SWEN90010 — High Integrity Systems Engineering

Lecture 12 - Ada — continued

Hira Syeda
hira.syeda@unimelb.edu.au
Level 2, Melbourne Connect

15th April, 2025

What did we learn last time about Ada

Ada has **module system** that provides information hiding, application interface

Ada has **strong, static type system**

Ada comes with several **runtime checks**: signed overflow
we will see going forward: array index out-of-bounds,
accessing unallocated memory, range errors

Ada turns **heisenbugs** into **crashes**

because of these features, **Ada is portable**

Ada prefers **code readability**

Main Takeaway

Ada is designed for writing safe and secure code

Application Areas

Mostly real-time embedded:
that are traditionally very difficult to program,
and Ada gives highly relivable code

Avionics Railway Defence Space

Robotics Crypto(graphy)

Application Areas

Mostly real-time embedded:
that are traditionally very difficult to program,
and Ada gives highly relivable code

Avionics Railway Defence Space

Robotics Crypto(graphy)

Lots of these
with formal verification
via SPARK

Muen verified Separation Kenrel

MISRA C

MISRA C: “safe” C subset used in automotive
(not all rules enforceable)

Ada is already much “safer” than C; better encapsulation

SPARK: an “even safer” subset of Ada, with
same runtime semantics (future lectures)

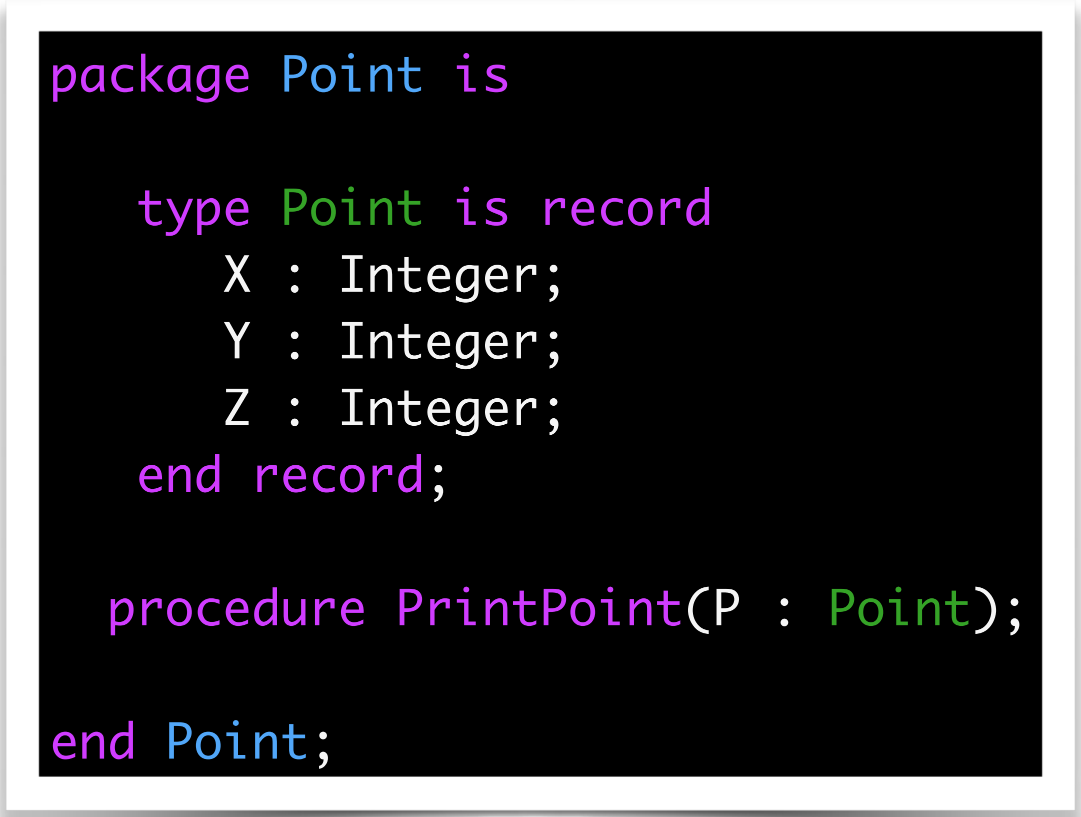
Ada: stronger type system, more static guarantees

Ada: more runtime checks
(e.g. array index out of bounds,
dynamic memory access etc.)

ADA: SOME INTERESTING HIGHLIGHTS

Packages

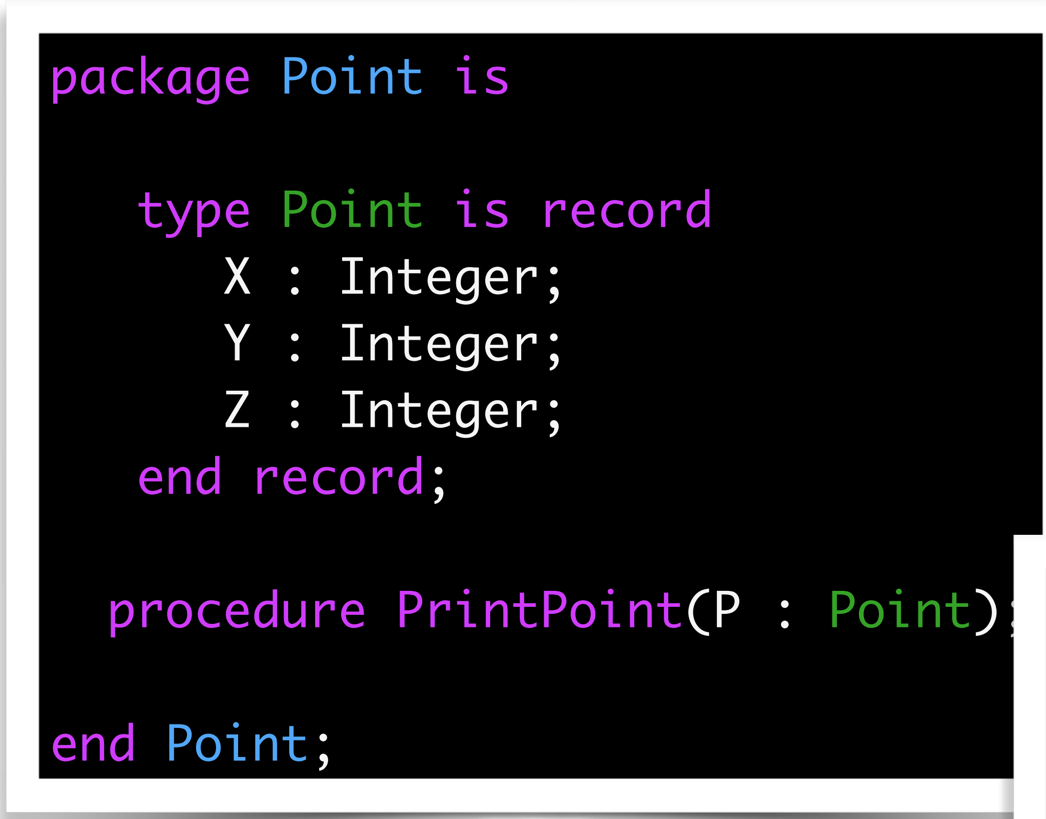
Packages



```
package Point is

   type Point is record
      X : Integer;
      Y : Integer;
      Z : Integer;
   end record;

  procedure PrintPoint(P : Point);

end Point;
```

Spec

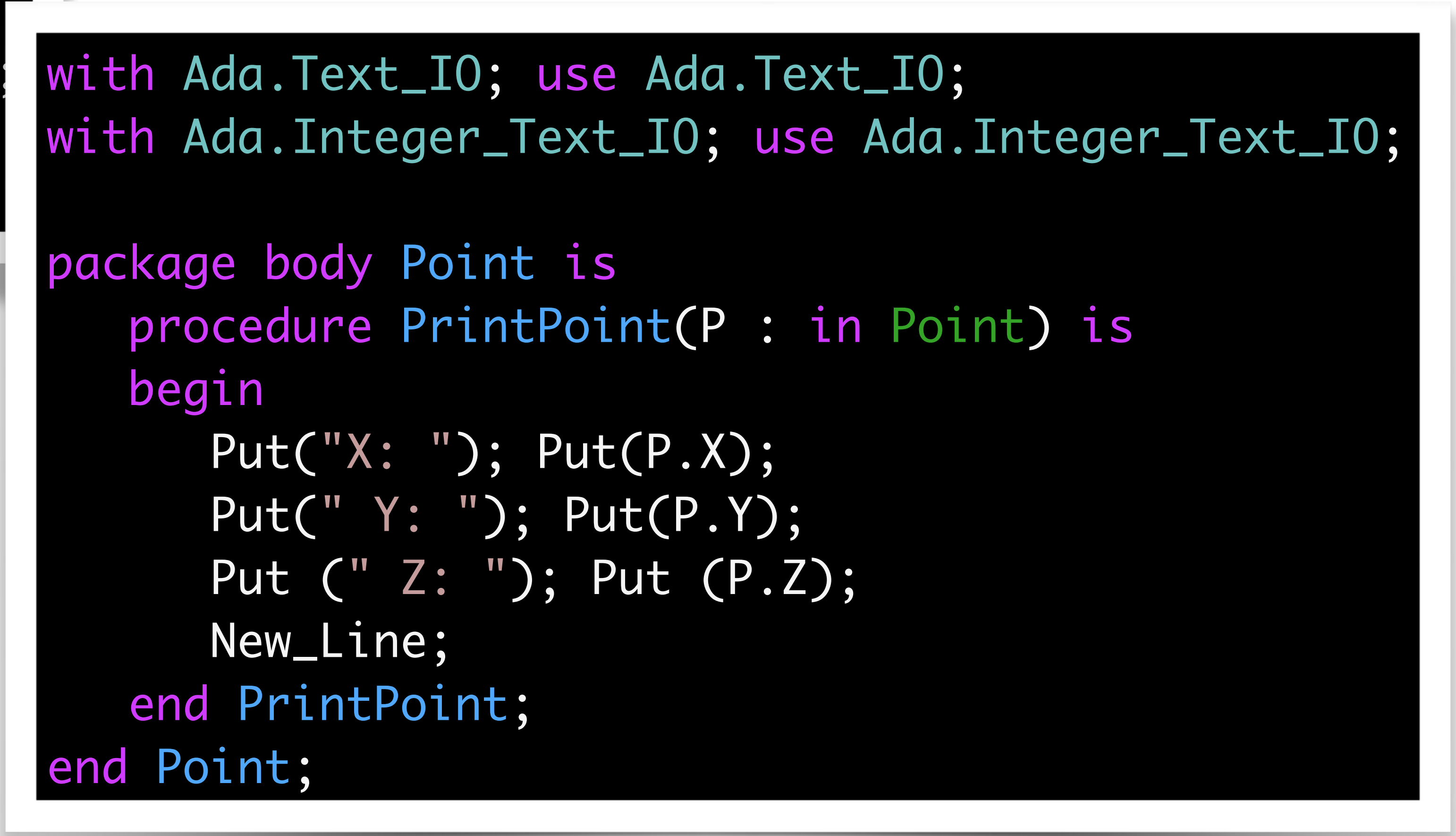
Packages



```
package Point is

   type Point is record
      X : Integer;
      Y : Integer;
      Z : Integer;
   end record;

  procedure PrintPoint(P : Point);

end Point;
```

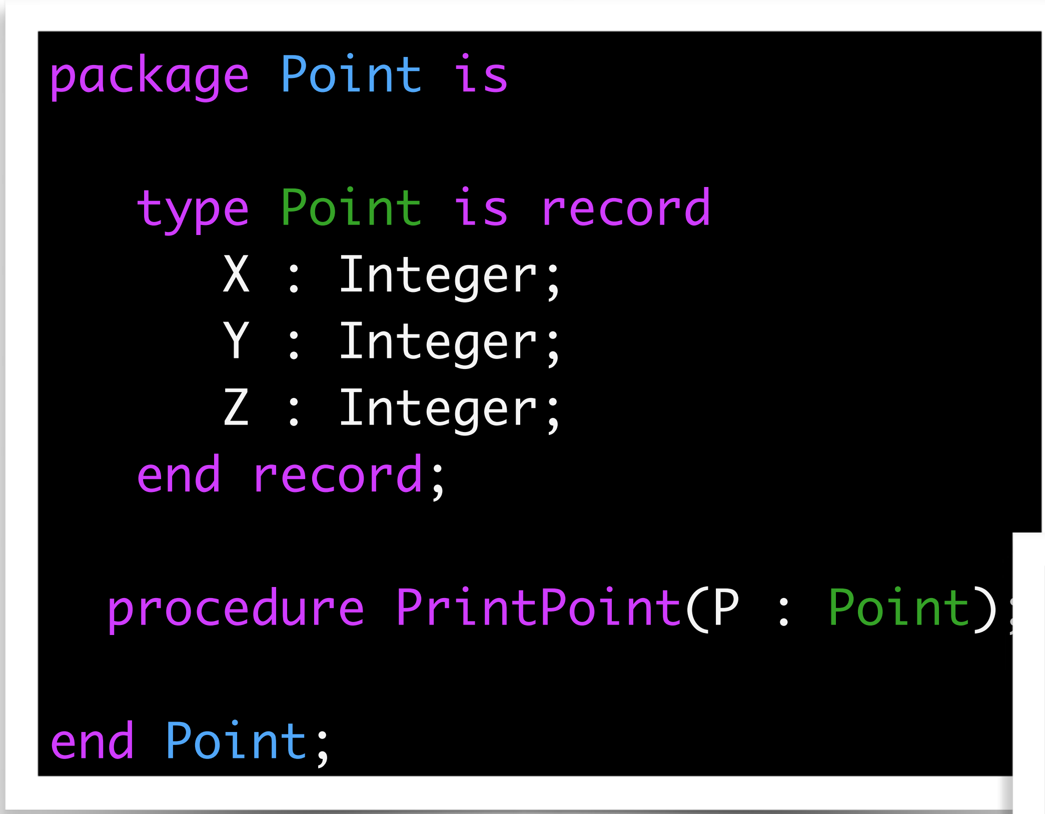
Spec

Body



```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

package body Point is
   procedure PrintPoint(P : in Point) is
   begin
      Put("X: "); Put(P.X);
      Put(" Y: "); Put(P.Y);
      Put (" Z: "); Put (P.Z);
      New_Line;
   end PrintPoint;
end Point;
```

Packages



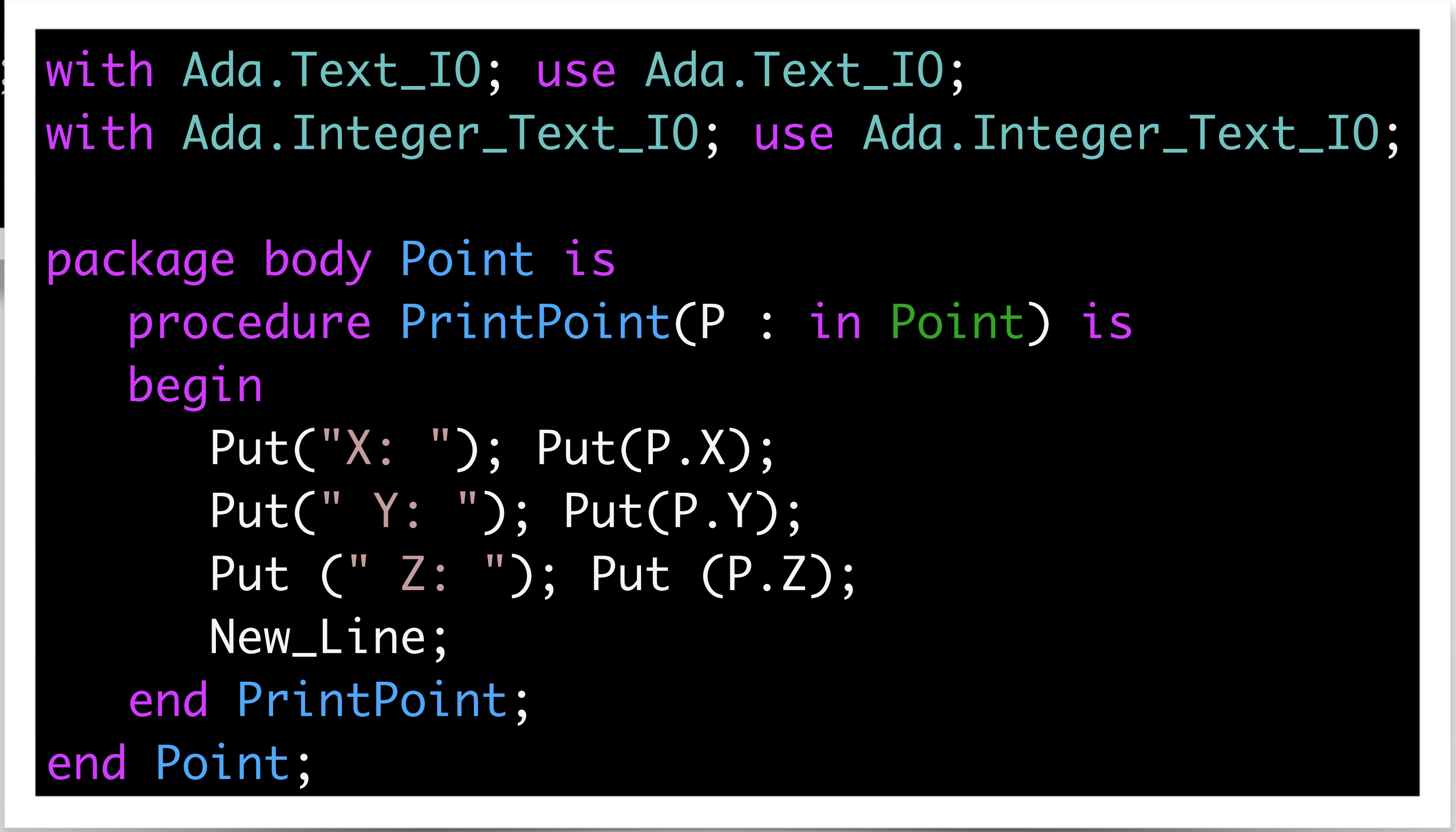
```
package Point is

   type Point is record
      X : Integer;
      Y : Integer;
      Z : Integer;
   end record;

  procedure PrintPoint(P : Point);

end Point;
```

