.Deposits.Count      := 1;
      This.Deposits.Values (1) := Initial_Deposit;
   end Open; -- invariant is now true
end Bank;
```

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

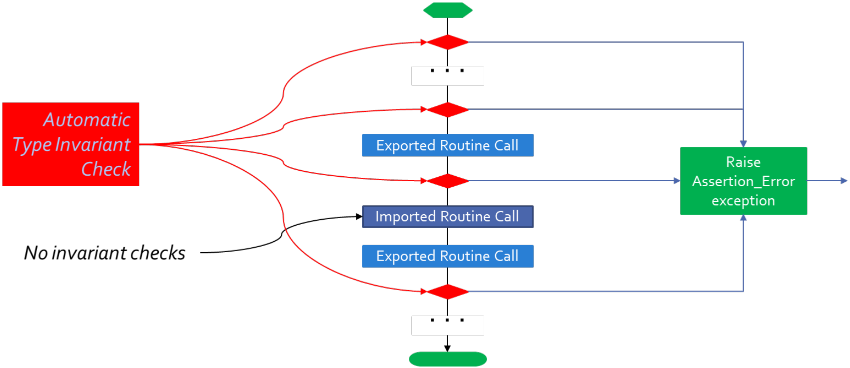
```
subtype Weekdays is Days range Mon .. Fri;

-- 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: Total Deposits - Total Withdrawals = 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_List)
      return Currency is
    Result : Currency := 0.0;
  begin
    for Value of This loop -- no iteration if list empty
      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_List;
  This.Deposits := Transactions.Empty_List;
  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 P is
   -- Type is private, so we can't use Default_Value here
   type T is private with Type_Invariant => Zero (T);
   procedure Op (This : in out T);
   function Zero (This : 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 T is new Integer with Default_Value => 0;
   function Zero (This : T) return Boolean is
   begin
      return (This = 0);
   end Zero;
end P;
```

Type Invariant Clause Placement

- Can move aspect clause to private part

```
package P is
  type T is private;
  procedure Op (This : in out T);
private
  type T is new Integer with
    Type_Invariant => T = 0,
    Default_Value => 0;
end P;
```

- 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 P is
   type Some_T is private;
   procedure Do_Something (X : in out Some_T);
private
   function Counter (I : Integer) return Boolean;
   type Some_T is new Integer with
      Type_Invariant => Counter (Integer (Some_T));
end P;

package body P is
   function Local_Do_Something (X : Some_T)
                                return Some_T is
      Z : Some_T := X + 1;
   begin
      return Z;
   end Local_Do_Something;
   procedure Do_Something (X : in out Some_T) is
   begin
      X := X + 1;
      X := Local_Do_Something (X);
   end Do_Something;
   function Counter (I : Integer)
                     return Boolean is
      ( True );
end P;
```

If **Do_Something** is called from outside of P, how many times is **Counter** called?

A. 1
B. 2
C. 3
D. 4

Quiz

```
package P is
   type Some_T is private;
   procedure Do_Something (X : in out Some_T);
private
   function Counter (I : Integer) return Boolean;
   type Some_T is new Integer with
      Type_Invariant => Counter (Integer (Some_T));
end P;

package body P is
   function Local_Do_Something (X : Some_T)
                                return Some_T is
      Z : Some_T := X + 1;
   begin
      return Z;
   end Local_Do_Something;
   procedure Do_Something (X : in out Some_T) is
   begin
      X := X + 1;
      X := Local_Do_Something (X);
   end Do_Something;
   function Counter (I : Integer)
                     return Boolean is
      ( True );
end P;
```

If **Do_Something** is called from outside of P, how many times is **Counter** called?

