CC BY SA

SPARK 2014: Object Oriented Programming

University.adacore.com

Copyright © AdaCore

What is Object Oriented Programming?

Object-oriented software construction is
the building of software systems as structured collections
of [...] abstract data type implementations.

Bertrand Meyer, "Object Oriented Software Construction"

Object Oriented Programming **is about:**

- isolating clients from implementation details (abstraction)
- isolating clients from the choice of data types (dynamic dispatching)

Object Oriented Programming **is not:**

- the same as prototype programming (class and objects)
- the same as scoping (class as the scope for methods)
- the same as code reuse (use a component in a record in SPARK)

Copyright © AdaCore

Prototypes and Scopes in SPARK

- **Types in SPARK come with methods aka *primitive operations***

```
type Int is range 1 .. 10;
function Equal (Arg1, Arg2 : Int) return Boolean;
procedure Bump (Arg : in out Int);

type Approx_Int is new Int;
--  implicit definition of Equal and Bump for Approx_Int
```

- **Scope for the prototype is current declarative region**
  - … or up to the first freezing point – point at which the type must be fully defined, e.g. when defining an object of the type

- **OOP without dynamic dispatching = Abstract Data Types**

Copyright © AdaCore

Classes in SPARK

- **Classes in SPARK are *tagged records***

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

function Equal (Arg1, Arg2 : Int) return Boolean;
procedure Bump (Arg : in out Int);

type Approx_Int is new Int with record
   Precision : Natural;
end record;
--  implicit definition of Equal and Bump for Approx_Int
```

- **Derived types are specializations of the root type**
  - typically with more components
  - inheriting the methods on the parent type
  - can add their own methods

Copyright © AdaCore

Methods in SPARK

- **Derived methods can be overriding or not**

```
overriding function Equal (Arg1, Arg2 : Approx_Int)
  return Boolean;
overriding procedure Bump (Arg : in out Approx_Int);

not overriding procedure Blur (Arg : in out Approx_Int);
```

- **Method called depends on static type**

```
I : Int;
Bump (I); -- call to Int.Bump
I.Bump; -- call to Int.Bump (object.method notation)


AI : Approx_Int;
Bump (AI); -- call to Approx_Int.Bump
Bump (Int(AI)); -- call to Int.Bump
```

Copyright © AdaCore

Dynamic dispatching in SPARK

- **Class-wide types**
  - type of object that triggers dispatching
  - method called depends on dynamic type

```
IC : Int'Class := Int'Class(I);
IC.Bump; -- call to Int.Bump

IC : Int'Class := Int'Class(AI);
IC.Bump; -- call to Approx_Int.Bump
```

- **Class-wide views of objects**
  - in Ada, usually manipulated through pointers
  - in SPARK, manipulated through parameter passing

```
procedure Call_Bump (Arg : in out Int'Class);
Call_Bump (Int'Class(I)); -- calls Int.Bump(I)
Call_Bump (Int'Class(AI)); -- calls Approx_Int.Bump(AI)
```

Copyright © AdaCore

Dynamic dispatching – A trivial example

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

procedure Bump (Arg : in out Int);

procedure Call_Bump
  (Arg : in out Int'Class) is
begin
   Arg.Bump;
end Call_Bump;
```

what is called here?

Copyright © AdaCore

The problems with dynamic dispatching

- **Control and data flow are not known statically**
  - control flow – which subprogram is called when dispatching
  - data flow – what data this subprogram is accessing
  - similar to callbacks through subprogram pointers
- **Avionics standard DO-178C lists 3 verification options**
  - run all tests on parent type where derived type is used instead
  - cover all possible methods at dispatching calls
  - prove type substitutability (Liskov Substitution Principle aka LSP)

Copyright © AdaCore

LSP – the SPARK solution to dynamic dispatching problems

- **Class-wide contracts on methods**
  - Pre'Class specifies strongest precondition for the hierarchy
  - Post'Class specifies weakest postcondition for the hierarchy

```
procedure Bump (Arg : in out Int) with
   Pre'Class  => Arg.Value < Arg.Max - 10,
   Post'Class => Arg.Value > Arg.Value'Old;
```

```
procedure Bump (Arg : in out Approx_Int) with
   Pre'Class  => Arg.Value > 100,
   Post'Class => Arg.Value = Arg.Value'Old;
```

```
procedure Bump (Arg : in out Approx_Int) with
   Pre'Class  => True,
   Post'Class => Arg.Value = Arg.Value'Old + 10;
```

```
procedure Bump (Arg : in out Approx_Int);
   -- inherited Pre'Class from Int.Bump
   -- inherited Post'Class from Int.Bump
```