Spec

the **interface**

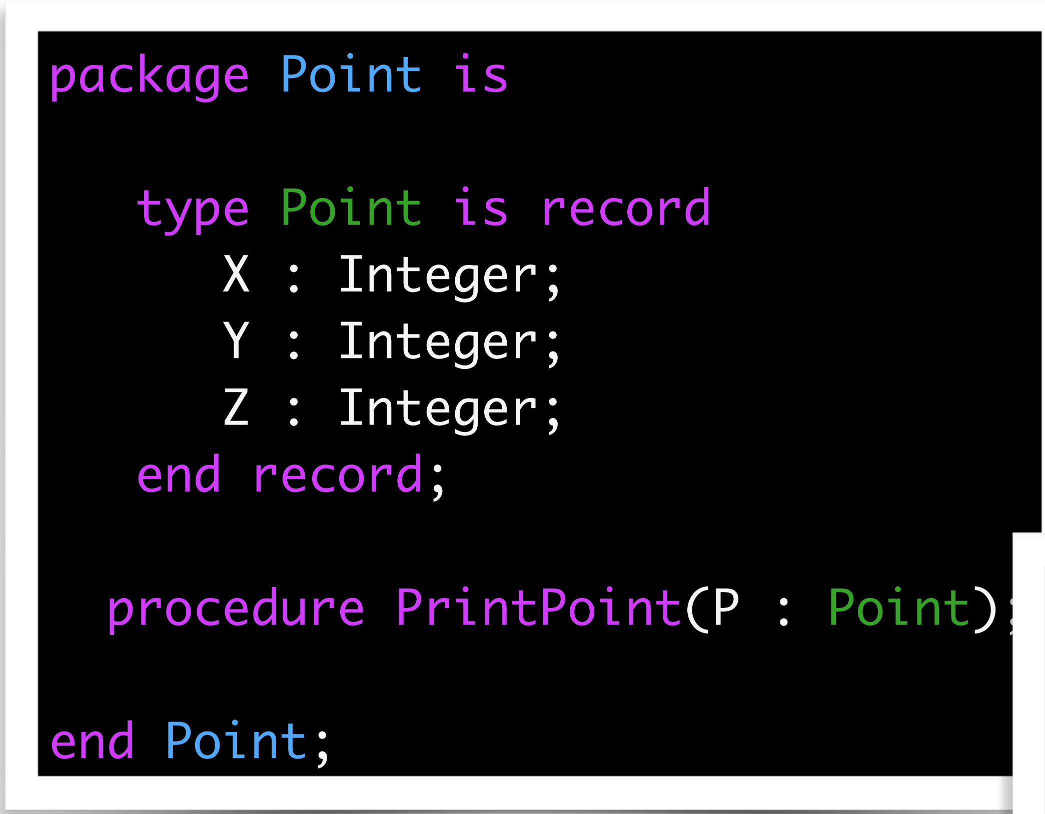
says “what”

Body



```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

package body Point is
   procedure PrintPoint(P : in Point) is
   begin
      Put("X: "); Put(P.X);
      Put(" Y: "); Put(P.Y);
      Put (" Z: "); Put (P.Z);
      New_Line;
   end PrintPoint;
end Point;
```

Packages



```
package Point is

   type Point is record
      X : Integer;
      Y : Integer;
      Z : Integer;
   end record;

  procedure PrintPoint(P : Point);

end Point;
```

Spec

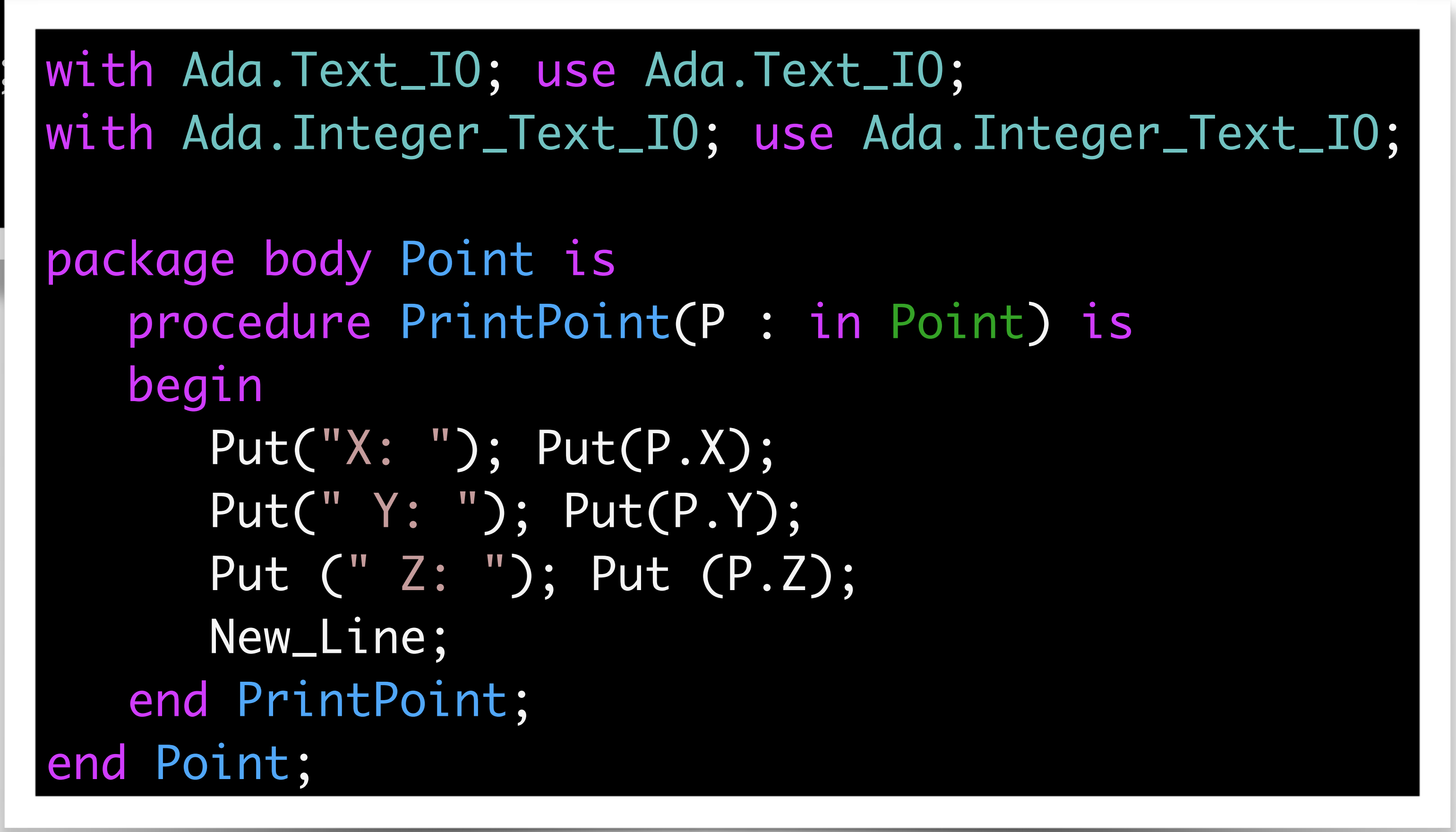
the **interface**

says “what”

says “how”

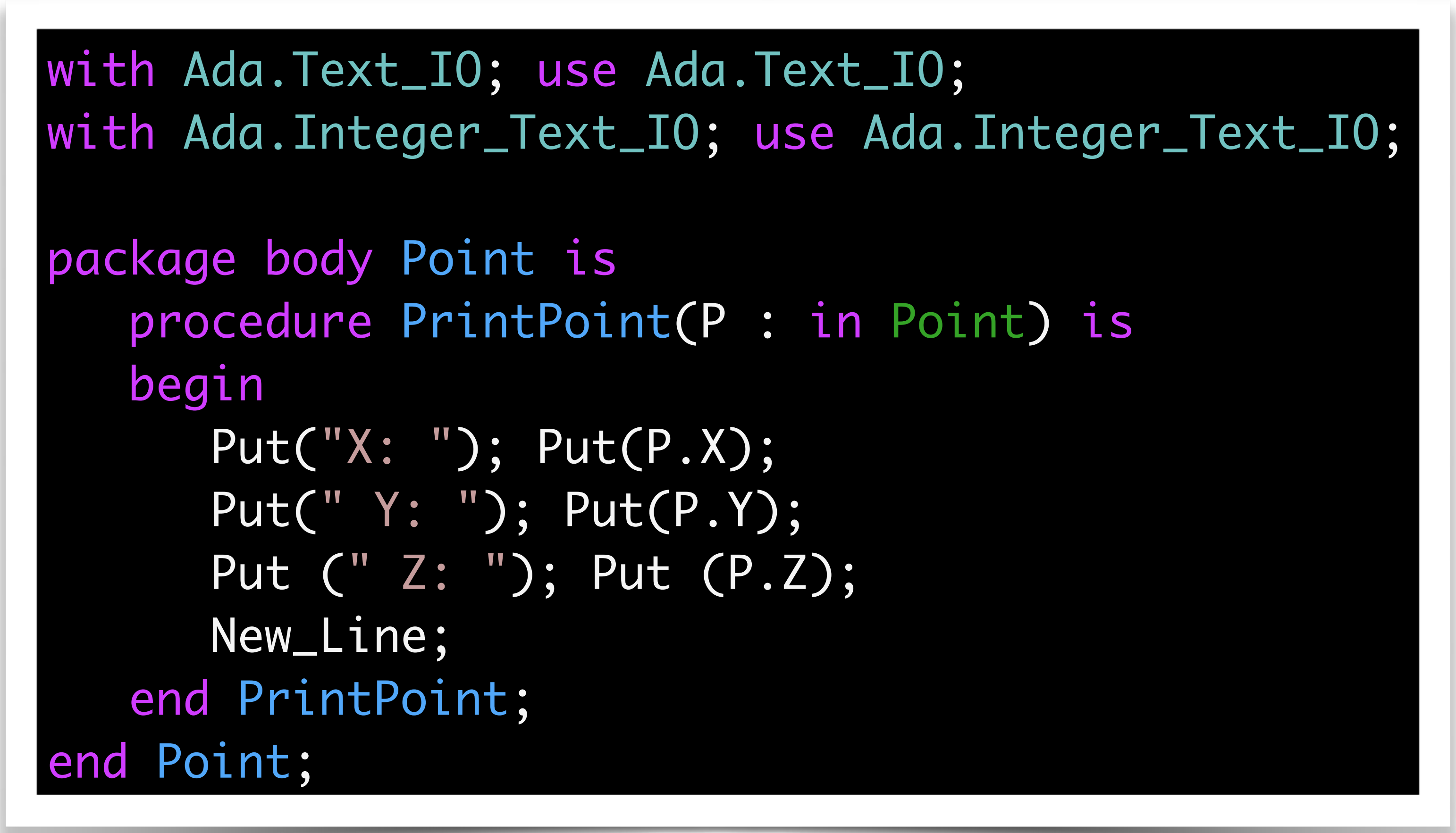
the **implementation**

Body



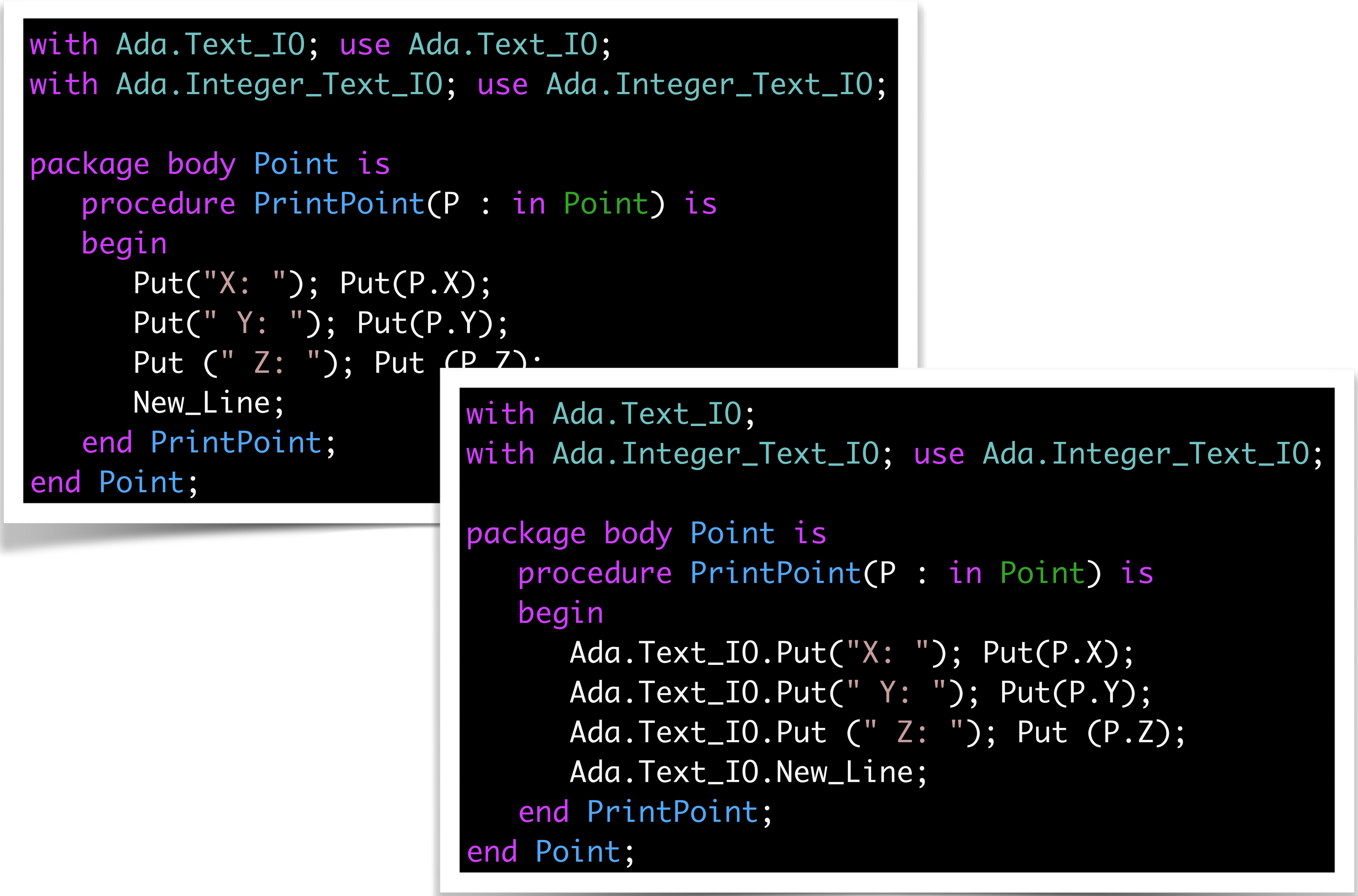
```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

package body Point is
   procedure PrintPoint(P : in Point) is
   begin
      Put("X: "); Put(P.X);
      Put(" Y: "); Put(P.Y);
      Put (" Z: "); Put (P.Z);
      New_Line;
   end PrintPoint;
end Point;
```

Packages: **with** and **use**



```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

package body Point is
   procedure PrintPoint(P : in Point) is
   begin
      Put("X: "); Put(P.X);
      Put(" Y: "); Put(P.Y);
      Put (" Z: "); Put (P.Z);
      New_Line;
   end PrintPoint;
end Point;
```

Packages: **with** and **use**

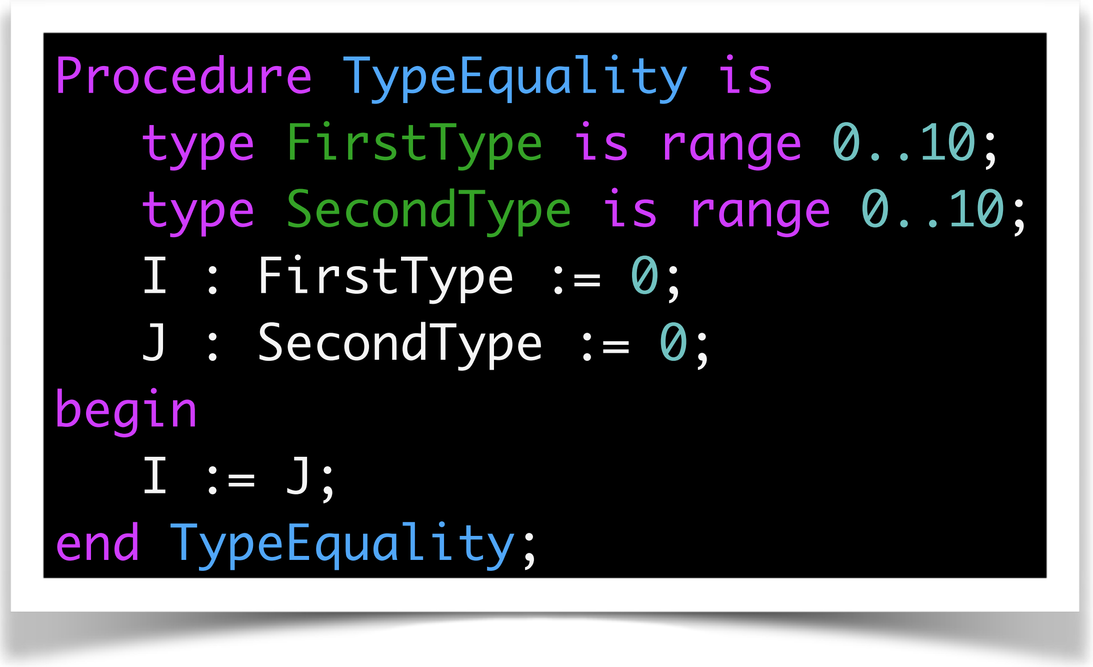


```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

package body Point is
   procedure PrintPoint(P : in Point) is
   begin
      Put("X: "); Put(P.X);
      Put(" Y: "); Put(P.Y);
      Put (" Z: "); Put (P.Z);
      New_Line;
   end PrintPoint;
end Point;
```

```
with Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

package body Point is
   procedure PrintPoint(P : in Point) is
   begin
      Ada.Text_IO.Put("X: "); Put(P.X);
      Ada.Text_IO.Put(" Y: "); Put(P.Y);
      Ada.Text_IO.Put (" Z: "); Put (P.Z);
      Ada.Text_IO.New_Line;
   end PrintPoint;
end Point;
```

