The goal of an engineer is to retire without having caused any major catastrophe

- Dilbert
Today

- Program robustness
- Exception handling

November 2, 1988 Internet Worm
- A self-replicating program was released upon the Internet
- This program (a worm) invaded VAX and Sun computers running versions of Berkeley UNIX, and used their resources to attack still more computers.
- Within the space of hours this program had spread across the U.S., infecting thousands of computers and making many of them unusable due to the burden of its activity.
- **Cause:** undetected buffer overflow in C routine `gets()`
• 1986: Therac 25 radiation machine kills several patients
  – **Cause**: poor testing of the software

• June 4, 1996: 1st flight of Ariane 5 aborted: Ariane 5 destroyed
  – **Cause**: Code from Ariane 4 guidance system was reused in Ariane 5 but not tested.

• September 23 1999: Mars Orbiter stops communicating with NASA
  – **Cause**: Approach orbit angle was incorrect because of inconsistency between units of measurement

---

**Errors**

• No programmer is perfect
  – The good ones handle errors gracefully

• Errors
  – Compile time
  – Link-time

• Run-time errors
  – Program errors
  – **User errors**
  – **Exceptions**
User Errors

- User provides invalid input
  - types in name of file that does not exist
  - provides program argument with value outside legal bounds

- Detect using “if” checks in program
  - Program should print message and recover gracefully
  - Possibly ask user for new input

Exceptions

- Rare errors “exceptional” from which recovery may be possible
  - User hits interrupt key
  - Arithmetic overflow

- Detected by hardware or operating system
  - Program can handle them using exception handlers
  - Not usually possible/practical to detect with conditional checks
Robustness

- Your program should never terminate without either
  - Completing successfully
  - Sending a meaningful error message

- Approaches to achieve Robustness
  - Debug
  - Defensive programming
    - Conditional checks
    - Assertions
  - Exception handling

Finding Errors

- Try to “break” the program
  - What can go wrong?
  - What happens if it does?
  - Sometimes nothing needs to be done.
  - If that is a problem, how can we detect it?
  - What can we do about it?
    - Tell the user
    - Die gracefully
    - Recover reasonably
Ada’s Classification of Errors 1.1.5

1. Errors that are **required** to be detected **prior** to run time by every Ada implementation

2. Errors that are **required** to be detected **at** run time by the execution of an Ada program

3. Bounded errors

4. Erroneous execution

Exceptions – Ada Perspective

- An **exception** represents a kind of exceptional situation
  - An occurrence of such a situation (at run time) is called: **exception occurrence**

- To **raise** an exception is to abandon normal program execution

- Performing some actions in response to the arising of an exception is called **handling** the exception
Example

with Ada.Text_Io, Ada.Integer_Text_Io;
use Ada.Text_Io, Ada.Integer_Text_Io;

procedure Main is
    subtype Numrange is Integer range 1..10;
    Num : Numrange;
begin --main
    Put ("please enter an integer from 1 to 10: ");
    Get(Num); Skip_Line;
end;

Exception Declaration

• An exception_declaration declares a name for an exception

• User-defined exceptions:
  - Overflow, Underflow : exception;
  - Error : exception;

• Predefined language exceptions:
  - Constraint_Error, Program_Error,
    Storage_Error, and Tasking_Error
Exception Handlers

• The response to one or more exceptions is specified by an `exception_handler`

```
subprogram specification
declarations
begin
    statements
exception
    one or more exception handlers
end;
```

Exception Handling

• Operation:
  − When exception occurs, control jumps to the handler for that exception
  − When handler statements finish, subprogram terminates
  − Control **never** returns to point where exception occurred
  − If no handler, subprogram terminates and exception is passed back to its caller
    • Keep doing this until `main` reached with no handler (program crashes)
    • Or suitable handler found

```
subprogram specification
declarations
begin
    statements
exception
    one or more exception handlers
end;
```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Open_File is
  Filename : String (1 .. 30);
  Namelen : Natural;
  The_File : File_Type;

begin
  Put ("What file do you want to read? ");
  Get_Line (Filename, Namelen);
  Open (File => The_File,
        Name => Filename (1 .. Namelen),
        Mode => In_File);

exception
  when Status_Error =>
    Put_Line ("The file is already open");
  when Name_Error =>
    Put_Line ("There is no file with that name");
  when Use_Error =>
    Put_Line ("The file cannot be read");
  when others =>
    Put_Line ("Unexpected error on opening file");

end Open_File;

Raise Statements

• A raise_statement raises an exception

  - raise exception_name;

  - raise; -- re-raise the current exception
Block statement

- You can define your own block at any point in an Ada program.
- Its structure is similar to a subprogram:
  ```ada
  declare
    declarations
  begin
    normal sequence of statements
  exception
    exception handlers
  end;
  ```