A. 1
B. *2*
C. 3
D. 4

Type Invariants are only evaluated on entry into and exit from externally visible subprograms. So `Counter` is called when entering and exiting `Do_Something` - not `Local_Do_Something`, even though a new instance of `Some_T` is created

Subtype Predicates

Examples

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
procedure Predicates is

   subtype Even_T is Integer with Dynamic_Predicate => Even_T mod 2 = 0;
   type Serial_Baud_Rate_T is range 110 .. 115_200 with
      Static_Predicate => Serial_Baud_Rate_T in -- Non-contiguous range
          2_400 | 4_800 | 9_600 | 14_400 | 19_200 | 28_800 | 38_400 | 56_000;

   -- This must be dynamic because "others" will be evaluated at run-time
   subtype Vowel_T is Character with Dynamic_Predicate =>
         (case Vowel_T is when 'A' | 'E' | 'I' | 'O' | 'U' => True, when others => False);

   type Table_T is array (Integer range <>) of Integer;
   subtype Sorted_Table_T is Table_T (1 .. 5) with
        Dynamic_Predicate =>
        (for all K in Sorted_Table_T'Range =>
           (K = Sorted_Table_T'First or else Sorted_Table_T (K - 1) <= Sorted_Table_T (K)));

   J      : Even_T;
   Values : Sorted_Table_T := (1, 3, 5, 7, 9);

begin
   begin
      Put_Line ("J is" & J'Img);
      J := Integer'Succ (J); -- assertion failure here
      Put_Line ("J is" & J'Img);
      J := Integer'Succ (J); -- or maybe here
      Put_Line ("J is" & J'Img);
   exception
      when The_Err : others =>
         Put_Line (Exception_Message (The_Err));
   end;

   for Baud in Serial_Baud_Rate_T loop
      Put_Line (Baud'Image);
   end loop;

   Put_Line (Vowel_T'Image (Vowel_T'Succ ('A')));
   Put_Line (Vowel_T'Image (Vowel_T'Pred ('Z')));

   begin
      Values (3) := 0; -- not an exception
      Values     := (1, 3, 0, 7, 9); -- exception
   exception
      when The_Err : others =>
         Put_Line (Exception_Message (The_Err));
   end;
end 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

Subtype 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 Test is
  subtype Even is Integer with Dynamic_Predicate => Even mod 2 = 0;
  J, K : Even;
begin
  -- predicates are not checked here
  Put_Line ("K is" & K'Img);
  Put_Line ("J is" & J'Img);
  -- predicate is checked here
  K := J; -- assertion failure here
  Put_Line ("K is" & K'Img);
  Put_Line ("J is" & J'Img);
end Test;
```

- Output would look like

```
K is 1969492223
J is 4220029

raised SYSTEM.ASSERTIONS.ASSERT_FAILURE:
Dynamic_Predicate failed at 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;
J, K : 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 K 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 K
for K 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 W in Weekend loop
  GNAT.IO.Put_Line (W'Img);
end loop;
```

- `for` loop generates code like

```
declare
  w : weekend := sun;
begin
  loop
    gnat__io__put_line__2 (w'Img);
    case w is
      when sun =>
        w := sat;
      when sat =>
        exit;
      when others =>
        w := weekend'succ(w);
    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 List is array (Days range <>) of Integer;
L : List (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;
K : 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 K in Table'Range =>
        (K = Table'First or else Table(K-1) <= Table(K)));
  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 K in Sorted_Table'Range =>
      (K = Sorted_Table'First
       or else Sorted_Table (K - 1) <= Sorted_Table (K)));
```

- 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 (D : Days_T) return Boolean is
   (D /= Sun and then D /= Sat);
```

Which of the following is a valid subtype predicate?

A.
```
subtype T is Days_T with
   Static_Predicate => T in Sun | Sat;
```

B.
```
subtype T is Days_T with Static_Predicate =>
   (if T = Sun or else T = Sat then True else False);
```

C.
```
subtype T is Days_T with
   Static_Predicate => not Is_Weekday (T);
```

D.
```
subtype T is Days_T with
   Static_Predicate =>
     case T is when Sat | Sun => True,
            when others => False;
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (D : Days_T) return Boolean is
   (D /= Sun and then D /= Sat);
```

Which of the following is a valid subtype predicate?

