Exceptions and Exception Handling

- Goal:
  - To illustrate the various models of exception handling and to show how exception handling can be used as a framework for implementing fault-tolerant systems

- Structure:
  - Exception handling in older real-time languages
  - Modern exception handling
  - Exception handling in Ada, Java and C
  - Recovery blocks and exceptions
  - Summary

Introduction

- There are a number of general requirements for an exception handling facility:
  - R1: The facility must be simple to understand and use
  - R2: The code for exception handling should not obscure understanding of the program's normal error-free operation
  - R3: The mechanism should be designed so that run-time overheads are incurred only when handling an exception
  - R4: The mechanism should allow the uniform treatment of exceptions detected both by the environment and by the program
  - R5: the exception mechanism should allow recovery actions to be programmed

EH in Older Real-time Languages

- Unusual return value or error return from a procedure or a function.
- C supports this approach

```
if(function_call(parameters) == AN_ERROR) {
        -- error handling code
} else {
   -- normal return code
};
```

- Meets the simplicity requirement R1 and allows recovery actions to be programmed (R5)
- Fails to satisfy R2, R3 and R4; the code is obtrusive, it entails overheads every time it is used, and it is not clear how to handle errors detected by the environment

EH: Forced Branch

- Used mainly in assembly languages,
  - the typical mechanism is for subroutines to skip return
  - the instruction following the subroutine call is skipped to indicate the presence/absence of an error
  - achieved by incrementing its return address (program counter) by the length of a simple jump instruction
  - where more than one exceptional return is possible, the PC can be manipulated accordingly

```
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
# normal processing
```

- Approach incurs little overhead(R3) and enables recovery actions to be programmed(R5). It can lead to obscure program structures and, therefore, violates requirements R1 and R2. R4 also cannot be satisfied

EH: Non-local Goto

- A high-level language version of a forced branch which uses label variables; e.g. RTL/2 - a non-local goto

```
svc data rrerr label erl; %a label variable % enddata
proc WhereErrorIsDetected();
  ...
  goto erl;
  ...
endproc;
proc Caller();
  ...
  WhereErrorIsDetected();
  ...
endproc;
proc main();
  ...
restart:
  ...
  erl := restart;
  ...
  Caller();
  ...
end proc;
```

EH: Non-local Goto

- When used in this way, the goto is more than just a jump; it implies an abnormal return from a procedure
- The stack must be unwound, until the environment restored is that of the procedure containing the declaration of the label
- The penalty of unwinding the stack is only incurred when an error has occurred so requirement R3 has been satisfied
- Although the use of gotos is very flexible (satisfying R4 and R5), they can lead to very obscure programs; they fail to satisfy the requirements R1 and R2

Procedure Variables

- With goto, the control flow of the program has been broken
- In RTL/2, the error label is generally used for unrecoverable errors and an error procedure variable used when control should be returned to the point where the error originated

```
svc data rrerr;
   label erl;
  proc(int) erp; % erp is a procedure variable %
enddata;
```

```
proc recover(int);
  ...
endproc;

proc WhereErrorIsDetected();
  ...
  if recoverable then erp(n)
  else goto erl end;
 ...
endproc;

proc Caller();
  ...
  WhereErrorIsDetected();
  ...
endproc;

proc main();
  ...
  erl := fail;
  erp := recover;
  ...
  Caller();
  ...
fail:
  ...
end proc
```

Programs can become very difficult to understand and maintain

Recently, languages like C++ provide default functions (within the context of language-level exception handling) which are called when no handler for an exception can be found

These default functions can be re-defined by the programmer.

Exceptions and their Representation

- Environmental detection and application error detection
- A synchronous exception is raised as an immediate result of a process attempting an inappropriate operation
- An asynchronous exception is raised some time after the operation causing the error; it may be raised in the process which executed the operation or in another process
- Asynchronous exceptions are often called asynchronous notifications or signals and will be considered later

Classes of Exceptions

- Detected by the environment and raised synchronously; e.g. array bounds error or divide by zero
- Detected by the application and raised synchronously, e.g. the failure of a program-defined assertion check
- Detected by the environment and raised asynchronously; e.g. an exception raised due to the failure of some health monitoring mechanism
- Detected by the application and raised asynchronously; e.g. one process may recognise that an error condition has occurred which will result in another process not meeting its deadline or not terminating correctly

Synchronous Exceptions

- There are two models for their declaration.
  - a constant name which needs to be explicitly declared, e.g. Ada
  - an object of a particular type which may or may not need to be explicitly declared; e.g. Java
- Ada: e.g., the exceptions that can be raised by the Ada RTS are declared in package Standard:

```
package Standard is
        ...
        Constraint_Error : exception;
        Program_Error : exception;
        Storage_Error : exception;
        Tasking_Error : exception;
        ...
end Standard;
```

- This package is visible to all Ada programs.