Copyright © AdaCore

LSP – verification of dynamic dispatching calls

- **Class-wide contracts used for dynamic dispatching calls**

```
procedure Call_Bump (Arg : in out Int'Class) with
   Pre  => Arg.Value < Arg.Max - 10,
   Post => Arg.Value > Arg.Value'Old
is
begin
   Arg.Bump;
end Call_Bump;
```

- **LSP applies to data dependencies too**
  - overriding method cannot read more global variables
  - overriding method cannot write more global variables
  - overriding method cannot have new input-output flows
  - SPARK RM defines Global'Class and Depends'Class (not yet implemented → use Global and Depends instead)

Copyright © AdaCore

LSP – class-wide contracts and data abstraction

- **Abstraction can be used in class-wide contracts**

```
type Int is tagged private;

function Get_Value (Arg : Int) return Integer;
function Small (Arg : Int) return Boolean with Ghost;

procedure Bump (Arg : in out Int) with
   Pre'Class  => Arg.Small,
   Post'Class => Arg.Get_Value > Arg.Get_Value'Old;
```

- **Typically use expression functions for abstraction**

```
private
   type Int is tagged record … end record;

   function Get_Value (Arg : Int) return Integer is
     (Arg.Value);
   function Small (Arg : Int) return Boolean is
     (Arg.Value < Arg.Max - 10);
```

Copyright © AdaCore

LSP – class-wide contracts, data abstraction and overriding

- **Abstraction functions can be overridden freely**
  - overriding needs not be weaker or stronger than overridden

```
function Small (Arg : Int) return Boolean is
  (Arg.Value < Arg.Max - 10);

function Small (Arg : Approx_Int) return Boolean is
  (True);

function Small (Arg : Approx_Int) return Boolean is
  (Arg.Value in 1 .. 100);
```

- **Inherited contract reinterpreted for derived class**

```
overriding procedure Bump (Arg : in out Approx_Int);
  --  inherited Pre'Class uses Approx_Int.Small
  --  inherited Post'Class uses Approx_Int.Get_Value
```

Copyright © AdaCore

Dynamic semantics of class-wide contracts

- **Class-wide precondition is the disjunction (or) of**
  - own class-wide precondition, and
  - class-wide preconditions of all overridden methods
- **Class-wide postcondition is the conjunction (and) of**
  - own class-wide postcondition, and
  - class-wide postconditions of all overridden methods
- **Plain Post + class-wide Pre/Post can be used together**
- **Proof guarantees no violation of contracts at runtime**
  - LSP guarantees stronger than dynamic semantics

Copyright © AdaCore

Redispatching and Extensions_Visible aspect

- **Redispatching is dispatching after class-wide conversion**

```
procedure Re_Call_Bump (Arg : in out Int) is
begin
   Int'Class(Arg).Bump;
end Re_Call_Bump;
```

```
formal parameter cannot be converted to class-wide type when
Extensions_Visible is False
```

- **Aspect Extensions_Visible allows class-wide conversion**
  - parameter mode used also for hidden components

```
procedure Re_Call_Bump (Arg : in out Int) with
   Extensions_Visible
is
begin
   Int'Class(Arg).Bump;
end Re_Call_Bump;
```

Copyright © AdaCore

Quiz

Copyright © AdaCore

Is this correct? 1/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Int is record
   Min, Max, Value : Integer;
end record;

procedure Bump (Arg : in out Int) with
   Pre'Class  => Arg.Value < Arg.Max - 10,
   Post'Class => Arg.Value > Arg.Value'Old;
```

Copyright © AdaCore

Is this correct? 1/10

NO