Types

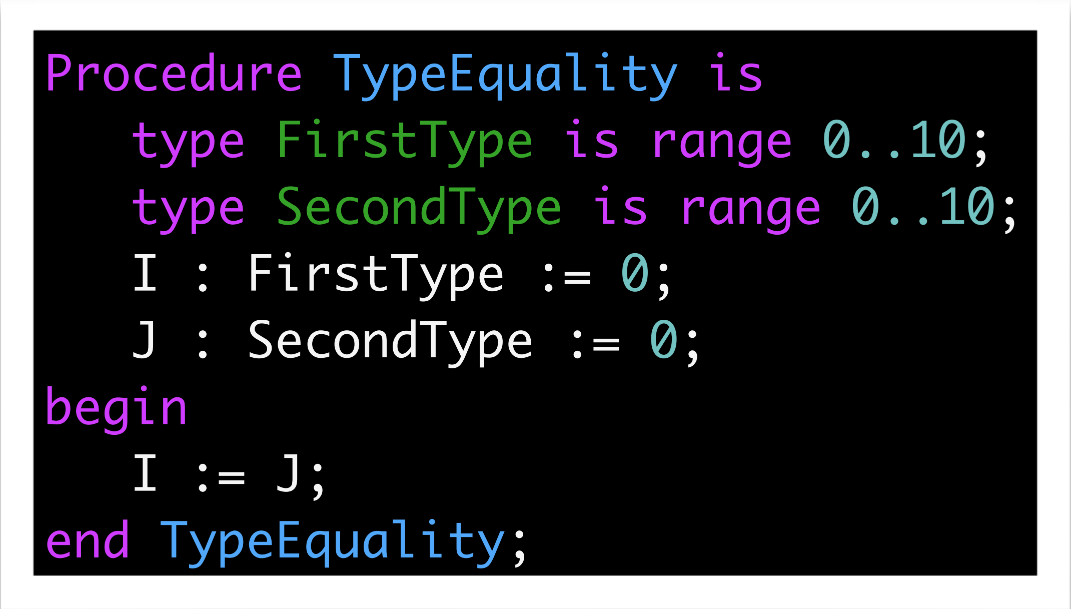
Type equality is by name



```
Procedure TypeEquality is
   type FirstType is range 0..10;
   type SecondType is range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end TypeEquality;
```

Types

Type equality is by name

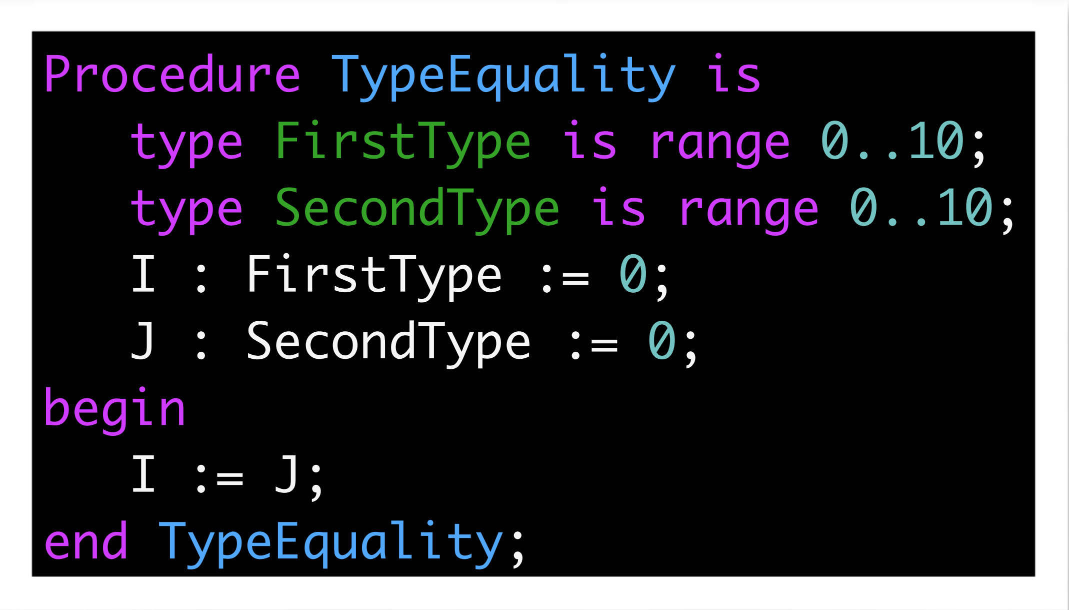


```
Procedure TypeEquality is
   type FirstType is range 0..10;
   type SecondType is range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end TypeEquality;
```

Will this code compile correctly?

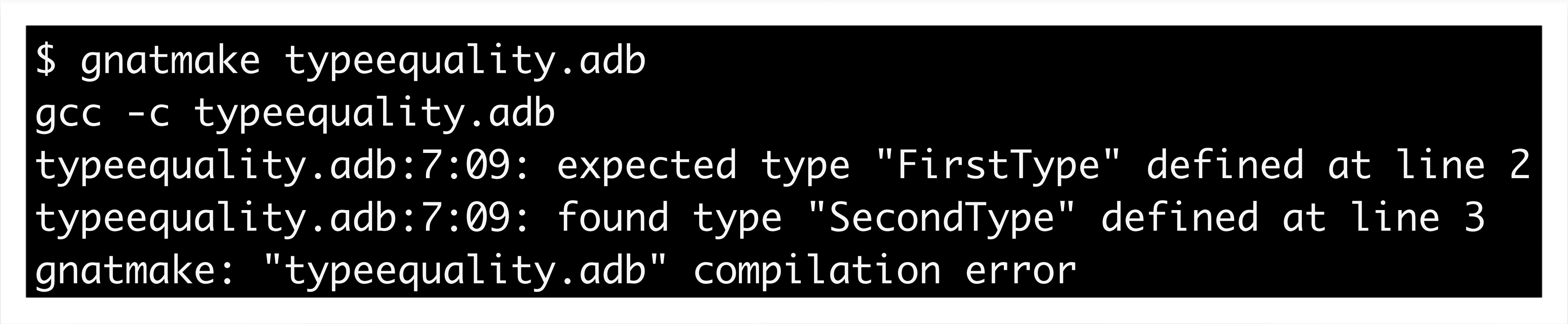
Types

Type equality is by name



```
Procedure TypeEquality is
   type FirstType is range 0..10;
   type SecondType is range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end TypeEquality;
```

Will this code compile correctly?

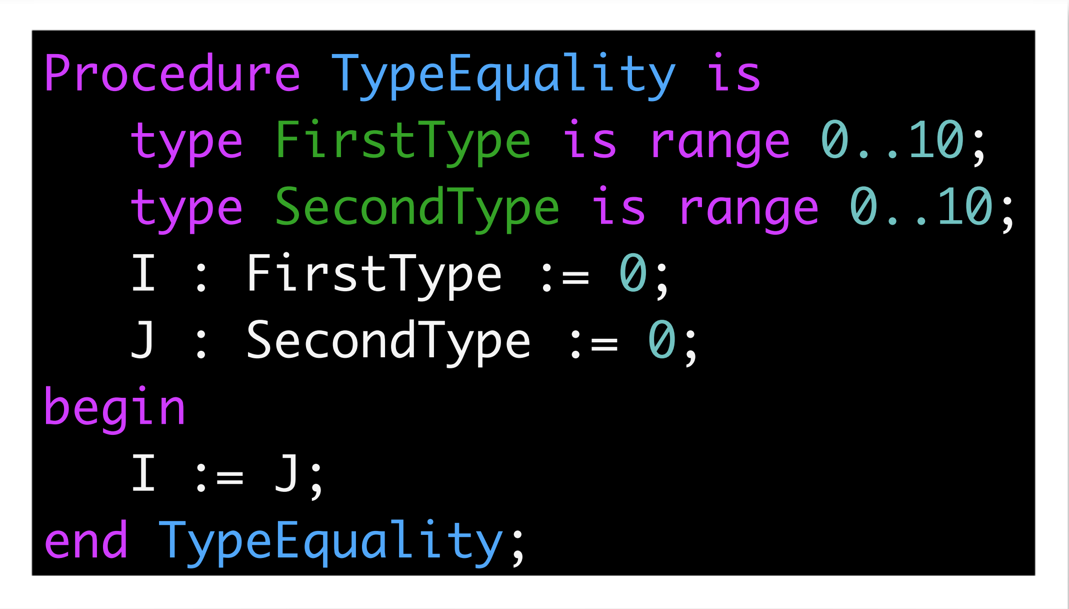


```
$ gnatmake typeequality.adb
gcc -c typeequality.adb
typeequality.adb:7:09: expected type "FirstType" defined at line 2
typeequality.adb:7:09: found type "SecondType" defined at line 3
gnatmake: "typeequality.adb" compilation error
```

Types

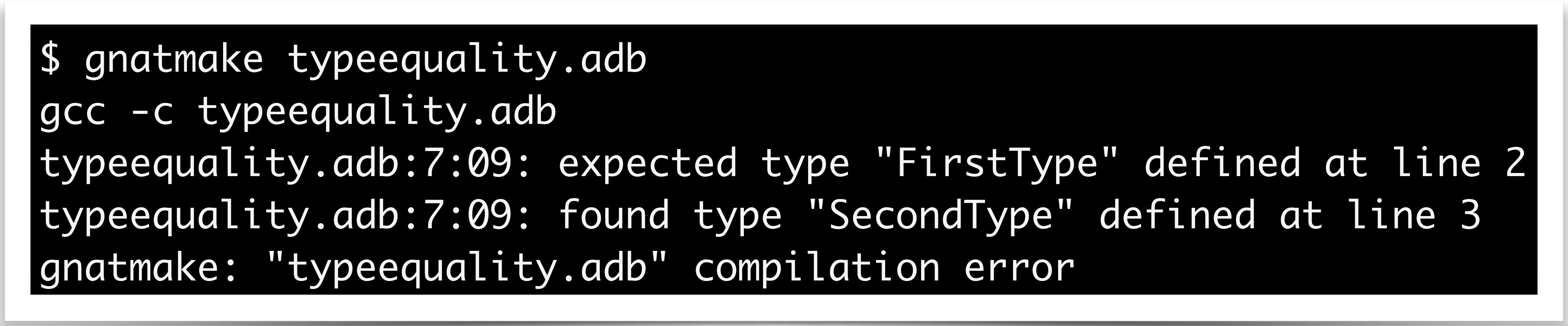
Type equality is by name

What errors does this catch statically?



```
Procedure TypeEquality is
   type FirstType is range 0..10;
   type SecondType is range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end TypeEquality;
```

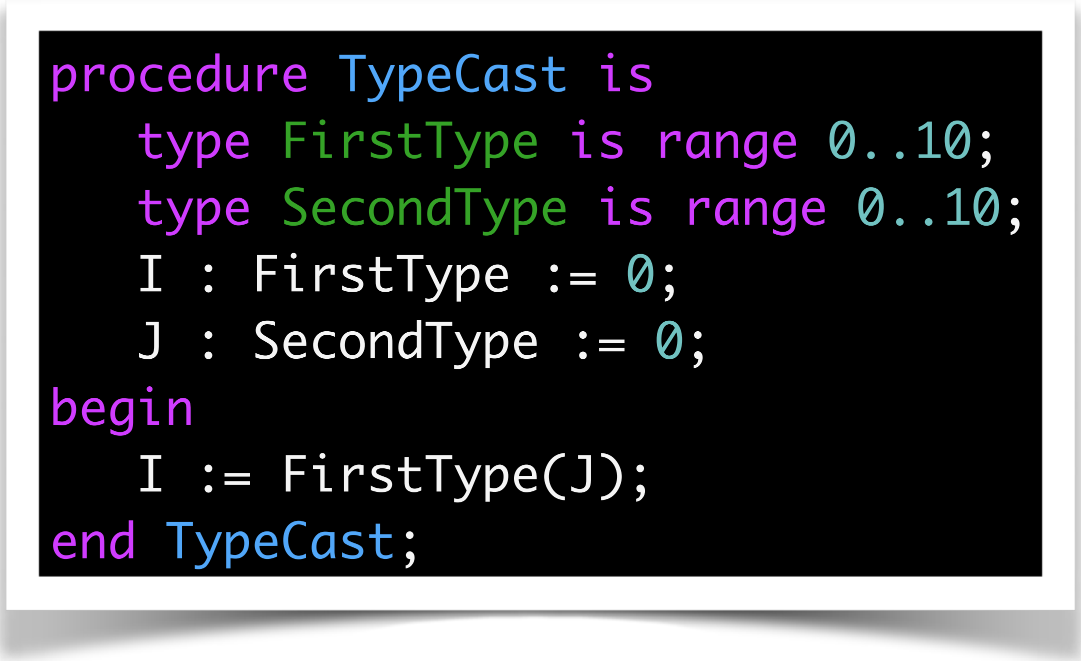
Will this code compile correctly?



```
$ gnatmake typeequality.adb
gcc -c typeequality.adb
typeequality.adb:7:09: expected type "FirstType" defined at line 2
typeequality.adb:7:09: found type "SecondType" defined at line 3
gnatmake: "typeequality.adb" compilation error
```

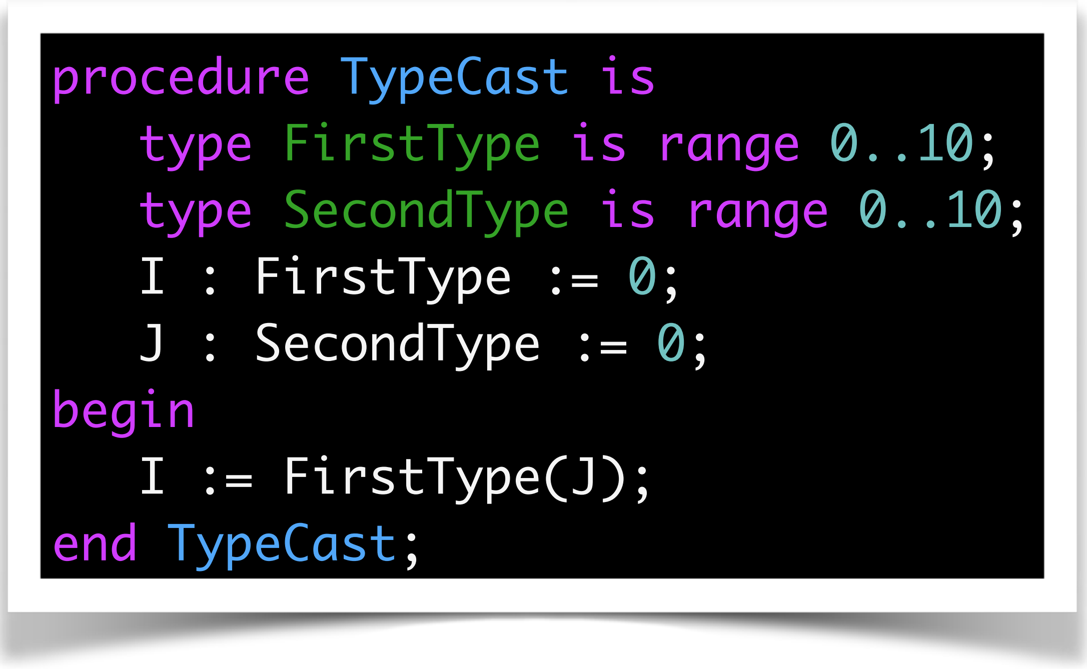
Type Casts

Type Casts



```
procedure TypeCast is
   type FirstType is range 0..10;
   type SecondType is range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := FirstType(J);
end TypeCast;
```

Type Casts

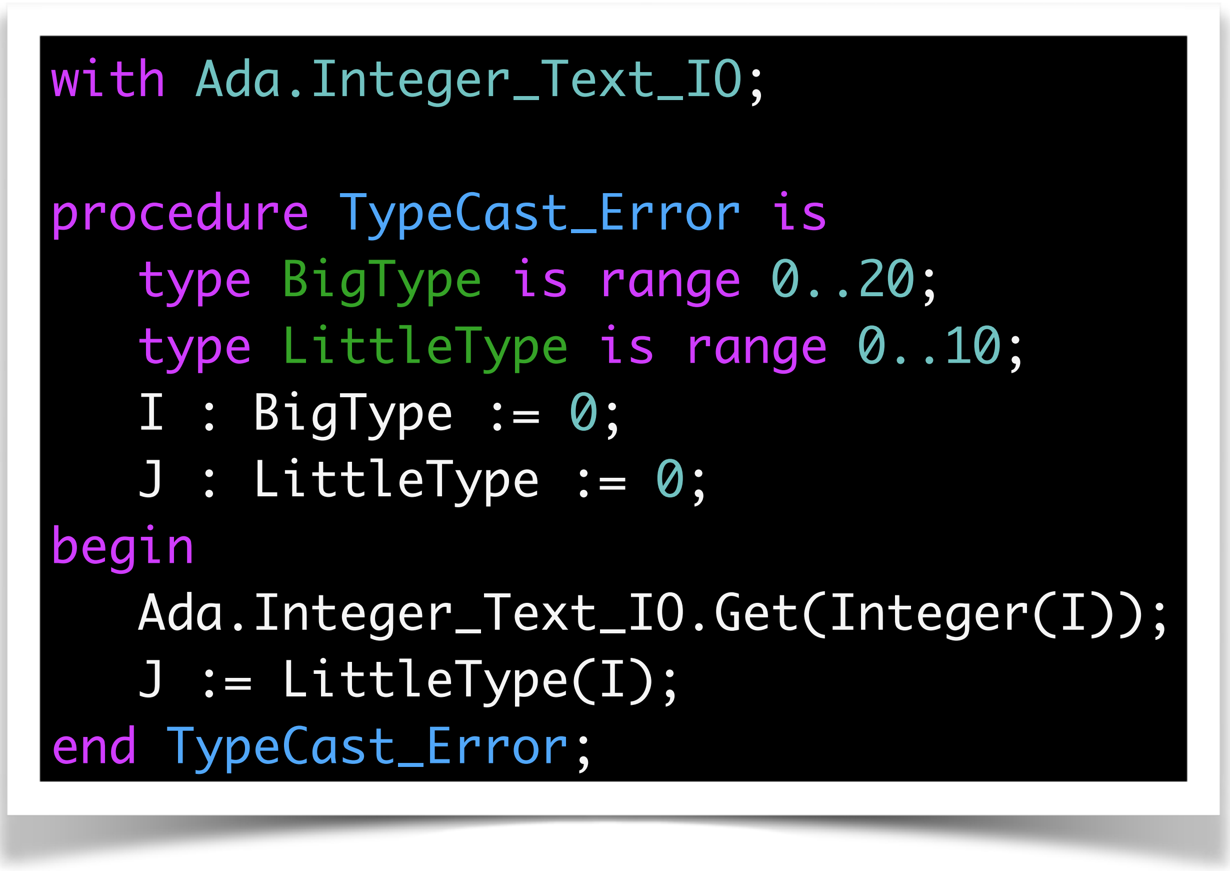


```
procedure TypeCast is
   type FirstType is range 0..10;
   type SecondType is range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := FirstType(J);
end TypeCast;
```

This compiles and runs without error.