Declare block for local variables

```
procedure main is
  x, y : integer;
begin
  statements;

  -- time to swap two variables
  declare
    temp : integer;
  begin
    temp := x;
    x := y;
    y := temp;
  end;

  more statements
end;
```
Exception in block statements

- The other reason for defining a block statement is to **enable local exception handling** (especially in a loop).
- Operation:
  - when an exception occurs:
    - execution transfers straight to its exception handler
    - appropriate exception handler is executed
    - execution of the whole block statement terminates
    - execution continues with statement after the block
  - if no local exception handler:
    - block terminates immediately
    - control passes to outer block, to see if it has an appropriate exception handler
    - etc.

Example program

```ada
--Safe I/O
with Ada.Text_Io; use Ada.Text_Io;

procedure Ex2 is
  type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  package Day_Io is new Enumeration_Io (Days); use Day_Io;
  Local_Day : Days; --entered by user
  Good_Day : Boolean := False; --loop control
begin
  while not Good_Day loop
    begin
      Put ("Enter a day name (first 3 letters) : ");
      Get (Local_Day);
      -- this point is only reached if valid entry given
      Good_Day := True;
      -- ...
    exception -- exception handler for while block
      when Data_Error =>
        Put ("Must be first 3 letters of a day name");
        New_Line; Skip_Line;
    end;
  end loop;
end Ex2;
```
Strategies for handling exceptions

- Three levels of ambition:
  1. Take control
     - try to act so program can continue
  2. Identify exception for handling elsewhere
     - detect, identify, pass it on
  3. Ignore
     - program halts (crashes)

1. Take control

- **Example**: \( \tan(x) \) may be impossible to compute or represent
- Constraint\_Error exception can be detected and handled
- Handle the exception locally; caller never realizes anything was wrong.

```pascal
function Tan (X : Float )
return Float is
begin
  return Sin(X) / Cos(X);
exception
  when Constraint_Error =>
    if (Sin(X)>=0.0 and Cos(X)>= 0.0) or (Sin(X)< 0.0 and Cos(X)< 0.0)
      then
        return Float'Last;
      else
        return -Float'Last;
      end if;
end Tan;
```
2. Pass exception back

- **raise** statement in exception handler:
  - perhaps take some action locally
  - identify exception and pass it back to caller

```pascal
function Tan (X : Float )
  return Float is
begin
  return Sin(X) / Cos(X);
exception
  when Constraint_Error =>
    Put_Line ("The value of tangent is too big");
    raise;
end Tan;
```

3. Ignore the exception

- No example is needed of the third level of ambition. We are all familiar with that one!

  It is (probably) what we all have been doing all the time up to now ...
Exceptions in Input/Output A.13

- TEXT_IO defines several exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_error</td>
<td>invalid data type, data has wrong form</td>
</tr>
<tr>
<td>status_error</td>
<td>try to open an already open file</td>
</tr>
<tr>
<td>mode_error</td>
<td>try to read from an output file</td>
</tr>
<tr>
<td>name_error</td>
<td>no such file</td>
</tr>
<tr>
<td>use_error</td>
<td>try to open printer for reading</td>
</tr>
<tr>
<td>end_error</td>
<td>EOF encountered while reading</td>
</tr>
<tr>
<td>layout_error</td>
<td>SET_COL beyond LINELENGTH limit</td>
</tr>
<tr>
<td>device_error</td>
<td>hardware failure</td>
</tr>
</tbody>
</table>