```
type Int is record
   Min, Max, Value : Integer;
end record;

procedure Bump (Arg : in out Int) with
   Pre'Class  => Arg.Value < Arg.Max - 10,
   Post'Class => Arg.Value > Arg.Value'Old;
```

Class-wide contracts are only allowed on tagged records.

Copyright © AdaCore

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

procedure Bump (Arg : in out Int) with
   Pre  => Arg.Value < Arg.Max - 10,
   Post => Arg.Value > Arg.Value'Old;
```

Copyright © AdaCore

Is this correct? 2/10

NO

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

procedure Bump (Arg : in out Int) with
   Pre  => Arg.Value < Arg.Max - 10,
   Post => Arg.Value > Arg.Value'Old;
```

Plain precondition on dispatching subprogram is not allowed in SPARK. Otherwise it would have to be both weaker and stronger than the class-wide precondition (because they are both checked dynamically on both plain calls and dispatching calls).

Plain postcondition is allowed, and should be stronger than class-wide postcondition (plain postcondition used for plain calls).

Copyright © AdaCore

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Bump (Arg : in out Int) with
   Pre'Class  => Arg.Value < Arg.Max - 10,
   Post'Class => Arg.Value > Arg.Value'Old;
```

```
overriding procedure Bump (Arg : in out Approx_Int) with
   Post'Class => Arg.Value = Arg.Value'Old + 10
is
begin
   Arg.Value := Arg.Value + 10;
end Bump;
```

Copyright © AdaCore

Is this correct? 3/10

YES

```
procedure Bump (Arg : in out Int) with
   Pre'Class  => Arg.Value < Arg.Max - 10,
   Post'Class => Arg.Value > Arg.Value'Old;
```

```
overriding procedure Bump (Arg : in out Approx_Int) with
   Post'Class => Arg.Value = Arg.Value'Old + 10
is
begin
   Arg.Value := Arg.Value + 10;
end Bump;
```

Class-wide precondition of Int.Bump is inherited by Approx_Int.Bump. Class-wide postcondition of Approx_Int.Bump is stronger than the one of Int.Bump.

Copyright © AdaCore

Is this correct? 4/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

function "+" (Arg1, Arg2 : Int) return Int with
   Pre'Class => Arg1.Min = Arg2.Min
            and Arg1.Max = Arg2.Max;

type Approx_Int is new Int with record
   Precision : Natural;
end record;

-- inherited function “+”
```

Copyright © AdaCore

Is this correct? 4/10

NO

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

function "+" (Arg1, Arg2 : Int) return Int with
   Pre'Class => Arg1.Min = Arg2.Min
            and Arg1.Max = Arg2.Max;

type Approx_Int is new Int with record
   Precision : Natural;
end record;

-- inherited function "+"
```

type must be declared abstract or "+" overridden

Copyright © AdaCore

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

procedure Reset (Arg : out Int);

type Approx_Int is new Int with record
   Precision : Natural;
end record;

-- inherited procedure Reset
```

Copyright © AdaCore

Is this correct? 5/10

NO

```
type Int is tagged record
   Min, Max, Value : Integer;
end record;

procedure Reset (Arg : out Int);

type Approx_Int is new Int with record
   Precision : Natural;
end record;

-- inherited procedure Reset
```

```
type must be declared abstract or "Reset" overridden
"Reset" is subject to Extensions_Visible False
```

Copyright © AdaCore

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Int is tagged record … end record;

procedure Reset (Arg : out Int) with Extensions_Visible
is
begin
   Arg := Int'(Min   => -100,
               Max   => 100,
               Value => 0);
end Reset;

type Approx_Int is new Int with record … end record;

-- inherited procedure Reset
```

Copyright © AdaCore

Is this correct? 6/10

NO

```
type Int is tagged record … end record;

procedure Reset (Arg : out Int) with Extensions_Visible
is
begin
   Arg := Int'(Min   => -100,
               Max   => 100,
               Value => 0);
end Reset;

type Approx_Int is new Int with record … end record;

-- inherited procedure Reset
```

high: extension of "Arg" is not initialized in "Reset"

Copyright © AdaCore

Is this correct? 7/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Int is tagged record … end record;
function Zero return Int;

procedure Reset (Arg : out Int) with Extensions_Visible
is
begin
   Int'Class(Arg) := Zero;
end Reset;

type Approx_Int is new Int with record … end record;
overriding function Zero return Approx_Int;

