



# Exceptions

[university.adacore.com](http://university.adacore.com)

# Exceptions in Ada

- **Ada puts a high emphasis on the program specification**
- **Some aspects of the specification can be statically checked (compiler errors) some other aspects are dynamically checked (run-time errors)**
- **Run-time errors lead to an exception being raised**
- **Additional checks can be written by the developer**
- **Exception can be explicitly raised by the specification**

## Caveat

- **Exceptions should be reserved to indicate exceptional behavior (after all that's why they're called exceptions)**
- **Exceptions are expensive to raise and propagate**
- **Exceptions lead to code that is difficult to analyze statically**
- **Too many exceptions complicate program debugging**
- **It is reasonable to expect that a correct program should not raise exceptions**

# The most common exception: Constraint\_Error

- **Constraint\_Error** is raised as soon as a constraint in the program is violated
  - Scalar type range, array index range, discriminant value, dereference of null pointer...

```
type My_Array is array (Integer range 1 .. 10) of Integer;  
  
V : My_Array;  
begin  
  
    V (11) = 0; -- exception raised at run-time
```

- This ensures that errors are found as soon as possible, and that code is protected from certain classes of invalid values
- In obvious cases, the compiler will warn of exceptions that will be raised unconditionally at run-time

# Protecting a piece of code against Exception

- Exceptions can be caught by a *handler* at the end of a block of statements

Ada	C++
<pre>begin   -- some code exception   when Constraint_Error =&gt;     -- some code end;</pre>	<pre>try {   // some code } catch (Constraint_Error e) {   // some code }</pre>

- Several exceptions can be handled by the same handler

```
begin
  -- some code
exception
  when Constraint_Error | Storage_Error =>
    -- some code
  when others =>
    -- code for all other exceptions
end;
```

# Exception handlers only protect the sequence of statements



- Exceptions that may occur in a declarative part can only be caught by the calling frame or the enclosing frame

```
P (0);
```

```
procedure P (V : Integer) is
  X : Integer := 1 / V;
begin
  null;
exception
  when others =>
    Put_Line ("Something went wrong");
end P;
```

**Nothing gets printed, call fails**

```
procedure P (V : Integer) is
begin
  My_Block:
  declare
    X : Integer := 1 / V;
  begin
    null;
  end My_Block;
exception
  when others =>
    Put_Line ("Something went wrong");
end P;
```

**Prints "Something went wrong"**  
**Call terminates normally**

# Exception Propagation

- When not caught by a given frame, the exception is *propagated*
- The enclosing frame or caller has a chance to catch an exception, or to propagate it. If the exception is not handled somewhere, the program terminates.

```
begin
  P (0);
exception
  when others =>
    Put_Line ("Call to P went wrong");
end;
```

**Prints "Call to P went wrong"**  
**Sequence continues normally**

```
procedure P (V : Integer) is
  X : Integer := 1 / V;
begin
  null;
exception
  when others =>
    Put_Line ("Something went wrong");
end P;
```

**Nothing gets printed, call fails**

# Exception Declaration and Raise

- **Ada exceptions are a specific kind of entity**
  - associated with a scope and obey visibility rules
  - declared like a constant

```
My_Exception : exception;
```

- **The runtime environment can raise predefined exceptions**
  - Constraint\_Error, Program\_Error, Storage\_Error, ...
- **Exceptions can be explicitly re-raised in a handler**

```
exception  
  when others =>  
    raise;  
end;
```



# Exception message

- **An exception can be raised explicitly, optionally associated with a message**
  - Ada doesn't provide support for arbitrary object propagation

```
raise My_Exception;  
raise My_Exception with "My message";
```

- **The exception message can be retrieved through an *exception occurrence*, using the services provided by Ada.Exceptions**

```
with Ada.Exceptions; use Ada.Exceptions;  
  
[...]  
  
exception  
  when E : others =>  
    Put_Line (Exception_Message (E));  
end;
```



# ? Quiz



## What's the output? (1/10)

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

procedure P is
  type Positive is range 0 .. 10;
  V : Positive := 10;
begin
  V := V + 1;
  Put_Line (Positive'Image (V));
end P;
```



## What's the output? (1/10)

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

procedure P is
  type Positive is range 0 .. 10;
  V : Positive := 10;
begin
  V := V + 1;
  Put_Line (Positive'Image (V));
end P;
```

**Nothing, program stops on an exception.**



## What's the output? (2/10)

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

procedure P is
  type Positive is range 0 .. 10;
  V : Positive := 10;
begin
  V := V + 1;
  Put_Line (Positive'Image (V));
exception
  when Constraint_Error =>
    Put_Line ("CE");
end P;
```



## What's the output? (2/10)

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

procedure P is
  type Positive is range 0 .. 10;
  V : Positive := 10;
begin
  V := V + 1;
  Put_Line (Positive'Image (V));
exception
  when Constraint_Error =>
    Put_Line ("CE");
end P;
```

**CE, the exception  
Constraint\_Error is caught**



## What's the output? (3/10)

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

procedure E is
begin
  My_Block:
  declare
    A : Positive;
  begin
    A := -5;
  exception
    when Constraint_Error =>
      Put_Line ("caught it");
  end My_Block;
exception
  when others =>
    Put_Line ("last chance handler");
end E;
```



## What's the output? (3/10)

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

procedure E is
begin
  My_Block:
  declare
    A : Positive;
  begin
    A := -5;
  exception
    when Constraint_Error =>
      Put_Line ("caught it");
  end My_Block;
exception
  when others =>
    Put_Line ("last chance handler");
end E;
```