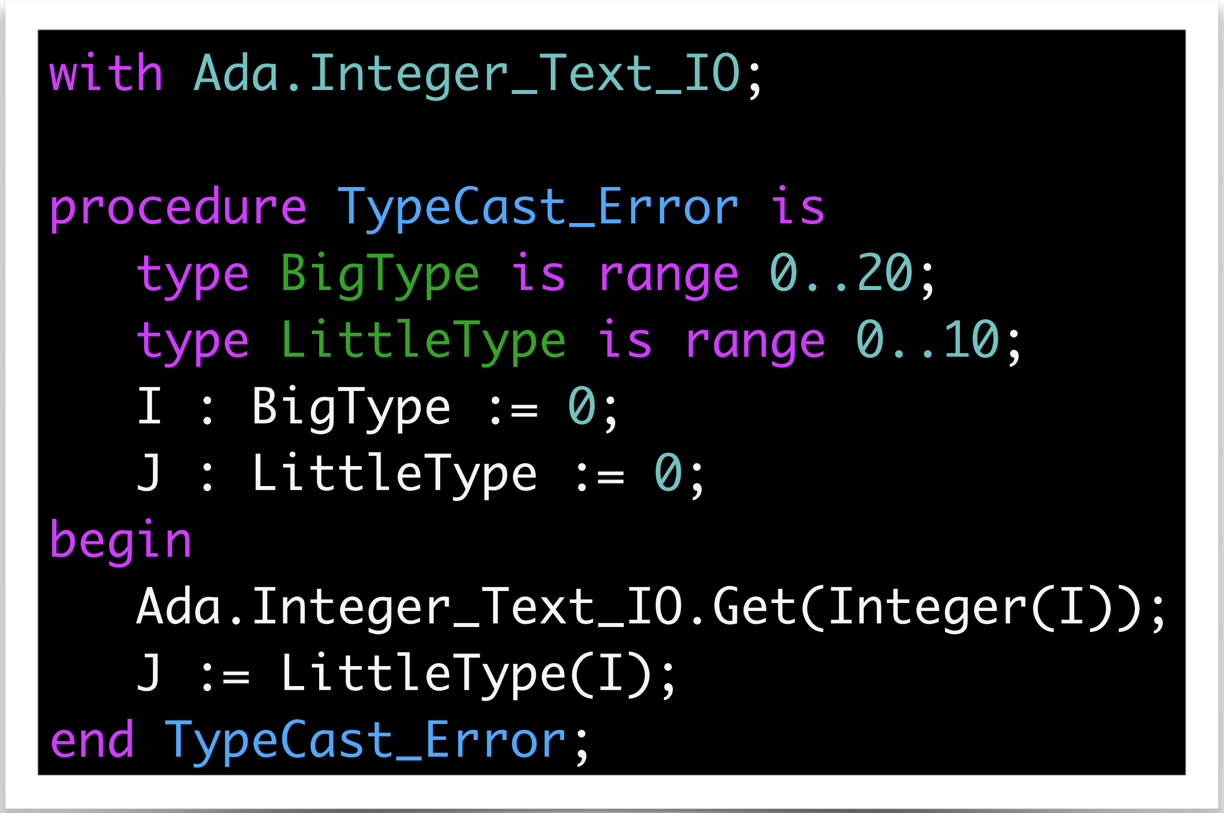
Type Casts and Errors

Type Casts and Errors



```
with Ada.Integer_Text_IO;

procedure TypeCast_Error is
   type BigType is range 0..20;
   type LittleType is range 0..10;
   I : BigType := 0;
   J : LittleType := 0;
begin
   Ada.Integer_Text_IO.Get(Integer(I));
   J := LittleType(I);
end TypeCast_Error;
```

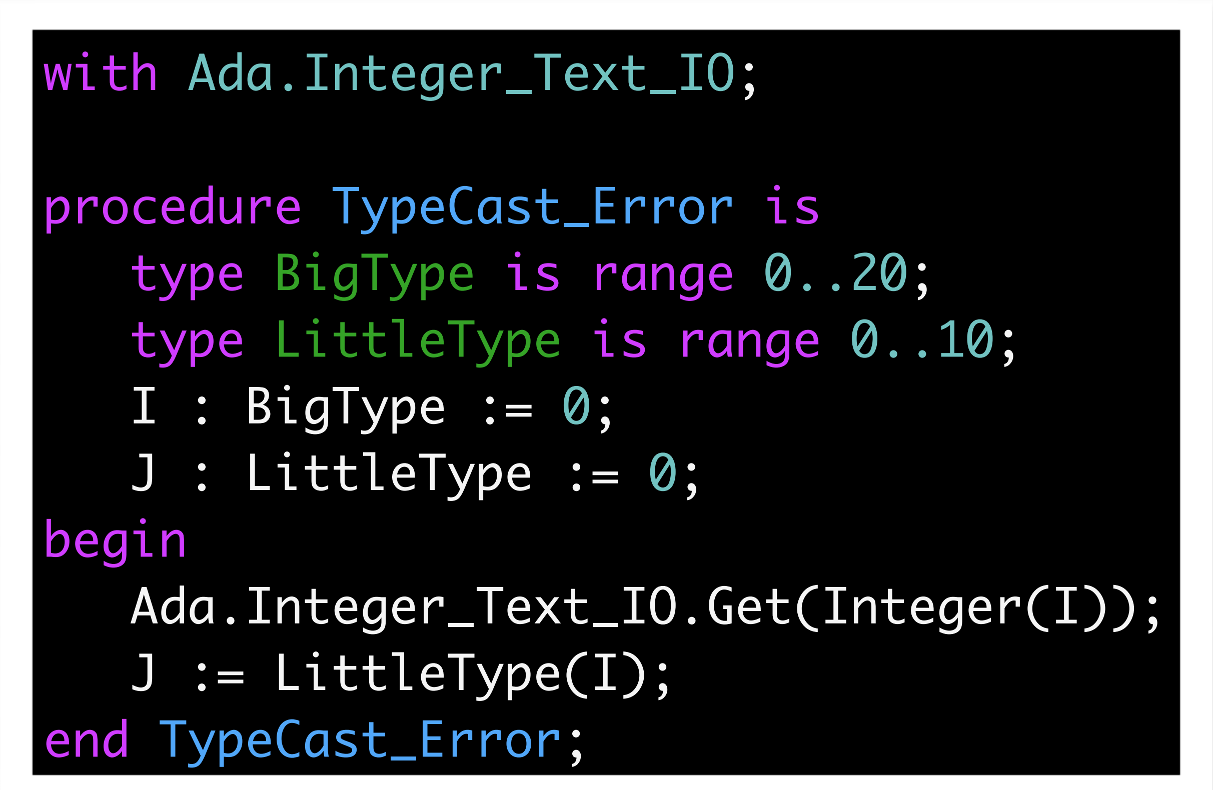
Type Casts and Errors



```
with Ada.Integer_Text_IO;

procedure TypeCast_Error is
   type BigType is range 0..20;
   type LittleType is range 0..10;
   I : BigType := 0;
   J : LittleType := 0;
begin
   Ada.Integer_Text_IO.Get(Integer(I));
   J := LittleType(I);
end TypeCast_Error;
```

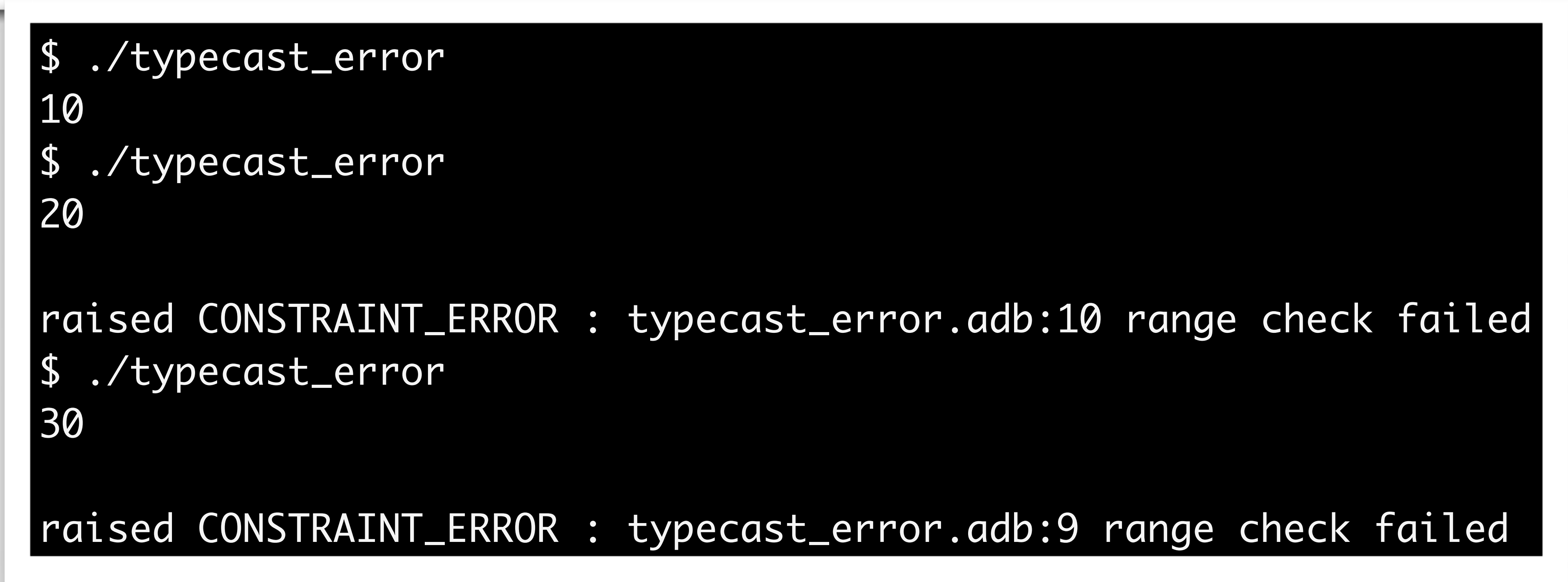
Compiles without error.

Type Casts and Errors



```
with Ada.Integer_Text_IO;

procedure TypeCast_Error is
   type BigType is range 0..20;
   type LittleType is range 0..10;
   I : BigType := 0;
   J : LittleType := 0;
begin
   Ada.Integer_Text_IO.Get(Integer(I));
   J := LittleType(I);
end TypeCast_Error;
```

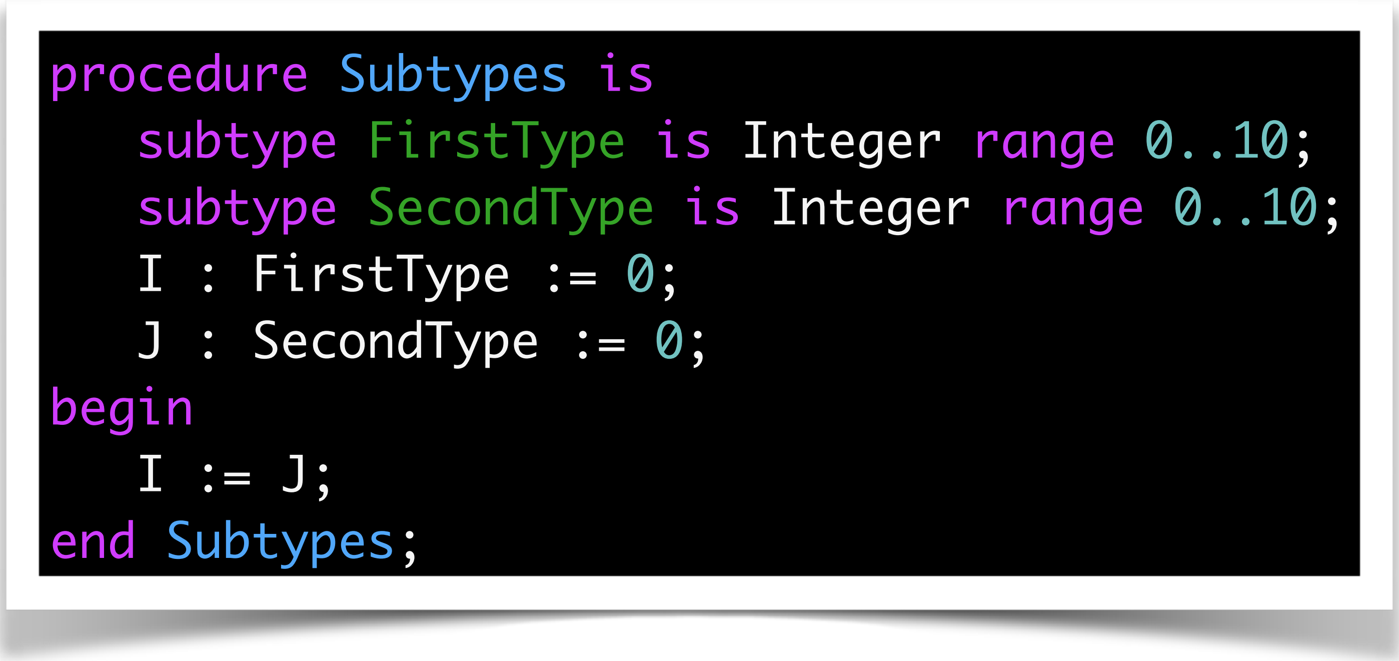
Compiles without error.



```
$ ./typecast_error
10
$ ./typecast_error
20

raised CONSTRAINT_ERROR : typecast_error.adb:10 range check failed
$ ./typecast_error
30

raised CONSTRAINT_ERROR : typecast_error.adb:9 range check failed
```

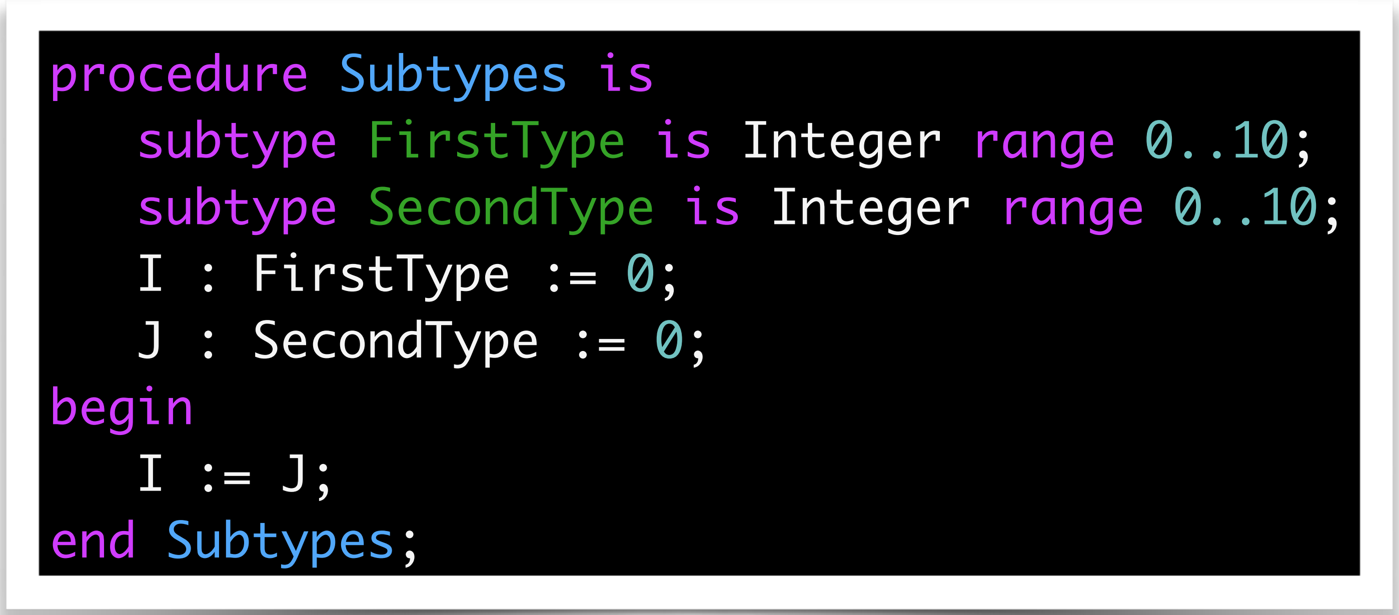
Subtypes

Subtypes



```
procedure Subtypes is
   subtype FirstType is Integer range 0..10;
   subtype SecondType is Integer range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end Subtypes;
```

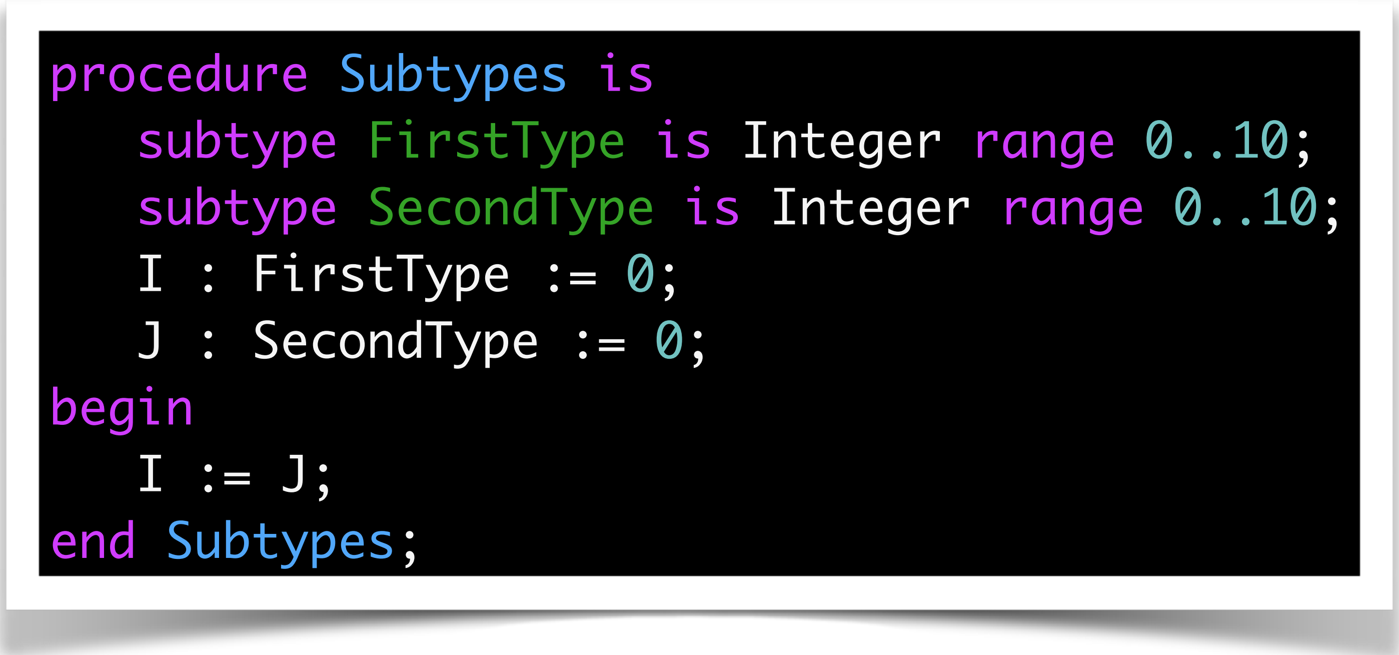
Subtypes



```
procedure Subtypes is
   subtype FirstType is Integer range 0..10;
   subtype SecondType is Integer range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end Subtypes;
```

This compiles and executes with no errors.