Reading enumeration values

--Safe I/O (again)

```pascal
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
function Get_Day return Days is
    Local_Day : Days;  -- entered by user
    Good_Day : Boolean := False;  -- loop control
begin
    while not Good_Day loop
        begin -- while block
            Put ("Enter a day name (first 3 letters) : ");
            Get (Local_Day);
            -- this point only reached if valid entry given
            Good_Day := True;
        exception -- exception handler for while block
            when Data_Error =>
                Put ("Must be first 3 letters of a day name");
                New_Line;  Skip_Line;
        end; -- while block
    end loop;
    Skip_Line;
    return Local_Day;
end Get_Day;
```
Reading enumeration values, with range checks

```
type Week_Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
subtype Work_Days is Week_Days range Mon .. Fri;
subtype Weekend_Days is Week_Days range Sat .. Sun;

procedure Safe_Get_Day (Out_Day : out Week_Days;
                         Min : in  Week_Days := Work_Days'First;
                         Max : in  Week_Days := Work_Days'Last ) is

  Local_Day : Week_Days;
  Good_Day : Boolean := False;

begin
  Good_Day loop
  begin
    Put("Enter an day between ");
    Put( Min ); Put(" and "); Put( Max ); Put(" ");
    Get( Local_Day );
    if (Local_Day < Min) or (Local_Day > Max) then
      raise Data_Error;
    else
      Good_Day := True;
    end if;
  end loop
  -- this point is reached only when input is a day code
  exception
    when Data_Error =>
      Put_Line("Invalid day!. Good days are ");
      for This_Day in Week_Days range Min .. Max loop
        Put( This_Day ); Put(" ");
      end loop;
      New_Line; Skip_Line; -- tidy up terminal
    end loop;
    -- this point can only be reached when valid value input
    Skip_Line; -- tidy up terminal handling
  end Safe_Get_Day;
```
procedure Gen_Float_Input (  
  Out_Float : out Float;  
  Min, Max : in Float ) is  
  Local_Float : Float;  
  Good_One : Boolean := False;  
begin  
  -- gen_float_input  
  while not Good_One loop  
  begin  
  -- protected block of code  
    Put("Enter a float in range ");  
    Put( Min, Exp => 0 ); Put( " to ");  
    Put( Max, Exp => 0 ); Put( " ");  
    Get( Local_Float );  
    -- this point can only be reached if the get  
    -- did not raise the exception  
    -- now tested against limits specified  
    Good_One:=((Local_Float>=Min) and (Local_Float<=Max));  
    if not Good_One then  
      raise Data_Error;  
    end if;  
  end loop;  
  -- this point can only be reached when valid value  
  -- tidy up terminal handling  
  Skip_Line;  
  Out_Float := Local_Float;  
end Gen_Float_Input;
Opening a file

-- safe file opening

procedure Open_File (The_File : in out File_Type ) is
  Filename : String (1 .. 30);
  Namelen : Natural;
begin
  Put ("What file do you want to read? ");
  Get_Line (Filename, Namelen);
  Open (File => The_File, Name => Filename (1 .. Namelen),
        Mode => In_File);

exception
  when Status_Error =>
    Put_Line ("The file is already open");
  when Name_Error =>
    Put_Line ("There is no file with that name");
  when Use_Error =>
    Put_Line ("The file cannot be read");
  when others =>
    Put_Line ("Unexpected error on opening file");
end Open_File;

User defined exceptions 1(4)

• You can declare your own exception types

  Tan_Error : exception;

Example:

procedure Main is
  X, Res   : Float;
  Tan_Error : exception;
function Tan (X : Float )
  return Float is
begin
  return Sin(X)/Cos(X);
exception
  when Numeric_Error =>
    raise Tan_Error;
end;

begin
  Put ("Enter A Real Number X: "); Get (X);
  Res := Tan(X);
  Put ("Tan(X) is "); Put(Res); New_Line;

exception
  when Tan_Error =>
    Put_Line ("The Tangent is Too Big");
end;
Declaring exceptions in packages

- NUMERIC_ERROR may arise in \( \tan(x) \) function.

- Perhaps too unilateral to handle it locally, so prefer to pass an exception back to the caller.

- What to pass back?
  - NUMERIC_ERROR is too general
  - more specific name desirable (eg TAN_ERROR)

- Where to declare TAN_ERROR?
  - not in function TAN (invisible outside)
  - not in calling code (belongs with TAN)
  - best is in a package, along with TAN

---

Example

```--package specification
package TRIGONOMETRY is
  function SIN (X : FLOAT) return FLOAT;
  function COS (X : FLOAT) return FLOAT;
  function TAN (X : FLOAT) return FLOAT;
  TAN_ERROR : exception;
end TRIGONOMETRY;

--package body
package body TRIGONOMETRY is
  function SIN (X : FLOAT) return FLOAT is
    begin ... end SIN;
  function COS (X : FLOAT) return FLOAT is
    begin ... end COS;
  function TAN (X : FLOAT) return FLOAT is
    begin return SIN(X) / COS(X);
    exception
      when NUMERIC_ERROR => raise TAN_ERROR;
    end TAN;
end TRIGONOMETRY;
```

Shows how a package specification can define an exception; the package body can raise that exception when appropriate; and a user program can recognize and handle the exception.
Example

--user program
with Ada.TEXT_IO, TRIGONOMETRY;
procedure compute_tan is

    number, res : FLOAT;

    begin -- compute_tan
        loop
            begin
                PUT ("Enter a real number");
                exit when END_OF_FILE;
                GET (number); SKIP_LINE;
                res := tan(number);
                PUT("Tangent is "); PUT(res); NEW_LINE;
            exception
                when TAN_ERROR =>
                    PUT_LINE ("Tangent is too big");
            end;
        end loop;
    end compute_tan;