**"caught it", since the exception  
happens in the body of My\_Block**





## What's the output? (4/10)

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

procedure E is
begin
  My_Block:
  declare
    A : Positive;
  begin
    A := -5;
  exception
    when Constraint_Error =>
      Put_Line ("caught it");
      raise;
  end My_Block;
exception
  when others =>
    Put_Line ("last chance handler");
end E;
```



## What's the output? (4/10)

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

procedure E is
begin
  My_Block:
  declare
    A : Positive;
  begin
    A := -5;
  exception
    when Constraint_Error =>
      Put_Line ("caught it");
      raise;
  end My_Block;
exception
  when others =>
    Put_Line ("last chance handler");
end E;
```

"caught it" and "last chance handler"



## What's the output? (5/10)

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

procedure E is
begin
  My_Block:
  declare
    A : Positive := -1;
  begin
    A := -5;
  exception
    when Constraint_Error =>
      Put_Line ("caught it");
  end My_Block;
exception
  when others =>
    Put_Line ("last chance handler");
end E;
```



## What's the output? (5/10)

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

procedure E is
begin
  My_Block:
  declare
    A : Positive := -1;
  begin
    A := -5;
  exception
    when Constraint_Error =>
      Put_Line ("caught it");
  end My_Block;
exception
  when others =>
    Put_Line ("last chance handler");
end E;
```

"last chance handler" will be printed out from the handler declared in E as the exception occurs in the declarative section of My\_Block and not the body.



## What's the output? (6/10)

```
declare
  A, B : Integer;
begin
  A := 0;
  B := 5;

  if ((A /= 0) and ((B / A) = 0)) then
    Put_Line ("A");
  else
    Put_Line ("Division By Zero");
  end if;

exception
  when others =>
    Put_Line ("Exception!");
end;
```



## What's the output? (6/10)

```
declare
  A, B : Integer;
begin
  A := 0;
  B := 5;

  if ((A /= 0) and ((B / A) = 0)) then
    Put_Line ("A");
  else
    Put_Line ("Division By Zero");
  end if;

exception
  when others =>
    Put_Line ("Exception!");
end;
```

**"Exception!", the right operand  
of the and is always evaluated**



# Is this correct?

(7/10)



YES

(click on the check icon)

NO

(click on the error location(s))

```
package P is  
  
    procedure Call;  
  
end P;
```

```
package body P is  
  
    My_Exception : exception;  
  
    procedure Call is  
    begin  
        raise My_Exception;  
    end Call;  
  
end P;
```

```
with P; use P;  
  
procedure Main is  
begin  
    Call;  
exception  
    when My_Exception =>  
        Put_Line ("EXC");  
end Main;
```



Is this correct?

(7/10)



NO

```
package P is  
  
    procedure Call;  
  
end P;
```

```
package body P is  
  
    My_Exception : exception;  
  
    procedure Call is  
    begin  
        raise My_Exception;  
    end Call;  
  
end P;
```

```
with P; use P;  
  
procedure Main is  
begin  
    Call;  
exception  
    when My_Exception =>  
        Put_Line ("EXC");  
end Main;
```



**My\_Exception is not visible here, declared in the body of P. Could be handled in a "when others".**





# Is this correct?

(8/10)



YES

(click on the check icon)

NO

(click on the error location(s))

```
with P; use P;

procedure Main is
  C : Integer := 0;
begin
  while C < 10 loop
    Nested_Block:
    begin
      Call;
      C := C + 1;
    exception
      when others =>
        null;
    end Nested_Block;
  end loop;
end Main;
```

```
package P is

  procedure Call;

end P;
```

```
package body P is

  My_Exception : exception;

  procedure Call is
  begin
    raise My_Exception;
  end Call;

end P;
```




# Is this correct?

(8/10)



NO



```
with P; use P;

procedure Main is
  C : Integer := 0;
begin
  while C < 10 loop
    Nested_Block:
    begin
      Call;
      C := C + 1;
      exception
        when others =>
          null;
    end Nested_Block;
  end loop;
end Main;
```

```
package P is

  procedure Call;

end P;
```

```
package body P is

  My_Exception : exception;

  procedure Call is
  begin
    raise My_Exception;
  end Call;

end P;
```

Infinite loop here, as the exception jumps over the incrementing



# Is this correct?

(9/10)



YES

(click on the check icon)

NO

(click on the error location(s))

```
package P is
```

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

```
    function Safe_Get (Arr : A_Type ; V : Integer) return Integer;
```

```
end P;
```

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

```
package body P is
```

```
    function Safe_Get (Arr : A_Type ; V : Integer) return Integer is  
    begin
```

```
        return Arr (V);
```

```
    exception
```

```
        when Constraint_Error =>
```

```
            Put_Line ("Wrong Index");
```

```
    end Safe_Get;
```

```
end P;
```



# Is this correct?

(9/10)



NO

```
package P is

  type A_Type is array (Integer range <>) of Integer;

  function Safe_Get (Arr : A_Type ; V : Integer) return Integer;

end P;
```

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

package body P is

  function Safe_Get (Arr : A_Type ; V : Integer) return Integer is
  begin
    return Arr (V);
  exception
    when Constraint_Error =>
      Put_Line ("Wrong Index");
  end Safe_Get;

end P;
```



This exception handler is finishing a function, but has no return statement



# Is this correct?

## (10/10)



**YES**

(click on the check icon)

**NO**

(click on the error location(s))

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

procedure Main is

  type R is record
    F : Positive := -1;
  end record;

  V : R := (F => 1);

begin
  Put_Line (Positive'Image (V.F));
end Main;
```



Is this correct?

(10/10)



YES

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

procedure Main is

  type R is record
    F : Positive := -1;
  end record;

  V : R := (F => 1);

begin
  Put_Line (Positive'Image (V.F));
end Main;
```

**The explicit initialisation overrides the default invalid value, this is OK**



[university.adacore.com](http://university.adacore.com)