Subtypes

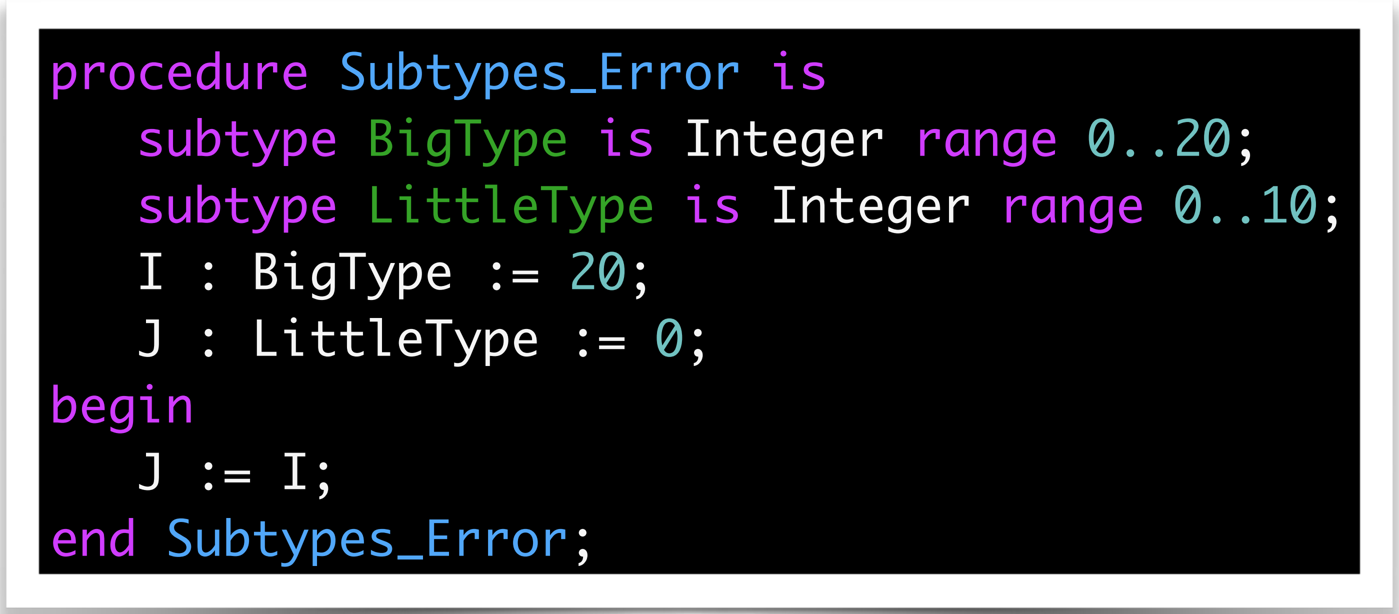


```
procedure Subtypes is
   subtype FirstType is Integer range 0..10;
   subtype SecondType is Integer range 0..10;
   I : FirstType := 0;
   J : SecondType := 0;
begin
   I := J;
end Subtypes;
```

This compiles and executes with no errors.

Casting is effectively implicit between subtypes.

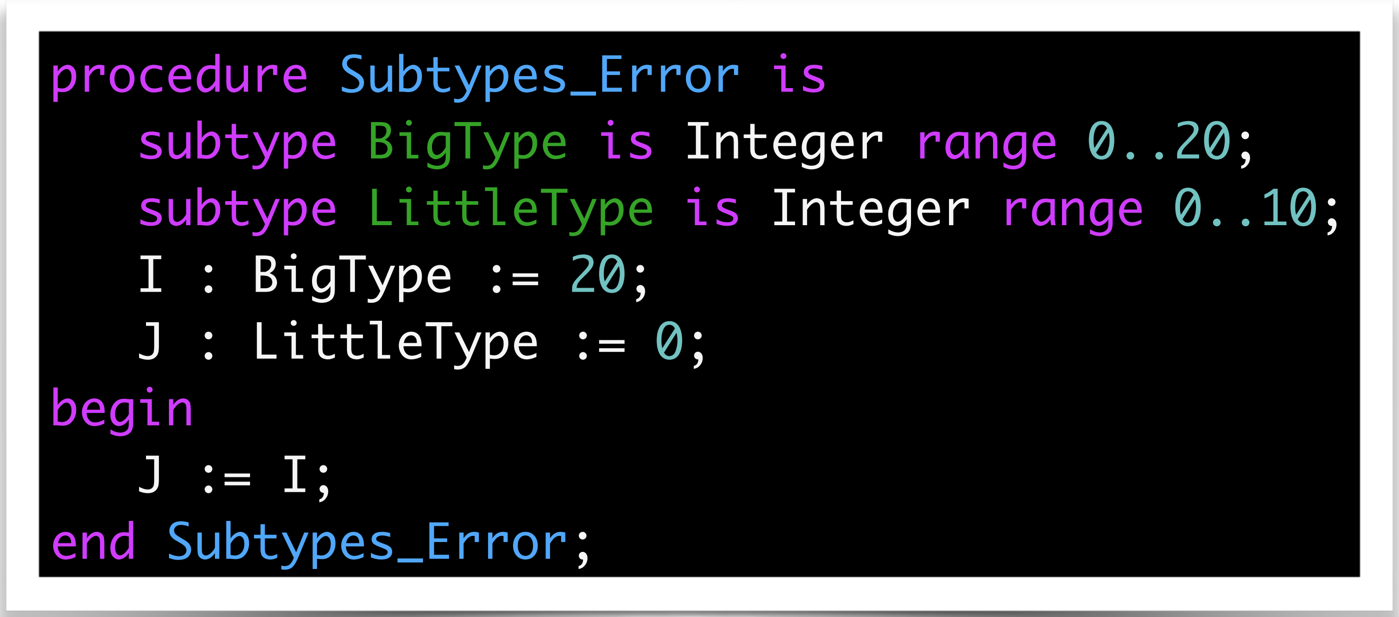
Subtypes



```
procedure Subtypes_Error is
   subtype BigType is Integer range 0..20;
   subtype LittleType is Integer range 0..10;
   I : BigType := 20;
   J : LittleType := 0;
begin
   J := I;
end Subtypes_Error;
```

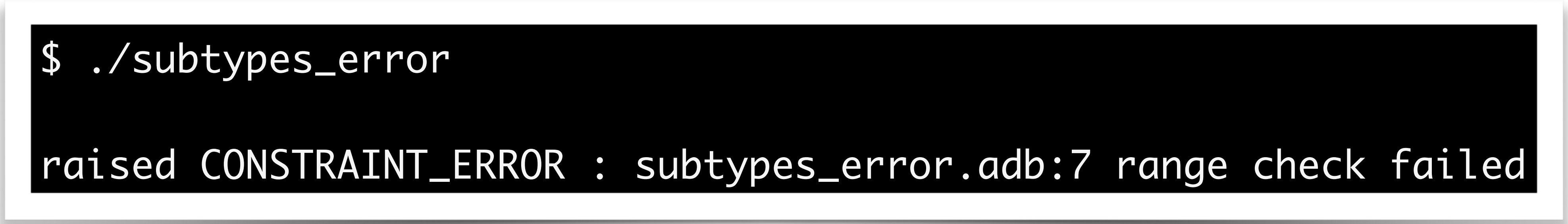
What happens when we run this?

Subtypes



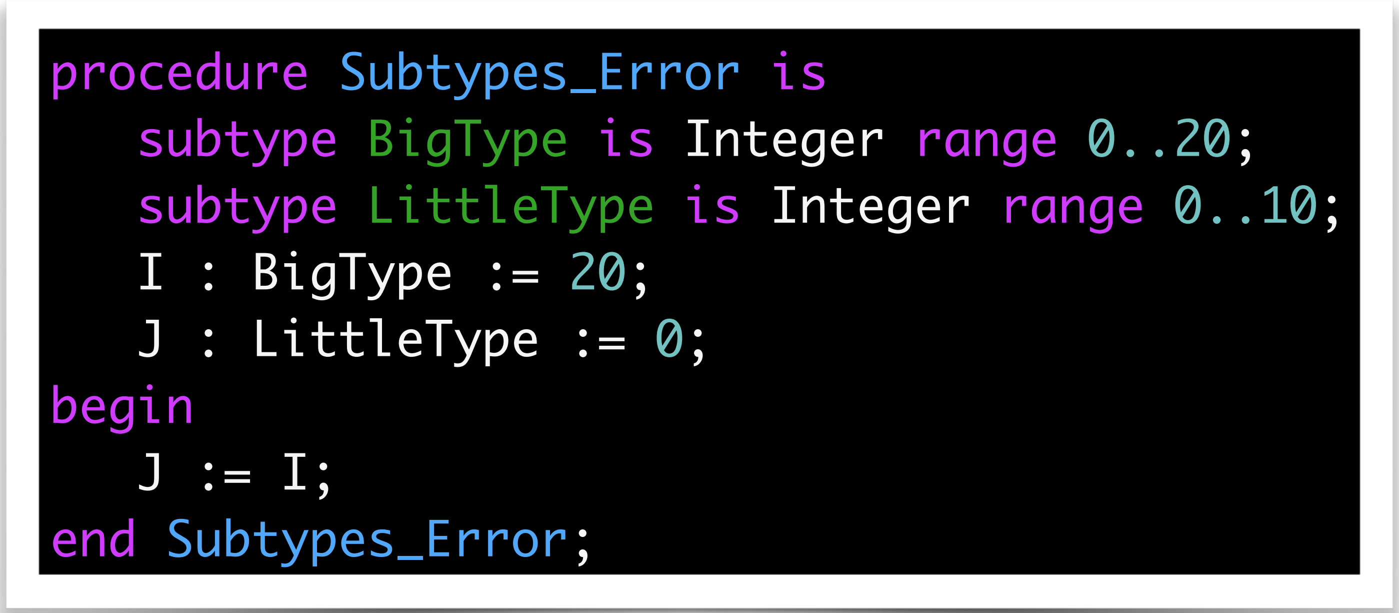
```
procedure Subtypes_Error is
   subtype BigType is Integer range 0..20;
   subtype LittleType is Integer range 0..10;
   I : BigType := 20;
   J : LittleType := 0;
begin
   J := I;
end Subtypes_Error;
```

What happens when we run this?



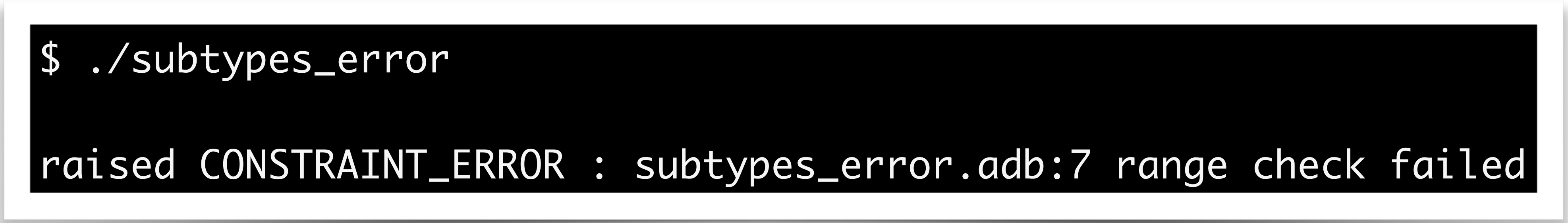
```
$ ./subtypes_error

raised CONSTRAINT_ERROR : subtypes_error.adb:7 range check failed
```

Subtypes



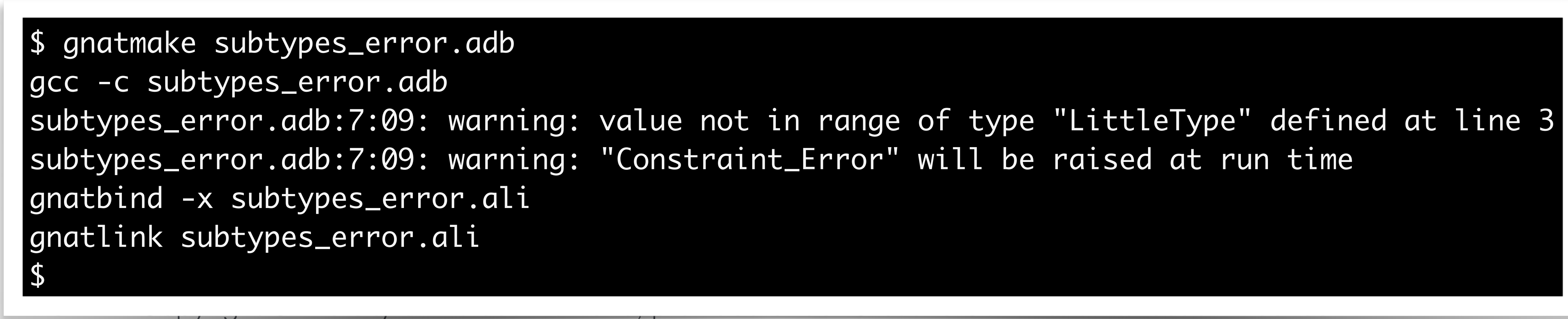
```
procedure Subtypes_Error is
   subtype BigType is Integer range 0..20;
   subtype LittleType is Integer range 0..10;
   I : BigType := 20;
   J : LittleType := 0;
begin
   J := I;
end Subtypes_Error;
```

What happens when we run this?



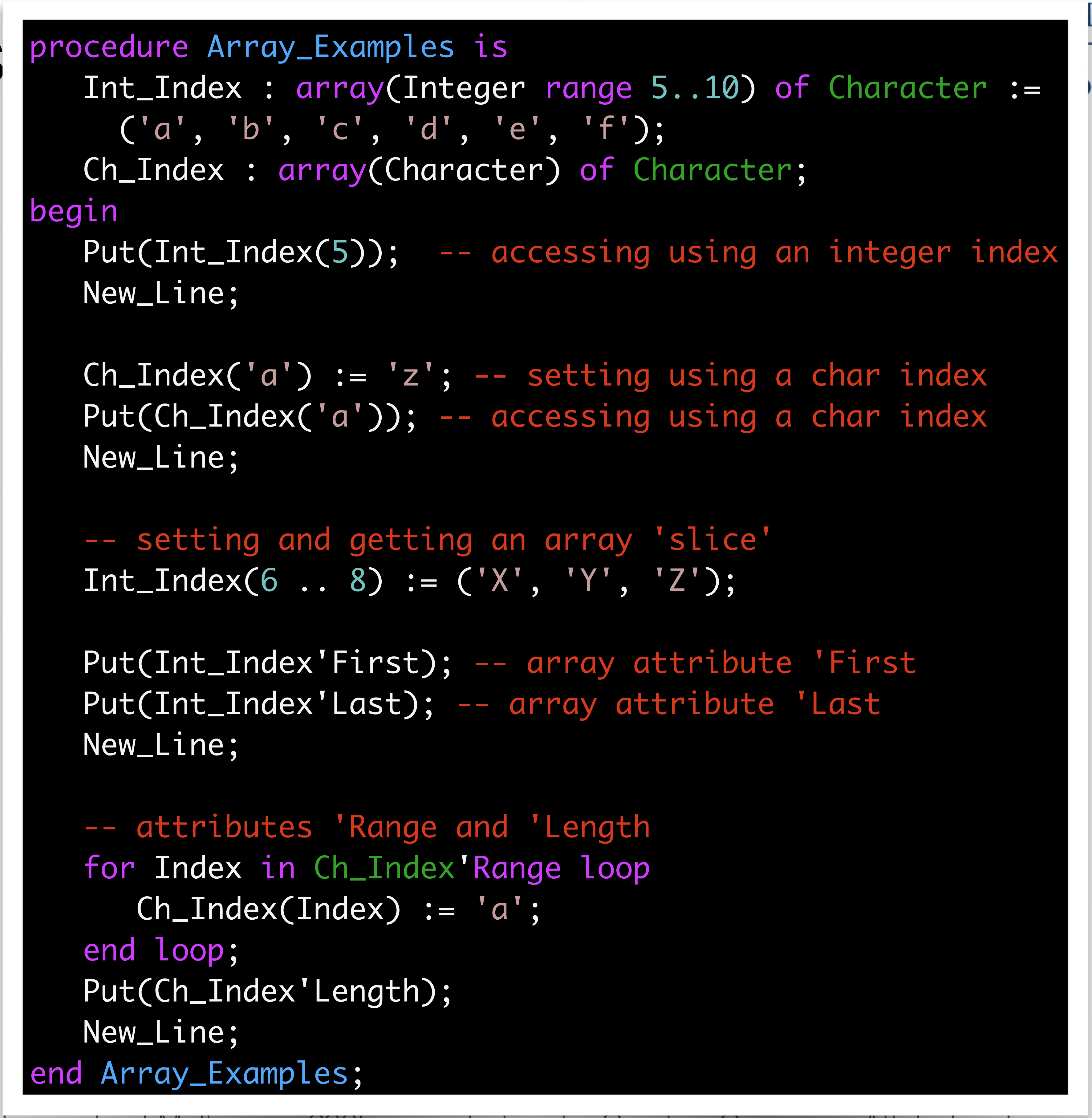
```
$ ./subtypes_error

raised CONSTRAINT_ERROR : subtypes_error.adb:7 range check failed
```

btw



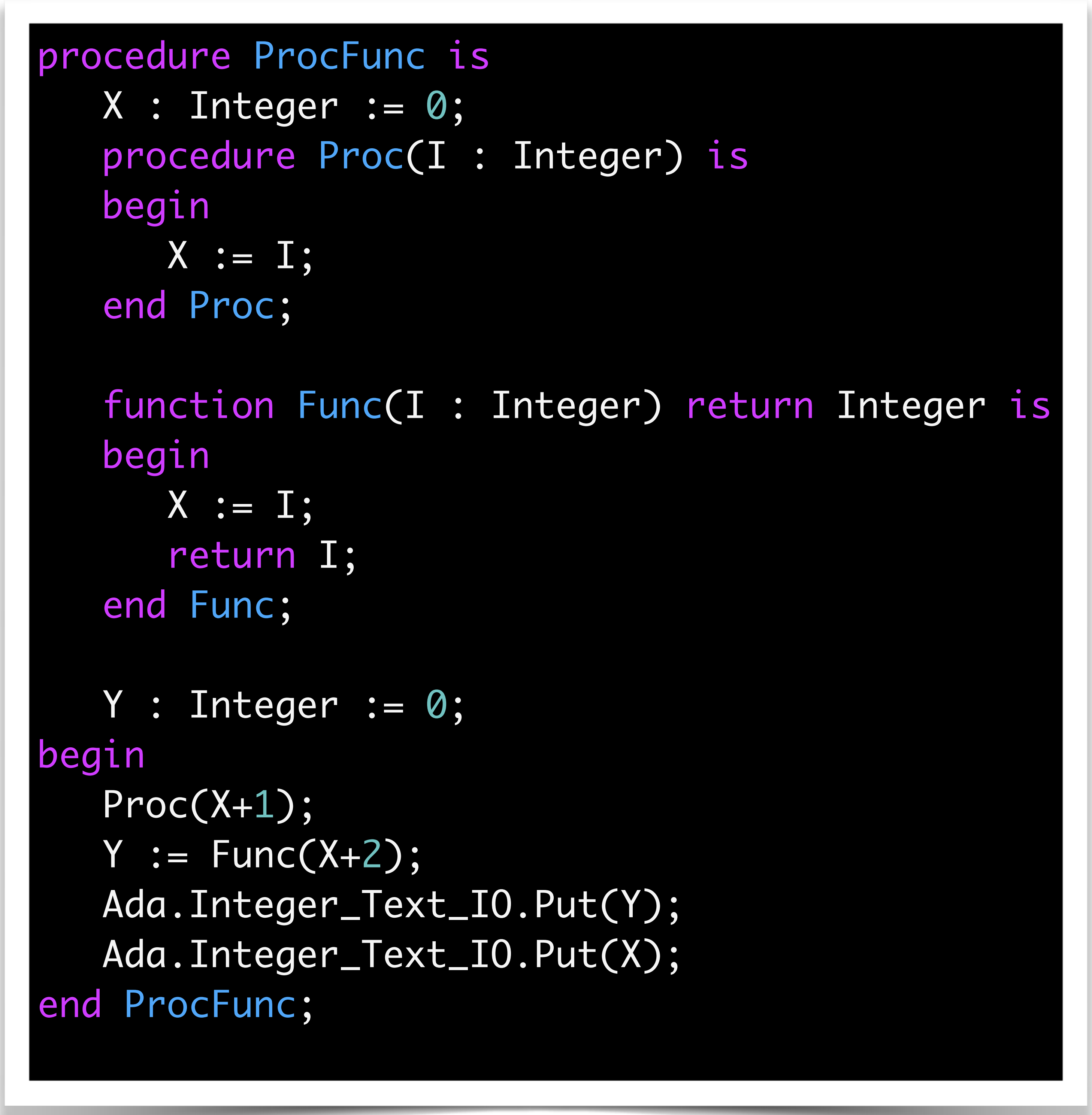
```
$ gnatmake subtypes_error.adb
gcc -c subtypes_error.adb
subtypes_error.adb:7:09: warning: value not in range of type "LittleType" defined at line 3
subtypes_error.adb:7:09: warning: "Constraint_Error" will be raised at run time
gnatbind -x subtypes_error.ali
gnatlink subtypes_error.ali
$
```