A. *`subtype T is Days_T with`*
   `Static_Predicate => T in Sun | Sat;`

B. `subtype T is Days_T with Static_Predicate =>`
   `(if T = Sun or else T = Sat then True else False);`

C. `subtype T is Days_T with`
   `Static_Predicate => not Is_Weekday (T);`

D. `subtype T is Days_T with`
   `Static_Predicate =>`
   `case T 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 run-time from GNAT STUDIO
    - Edit → Project Properties
    - **Build** → **Switches** → **Ada**
    - Click on *Enable assertions*

Type Contracts Lab Solution - Schedule (Spec)

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Schedule is
   Maximum_Classes : constant := 24;
   type Days_T is (Mon, Tue, Wed, Thu, Fri, None);
   type Time_T is delta 0.5 range 0.0 .. 23.5;
   type Classes_T is tagged private;
   procedure Add_Class (Classes    : in out Classes_T;
                        Name       :        String;
                        Day        :        Days_T;
                        Start_Time :        Time_T;
                        End_Time   :        Time_T) with
                        Pre => Count (Classes) < Maximum_Classes;
   procedure Print (Classes : Classes_T);
   function Count (Classes : Classes_T) return Natural;
private
   subtype Short_Class_T is Days_T with Static_Predicate => Short_Class_T in Mon | Wed | Fri;
   subtype Long_Class_T is Days_T with Static_Predicate => Long_Class_T in Tue | Thu;
   type Class_T is tagged record
      Name       : Unbounded_String := Null_Unbounded_String;
      Day        : Days_T           := None;
      Start_Time : Time_T           := 0.0;
      End_Time   : Time_T           := 0.0;
   end record;
   subtype Class_Size_T is Natural range 0 .. Maximum_Classes;
   subtype Class_Index_T is Class_Size_T range 1 .. Class_Size_T'Last;
   type Class_Array_T is array (Class_Index_T range <>) of Class_T;
   type Classes_T is tagged record
      Size : Class_Size_T := 0;
      List : Class_Array_T (Class_Index_T);
   end record with Type_Invariant =>
      (for all Index in 1 .. Size => Valid_Times (Classes_T.List (Index)));

   function Valid_Times (Class : Class_T) return Boolean is
     (if Class.Day in Short_Class_T then Class.End_Time - Class.Start_Time = 1.0
      elsif Class.Day in Long_Class_T then Class.End_Time - Class.Start_Time = 1.5
      else Class.End_Time >= Class.Start_Time);

   function Count (Classes : Classes_T) return Natural is (Classes.Size);
end Schedule;
```

Type Contracts Lab Solution - Schedule (Body)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Schedule is

   procedure Add_Class
     (Classes    : in out Classes_T;
      Name       :        String;
      Day        :        Days_T;
      Start_Time :        Time_T;
      End_Time   :        Time_T) is
   begin
      Classes.Size                := Classes.Size + 1;
      Classes.List (Classes.Size) :=
        (Name       => To_Unbounded_String (Name), Day => Day,
         Start_Time => Start_Time, End_Time => End_Time);
   end Add_Class;

   procedure Print (Classes : Classes_T) is
   begin
      for Index in 1 .. Classes.Size loop
         Put_Line
           (Days_T'Image (Classes.List (Index).Day) & ": " &
            To_String (Classes.List (Index).Name) & " (" &
            Time_T'Image (Classes.List (Index).Start_Time) & " -" &
            Time_T'Image (Classes.List (Index).End_Time) & " )");
      end loop;
   end Print;

end Schedule;
```

Type Contracts Lab Solution - Main

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
with Schedule;       use Schedule;
procedure Main is
   Classes : Classes_T;
begin
   Classes.Add_Class (Name       => "Calculus",
                      Day        => Mon,
                      Start_Time => 10.0,
                      End_Time   => 11.0);
   Classes.Add_Class (Name       => "History",
                      Day        => Tue,
                      Start_Time => 11.0,
                      End_Time   => 12.5);
   Classes.Add_Class (Name       => "Biology",
                      Day        => Wed,
                      Start_Time => 13.0,
                      End_Time   => 14.0);
   Classes.Print;
   begin
      Classes.Add_Class (Name       => "Biology",
                         Day        => Thu,
                         Start_Time => 13.0,
                         End_Time   => 14.0);
   exception
      when The_Err : others =>
         Put_Line (Exception_Information (The_Err));
   end;
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