-- inherited procedure Reset
```

Copyright © AdaCore

Is this correct? 7/10

```
type Int is tagged record … end record;
function Zero return Int;

procedure Reset (Arg : out Int) with Extensions_Visible
is
begin
   Int'Class(Arg) := Zero;
end Reset;

type Approx_Int is new Int with record … end record;
overriding function Zero return Approx_Int;

-- inherited procedure Reset
```

Redispatching ensures that Arg is fully initialized on return.

Copyright © AdaCore

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type File is tagged private;

procedure Create (F : out File) with
   Post'Class => F.Closed;
procedure Open_Read (F : in out File) with
   Pre'Class  => F.Closed,
   Post'Class => F.Is_Open;
procedure Close (F : in out File) with
   Pre'Class  => F.Is_Open,
   Post'Class => F.Closed;
```

```
procedure Use_File_System (F : out File'Class) is
begin
   F.Create;
   F.Open_Read;
   F.Close;
end Use_File_System;
```

Copyright © AdaCore

Is this correct? 8/10


```
type File is tagged private;

procedure Create (F : out File) with
   Post'Class => F.Closed;
procedure Open_Read (F : in out File) with
   Pre'Class  => F.Closed,
   Post'Class => F.Is_Open;
procedure Close (F : in out File) with
   Pre'Class  => F.Is_Open,
   Post'Class => F.Closed;
```

```
procedure Use_File_System (F : out File'Class) is
begin
   F.Create;
   F.Open_Read;
   F.Close;
end Use_File_System;
```

State automaton encoded in class-wide contracts is respected.

Copyright © AdaCore

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type File is new File_System.File with private;

procedure Create (F : out File) with
   Post'Class => F.Closed;
procedure Open_Read (F : in out File) with
   Pre'Class  => F.Closed,
   Post'Class => F.Is_Open and F.Is_Synchronized;
procedure Close (F : in out File) with
   Pre'Class  => F.Is_Open and F.Is_Synchronized;
   Post'Class => F.Closed;
```

```
procedure Use_File_System (F : out File'Class) is
begin
   F.Create;
   F.Open_Read;
   F.Close;
end Use_File_System;
```

Copyright © AdaCore

Is this correct? 9/10

NO

```
type File is new File_System.File with private;

procedure Create (F : out File) with
   Post'Class => F.Closed;
procedure Open_Read (F : in out File) with
   Pre'Class  => F.Closed,
   Post'Class => F.Is_Open and F.Is_Synchronized;
procedure Close (F : in out File) with
   Pre'Class  => F.Is_Open and F.Is_Synchronized;
   Post'Class => F.Closed;
```

```
procedure Use_File_System (F : out File'Class) is
begin
   F.Create;
   F.Open_Read;
   F.Close;
end Use_File_System;
```

medium: class-wide precondition might be stronger than overridden one

Copyright © AdaCore

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type File is new File_System.File with private;

procedure Create (F : out File) with
   Post'Class => F.Closed;
procedure Open_Read (F : in out File) with
   Pre'Class  => F.Closed,
   Post'Class => F.Is_Open;
procedure Close (F : in out File) with
   Pre'Class  => F.Is_Open;
   Post'Class => F.Closed;
```

```
private
   type File is new File_System.File with record
      In_Synch : Boolean;
   end record with
      Predicate => File_System.File (File).Closed
                   or In_Synch;
```

Predicate encodes the additional constraint on opened files.
Type invariants are not yet supported on tagged types in SPARK.

Copyright © AdaCore

Is this correct? 10/10


```
type File is new File_System.File with private;

procedure Create (F : out File) with
   Post'Class => F.Closed;
procedure Open_Read (F : in out File) with
   Pre'Class  => F.Closed,
   Post'Class => F.Is_Open;
procedure Close (F : in out File) with
   Pre'Class  => F.Is_Open;
   Post'Class => F.Closed;
```

```
private
   type File is new File_System.File with record
      In_Synch : Boolean;
   end record with
      Predicate => File_System.File (File).Closed
                   or In_Synch;
```

Predicate encodes the additional constraint on opened files.
Type invariants are not yet supported on tagged types in SPARK.

Copyright © AdaCore

CC BY SA

university.adacore.com

Copyright © AdaCore