Arrays



```
procedure Array_Examples is
   Int_Index : array(Integer range 5..10) of Character :=
     ('a', 'b', 'c', 'd', 'e', 'f');
   Ch_Index : array(Character) of Character;
begin
   Put(Int_Index(5));  -- accessing using an integer index
   New_Line;

   Ch_Index('a') := 'z'; -- setting using a char index
   Put(Ch_Index('a')); -- accessing using a char index
   New_Line;

   -- setting and getting an array 'slice'
   Int_Index(6 .. 8) := ('X', 'Y', 'Z');

   Put(Int_Index'First); -- array attribute 'First
   Put(Int_Index'Last); -- array attribute 'Last
   New_Line;

   -- attributes 'Range and 'Length
   for Index in Ch_Index'Range loop
      Ch_Index(Index) := 'a';
   end loop;
   Put(Ch_Index'Length);
   New_Line;
end Array_Examples;
```

Procedures and Functions

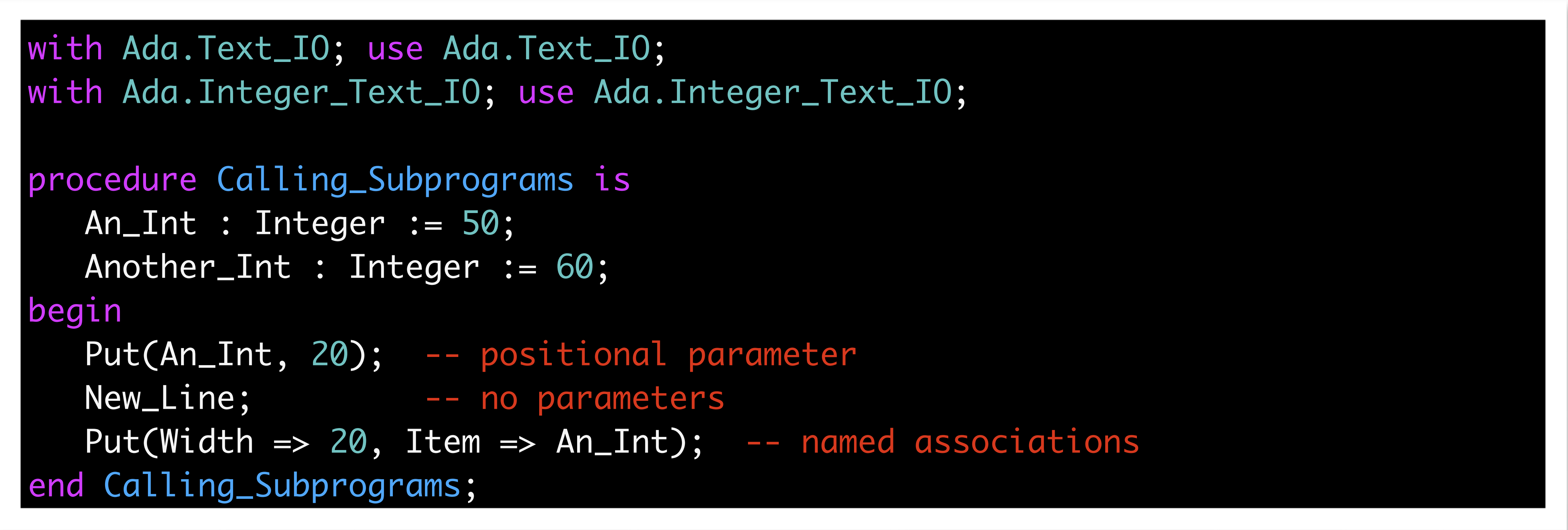


```
procedure ProcFunc is
   X : Integer := 0;
   procedure Proc(I : Integer) is
   begin
      X := I;
   end Proc;

   function Func(I : Integer) return Integer is
   begin
      X := I;
      return I;
   end Func;

   Y : Integer := 0;
begin
   Proc(X+1);
   Y := Func(X+2);
   Ada.Integer_Text_IO.Put(Y);
   Ada.Integer_Text_IO.Put(X);
end ProcFunc;
```

Nesting, static scope

Functions with side-effects (disallowed in SPARK)

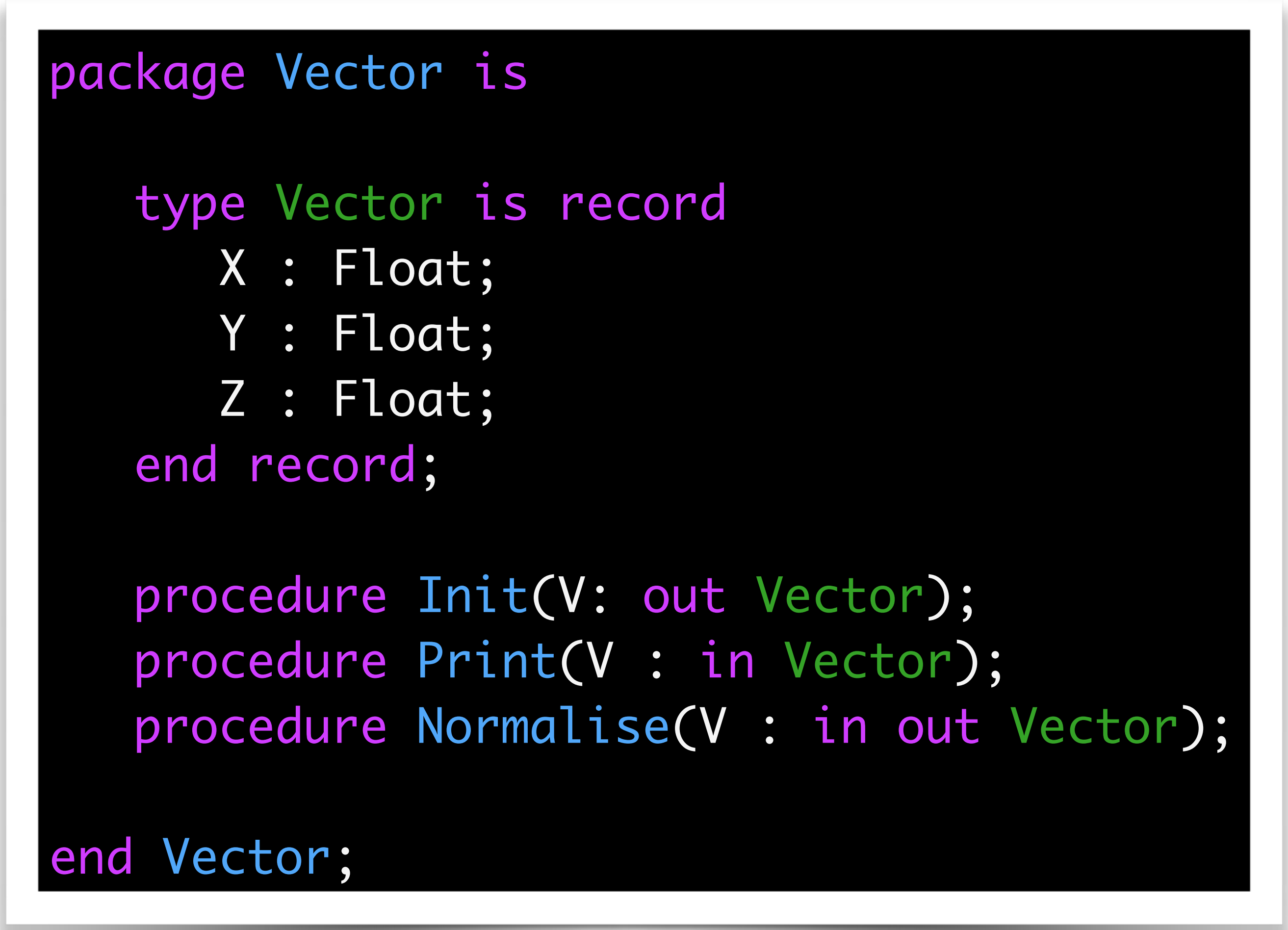
Parameter Passing



```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Calling_Subprograms is
   An_Int : Integer := 50;
   Another_Int : Integer := 60;
begin
   Put(An_Int, 20);   -- positional parameter
   New_Line;          -- no parameters
   Put(Width => 20, Item => An_Int);   -- named associations
end Calling_Subprograms;
```

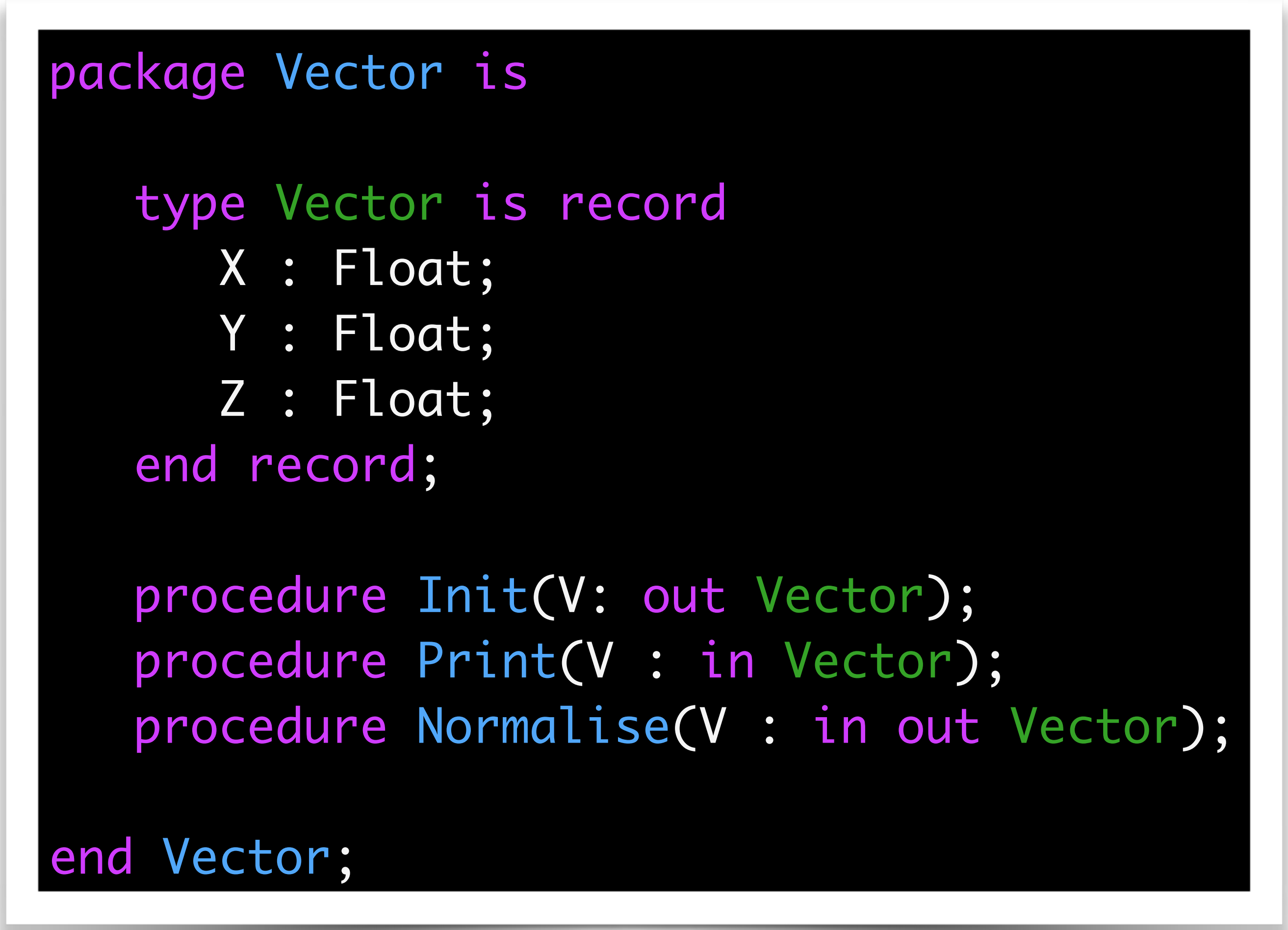
names come from names of **formal parameters** (which is why they must match between package spec and body)

Parameter Modes



```
package Vector is

   type Vector is record
      X : Float;
      Y : Float;
      Z : Float;
   end record;

   procedure Init(V: out Vector);
   procedure Print(V : in Vector);
   procedure Normalise(V : in out Vector);

end Vector;
```

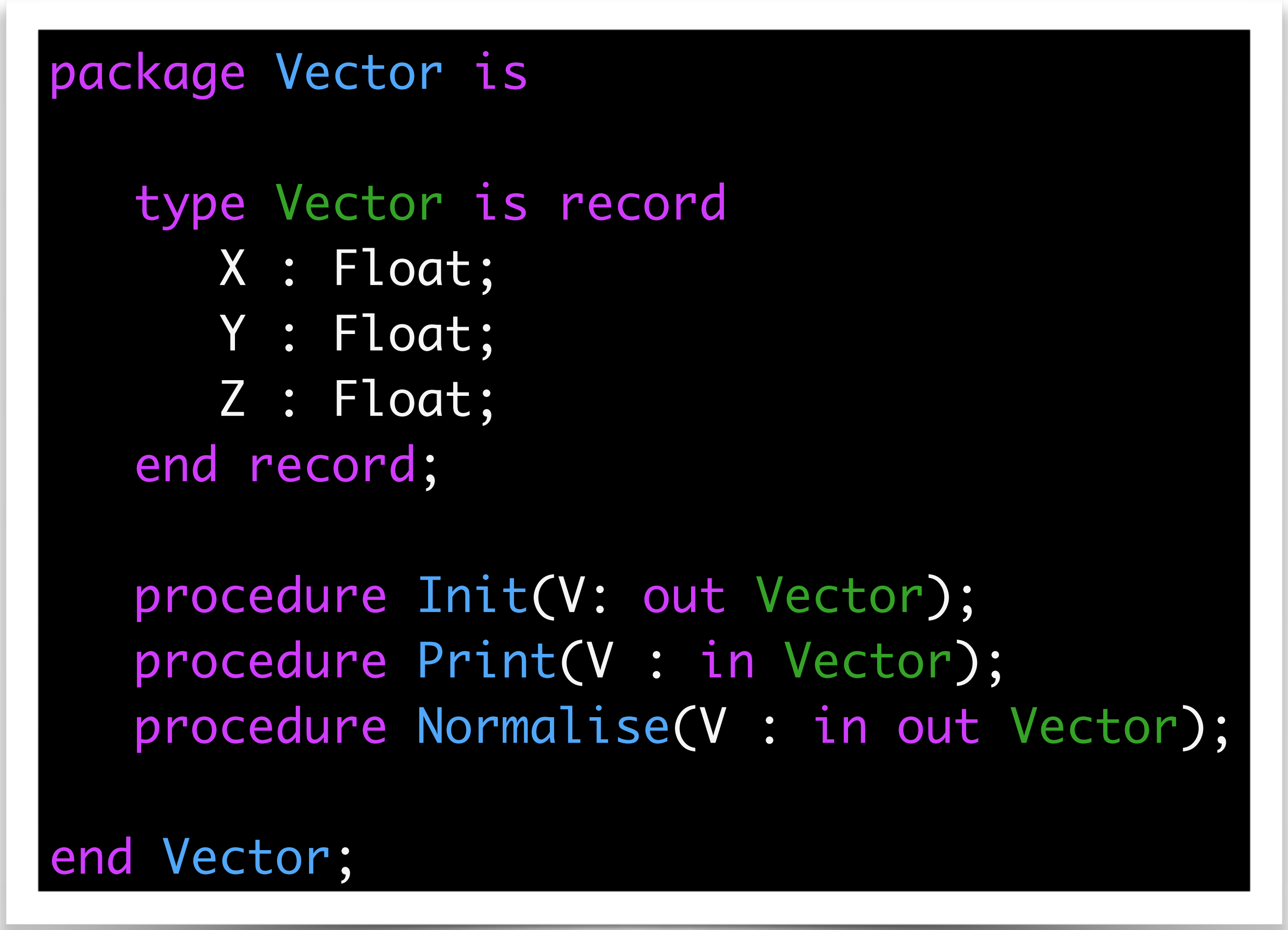
Parameter Modes



```
package Vector is

   type Vector is record
      X : Float;
      Y : Float;
      Z : Float;
   end record;

   procedure Init(V: out Vector);
   procedure Print(V : in Vector);
   procedure Normalise(V : in out Vector);

end Vector;
```

in: read only
out: write only
in out: read/write

Parameter Modes

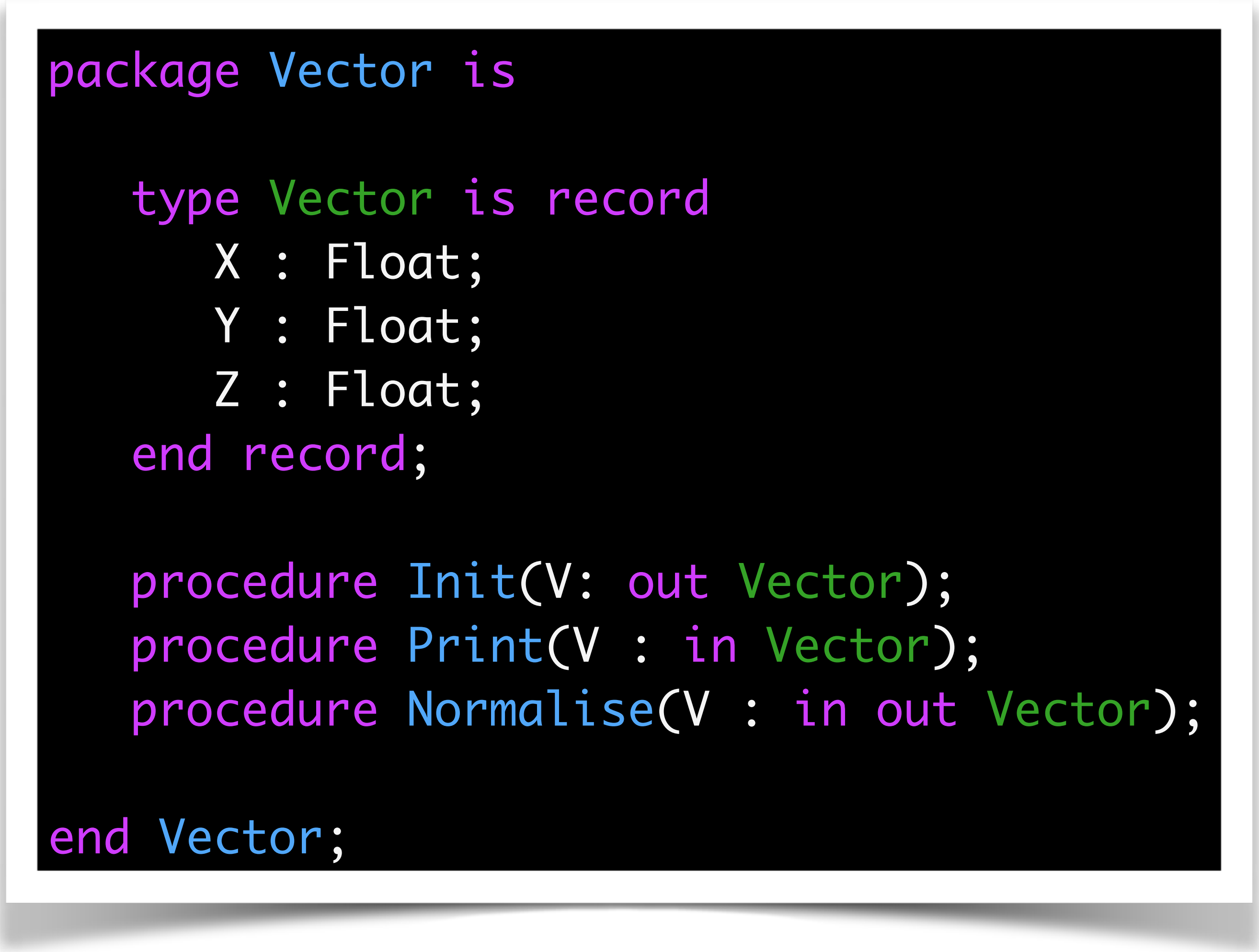


```
package Vector is

   type Vector is record
      X : Float;
      Y : Float;
      Z : Float;
   end record;

   procedure Init(V: out Vector);
   procedure Print(V : in Vector);
   procedure Normalise(V : in out Vector);

end Vector;
```

in: read only
out: write only
in out: read/write

Checked statically by the type system

Parameter Modes



```
package Vector is

   type Vector is record
      X : Float;
      Y : Float;
      Z : Float;
   end record;

   procedure Init(V: out Vector);
   procedure Print(V : in Vector);
   procedure Normalise(V : in out Vector);

end Vector;
```

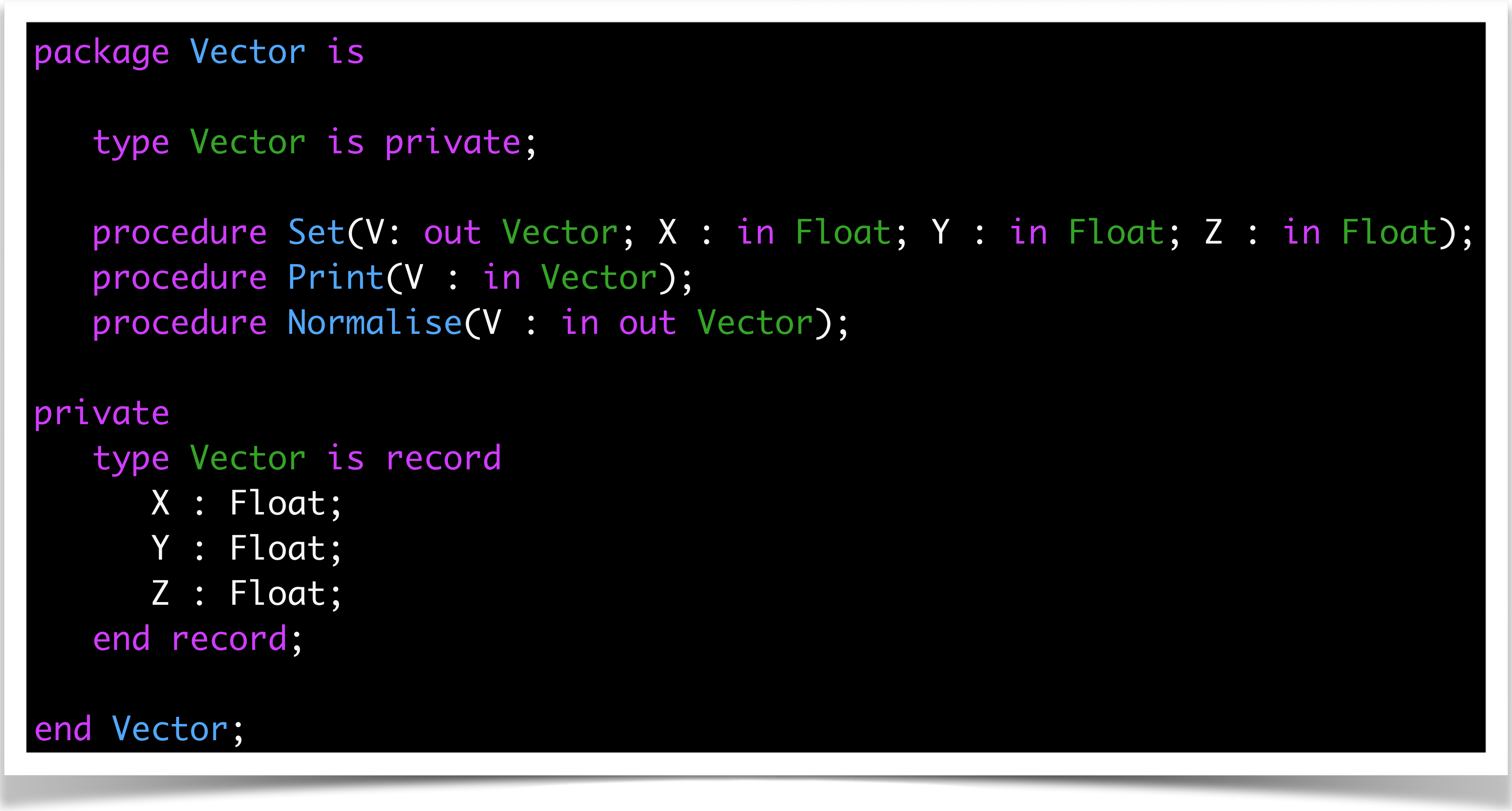
in: read only
out: write only
in out: read/write

Checked statically by the type system

Allows e.g. all records to be passed by pointer if the compiler desires.

Note: Strong typing *improving* performance over e.g. C

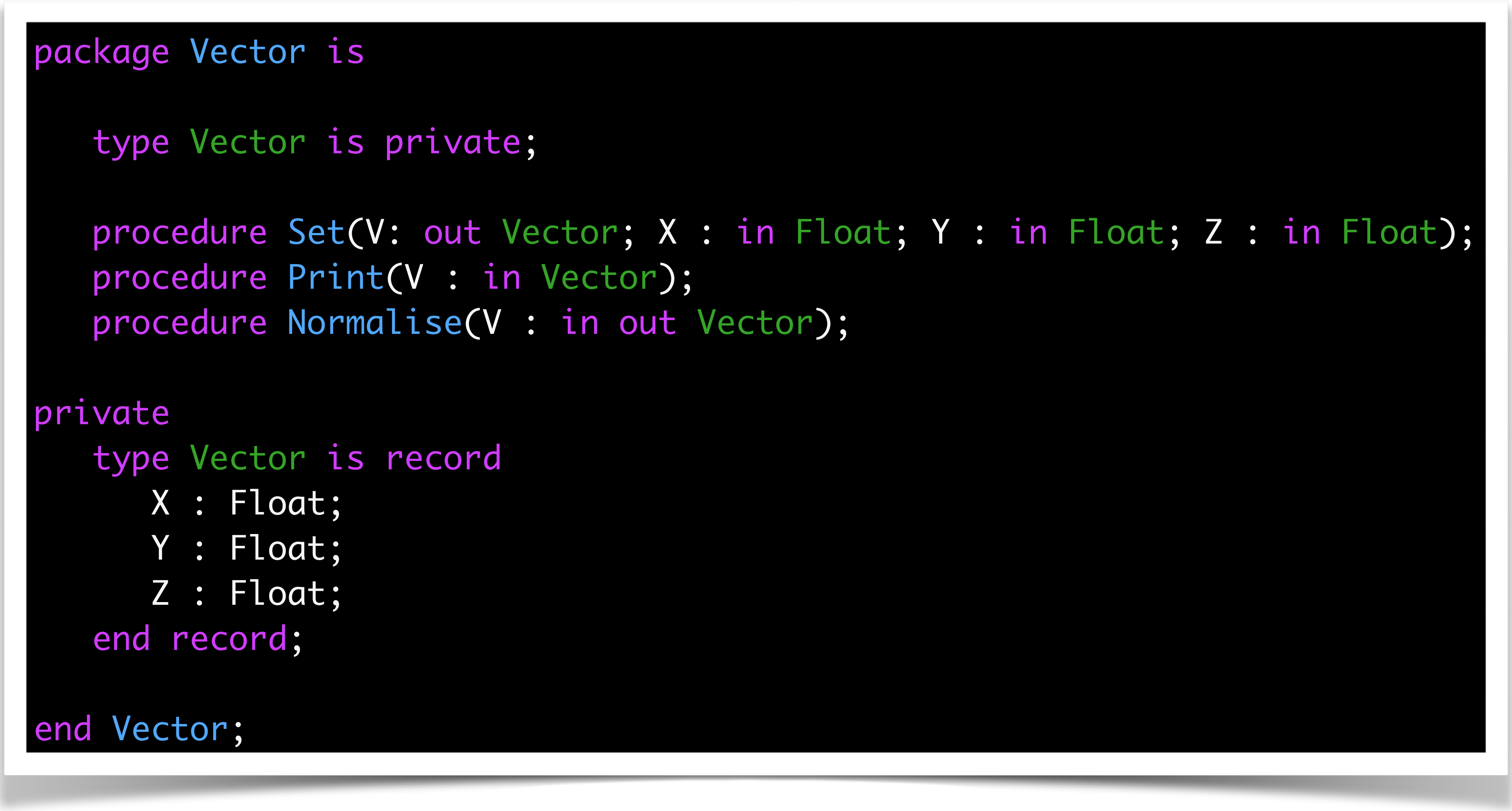
Implementation Hiding



```ada
package Vector is

   type Vector is private;

   procedure Set(V: out Vector; X : in Float; Y : in Float; Z : in Float);
   procedure Print(V : in Vector);
   procedure Normalise(V : in out Vector);

private
   type Vector is record
      X : Float;
      Y : Float;
      Z : Float;
   end record;

end Vector;
```

Vector’s fields can’t be accessed outside the package.

Private

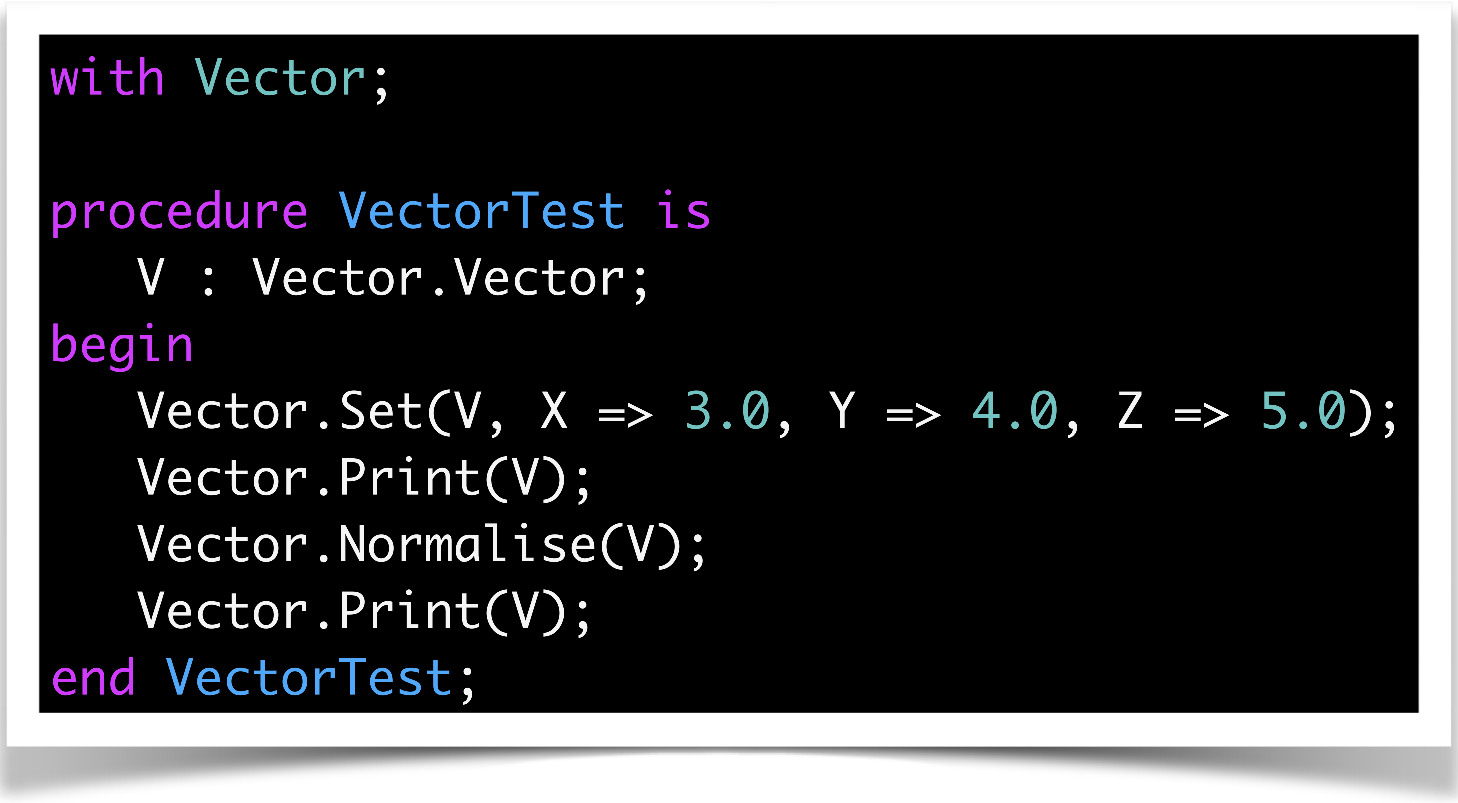


```ada
package Vector is

   type Vector is private;

   procedure Set(V: out Vector; X : in Float; Y : in Float; Z : in Float);
   procedure Print(V : in Vector);
   procedure Normalise(V : in out Vector);

private
   type Vector is record
      X : Float;
      Y : Float;
      Z : Float;
   end record;

end Vector;
```

Why does the **Vector** record need to be defined in the spec?

Private: A Hint

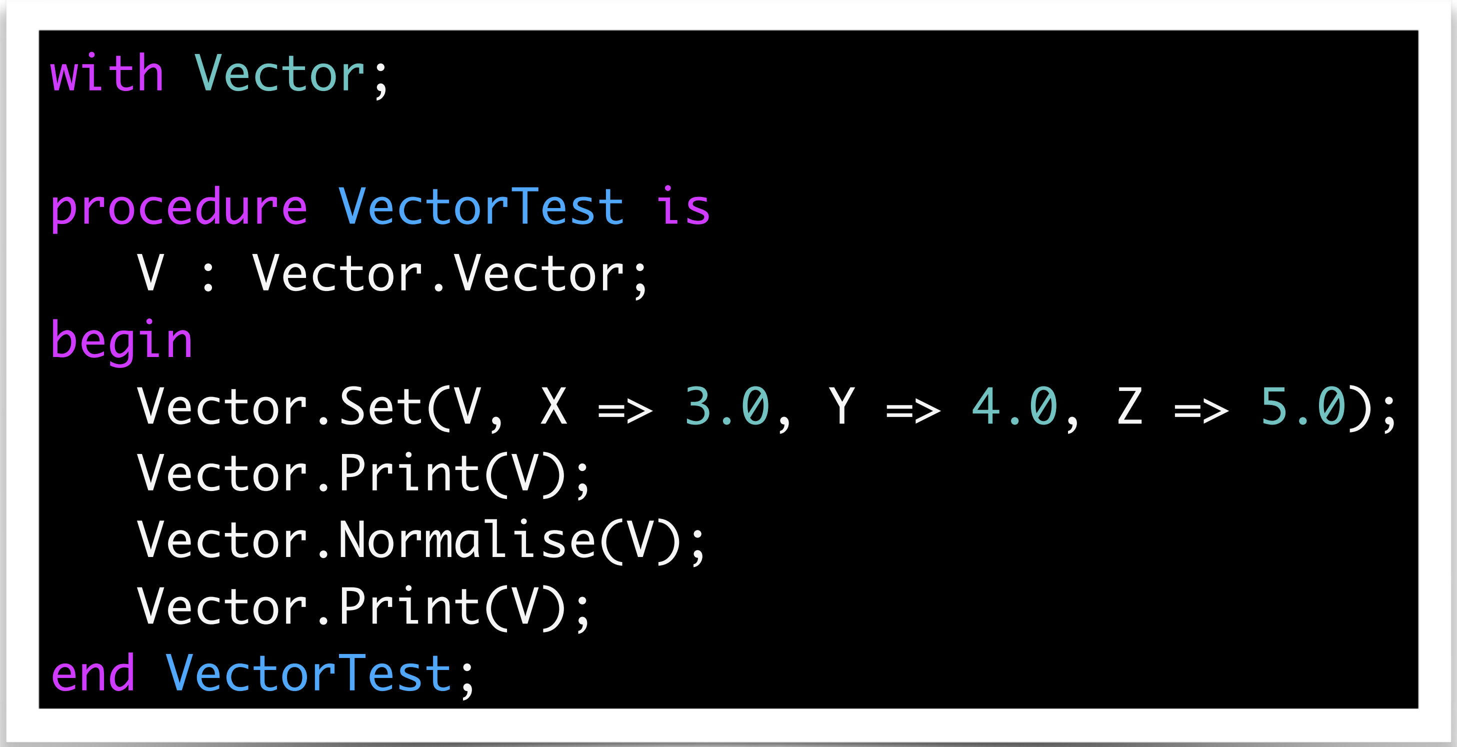
Why does the **Vector** record need to be defined in the spec?



```ada
with Vector;

procedure VectorTest is
   V : Vector.Vector;
begin
   Vector.Set(V, X => 3.0, Y => 4.0, Z => 5.0);
   Vector.Print(V);
   Vector.Normalise(V);
   Vector.Print(V);
end VectorTest;
```

Private: A Hint

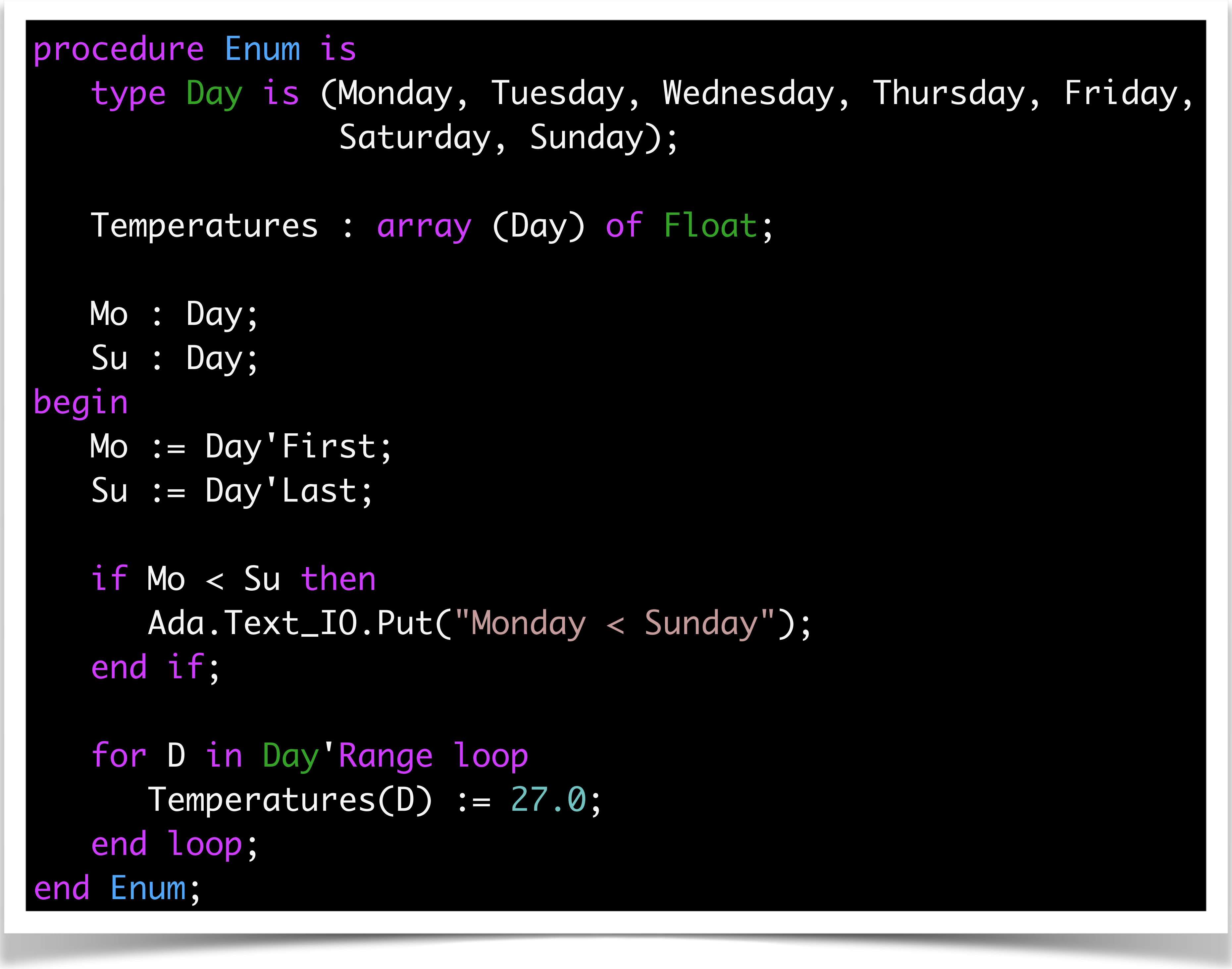
Why does the **Vector** record need to be defined in the spec?



```
with Vector;

procedure VectorTest is
   V : Vector.Vector;
begin
   Vector.Set(V, X => 3.0, Y => 4.0, Z => 5.0);
   Vector.Print(V);
   Vector.Normalise(V);
   Vector.Print(V);
end VectorTest;
```

Because the compiler needs to know how big a **Vector** is when it compiles client code that uses **Vector**s

Enumerations



```
procedure Enum is
   type Day is (Monday, Tuesday, Wednesday, Thursday, Friday,
                Saturday, Sunday);

   Temperatures : array (Day) of Float;

   Mo : Day;
   Su : Day;
begin
   Mo := Day'First;
   Su := Day'Last;

   if Mo < Su then
      Ada.Text_IO.Put("Monday < Sunday");
   end if;

   for D in Day'Range loop
      Temperatures(D) := 27.0;
   end loop;
end Enum;
```

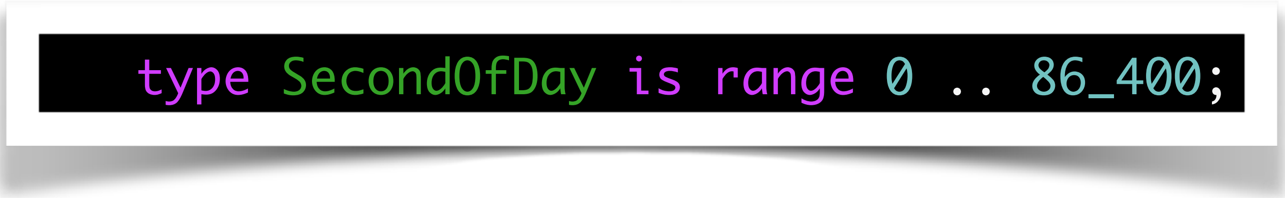
attributes

comparison

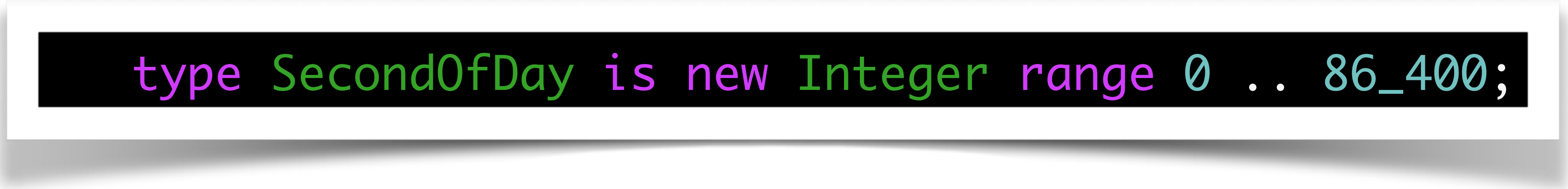
array
indexed by
enums

A Note on Standard Types

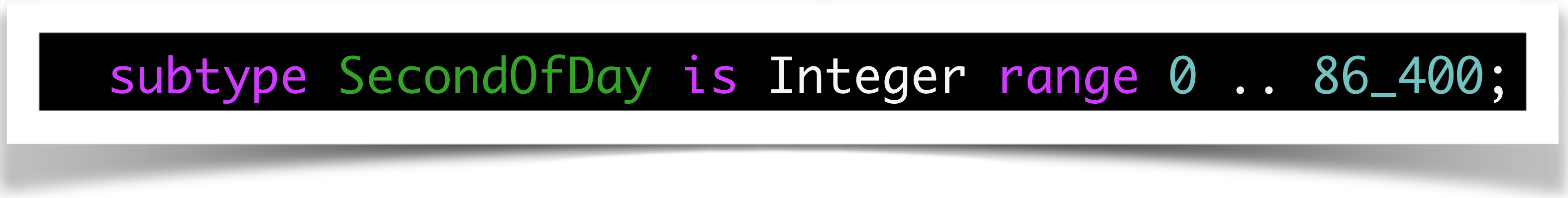
Which is best?



```
type SecondOfDay is range 0 .. 86_400;
```



```
type SecondOfDay is new Integer range 0 .. 86_400;
```

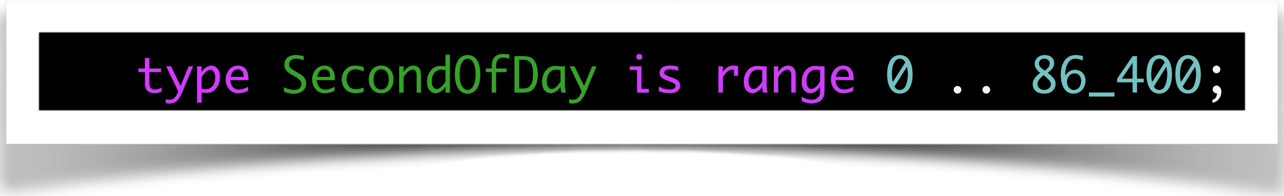


```
subtype SecondOfDay is Integer range 0 .. 86_400;
```

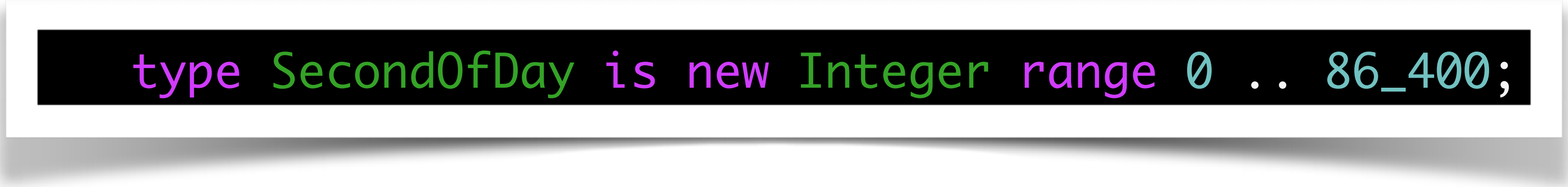
A Note on Standard Types

Which is best?

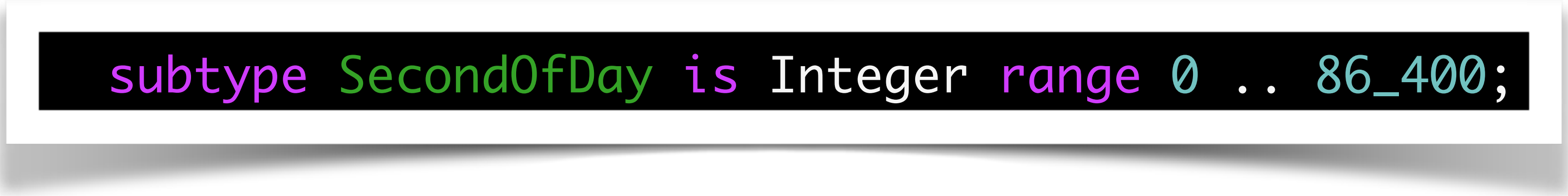
This one:



```
type SecondOfDay is range 0 .. 86_400;
```



```
type SecondOfDay is new Integer range 0 .. 86_400;
```

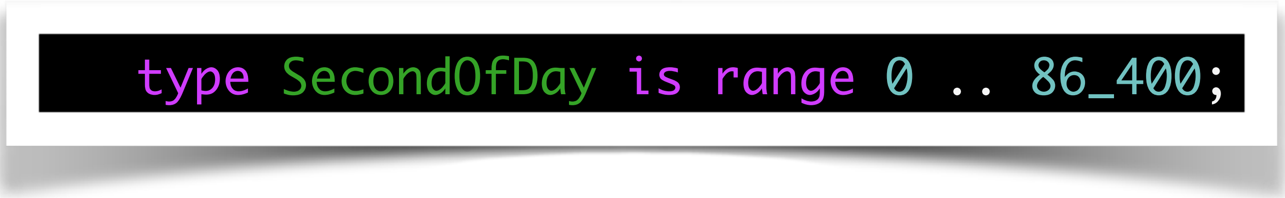


```
subtype SecondOfDay is Integer range 0 .. 86_400;
```

A Note on Standard Types

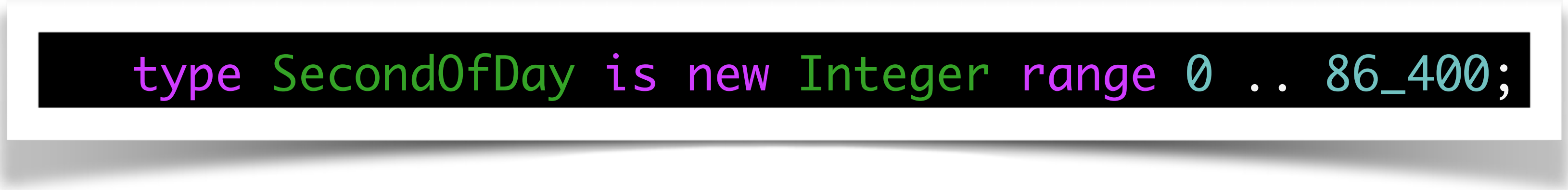
Which is best?

This one:

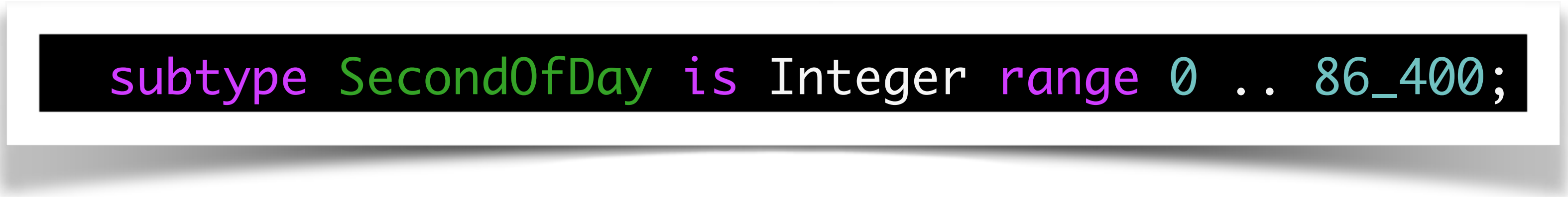


```
type SecondOfDay is range 0 .. 86_400;
```

More portable than this:



```
type SecondOfDay is new Integer range 0 .. 86_400;
```

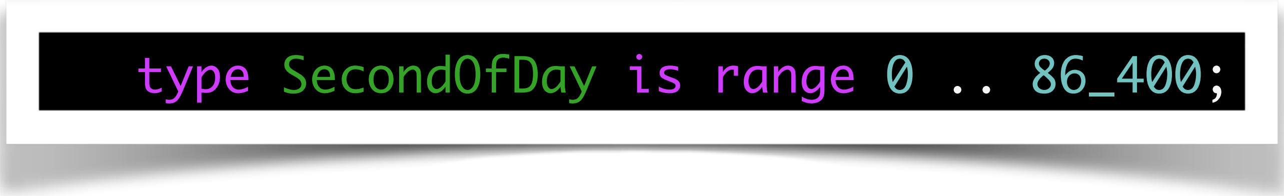


```
subtype SecondOfDay is Integer range 0 .. 86_400;
```

A Note on Standard Types

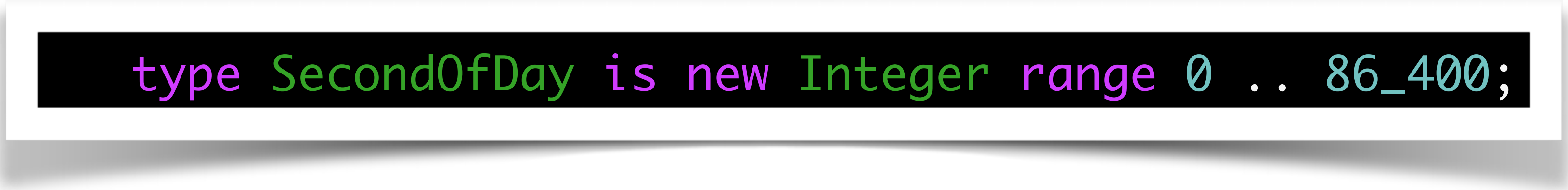
Which is best?

This one:



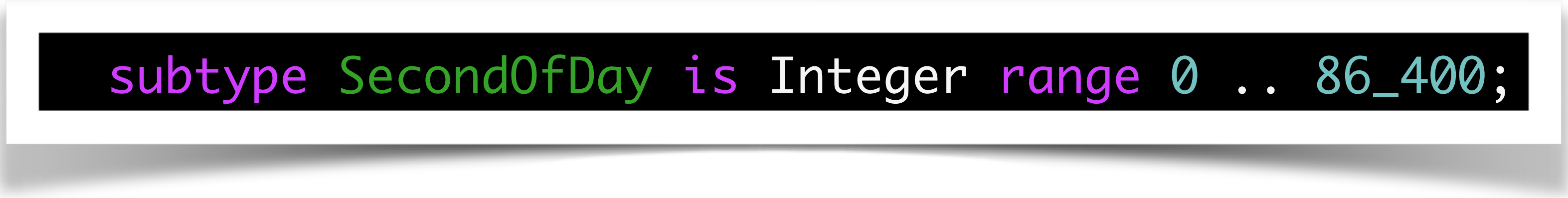
```
type SecondOfDay is range 0 .. 86_400;
```

More portable than this:



```
type SecondOfDay is new Integer range 0 .. 86_400;
```

More static checking than this:



```
subtype SecondOfDay is Integer range 0 .. 86_400;
```

Resources

Ada 2012 Reference Manual:
http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-TTL.html

(including what's in the standard library,
plus all the language features we don't cover:
dynamic memory, variant records,
OO, concurrency, etc.)

Ada Quality and Style Guide:
https://en.wikibooks.org/wiki/Ada_Style_Guide