The Domain of an Exception Handler

- Within a program, there may be several handlers for a particular exception
- Associated with each handler is a domain which specifies the region of computation during which, if an exception occurs, the handler will be activated
- The accuracy with which a domain can be specified will determine how precisely the source of the exception can be located

Ada

- In a block structured language, like Ada, the domain is normally the block.

```
declare
   subtype Temperature is Integer range 0 .. 100;
begin
   -- read temperature sensor and calculate its value
exception
   -- handler for Constraint_Error
end;
```

- Procedures, functions, accept statements etc. can also act as domains

Java

- Not all blocks can have exception handlers. Rather, the domain of an exception handler must be explicitly indicated and the block is considered to be **guarded**; in Java this is done using a try-block

```
try {
        // statements which may raise exceptions
}
catch (ExceptionType e) {
        // handler for e
}
```

Granularity of Domain

- Is the granularity of the block is inadequate?

```
declare
 subtype Temperature is Integer range 0 .. 100;
 subtype Pressure is Integer range 0 .. 50;
 subtype Flow is Integer range 0 .. 200;
begin
 -- read temperature sensor and calculate its value
 -- read pressure sensor and calculate its value
 -- read flow sensor and calculate its value
 -- adjust temperature, pressure and flow
 -- according to requirements
exception
 -- handler for Constraint_Error
end;
```

- The problem for the handler is to decide which calculation caused the exception to be raised
- Further difficulties arise when arithmetic overflow and underflow can occur

```
declare  -- First Solution: decrease block size
   subtype Temperature is Integer range 0 .. 100;
   subtype Pressure is Integer range 0 .. 50;
   subtype Flow is Integer range 0 .. 200;
begin
   begin
      -- read temperature sensor and calculate its value
   exception -- handler for Constraint_Error for temperature
   end;
   begin
      -- read pressure sensor and calculate its value
   exception -- handler for Constraint_Error for pressure
   end;
   begin
      -- read flow sensor and calculate its value
   exception -- handler for Constraint_Error for flow
   end;
   -- adjust temperature, pressure and flow according
   -- to requirements
exception  -- handler for other possible exceptions
end;
```

-- this **is** long-winded **and** tedious!

Solution 2: Allow exceptions to be handled at the statement level

```
-- NOT VALID Ada
declare
   subtype Temperature is Integer range 0 .. 100;
   subtype Pressure is Integer range 0 .. 50;
   subtype Flow is Integer range 0 .. 200;
begin
   Read_Temperature_Sensor;
      exception -- handler for Constraint_Error;
   Read_Pressure_Sensor;
       exception -- handler for Constraint_Error;
   Read_Flow_Sensor;
       exception -- handler for Constraint_Error;
   -- adjust temperature, pressure and flow
   -- according to requirements
end;
```

- The CHILL programming language has such a facility.
- But, it intermingles the EH code with the normal flow of operation, which violates Requirement R2

Solution 3

- Allow parameters to be passed with the exceptions.
- With Java, this is automatic as the exception is an object and, therefore, can contain as much information and the programmer wishes
- In contrast, Ada provides a predefined procedure Exception_Information which returns implementation-defined details on the occurrence of the exception

Exception propagation

- If there is no handler associated with the block or procedure
  - regard it as a programmer error which is reported at compile time
  - but an exception raised in a procedure can only be handled within the context from which the procedure was called
  - eg, an exception raised in a procedure as a result of a failed assertion involving the parameters
- CHILL requires that a procedure specifies which exceptions it may raise (that is, not handle locally); the compiler can then check the calling context for an appropriate handler
- Java allows a function to define which exceptions it can raise; however, unlike CHILL, it does not require a handler to be available in the calling context

Alternative Approach

- Look for handlers up the chain of invokers; this is called propagating the exception — the Ada and Java approach
- A problem occurs where exceptions have scope; an exception may be propagated outside its scope, thereby making it impossible for a handler to be found
- Most languages provide a catch all exception handler
- An unhandled exception causes a sequential program to be aborted
- If the program contains more than one process and a particular process does not handle an exception it has raised, then usually that process is aborted
- However, it is not clear whether the exception should be propagated to the parent process

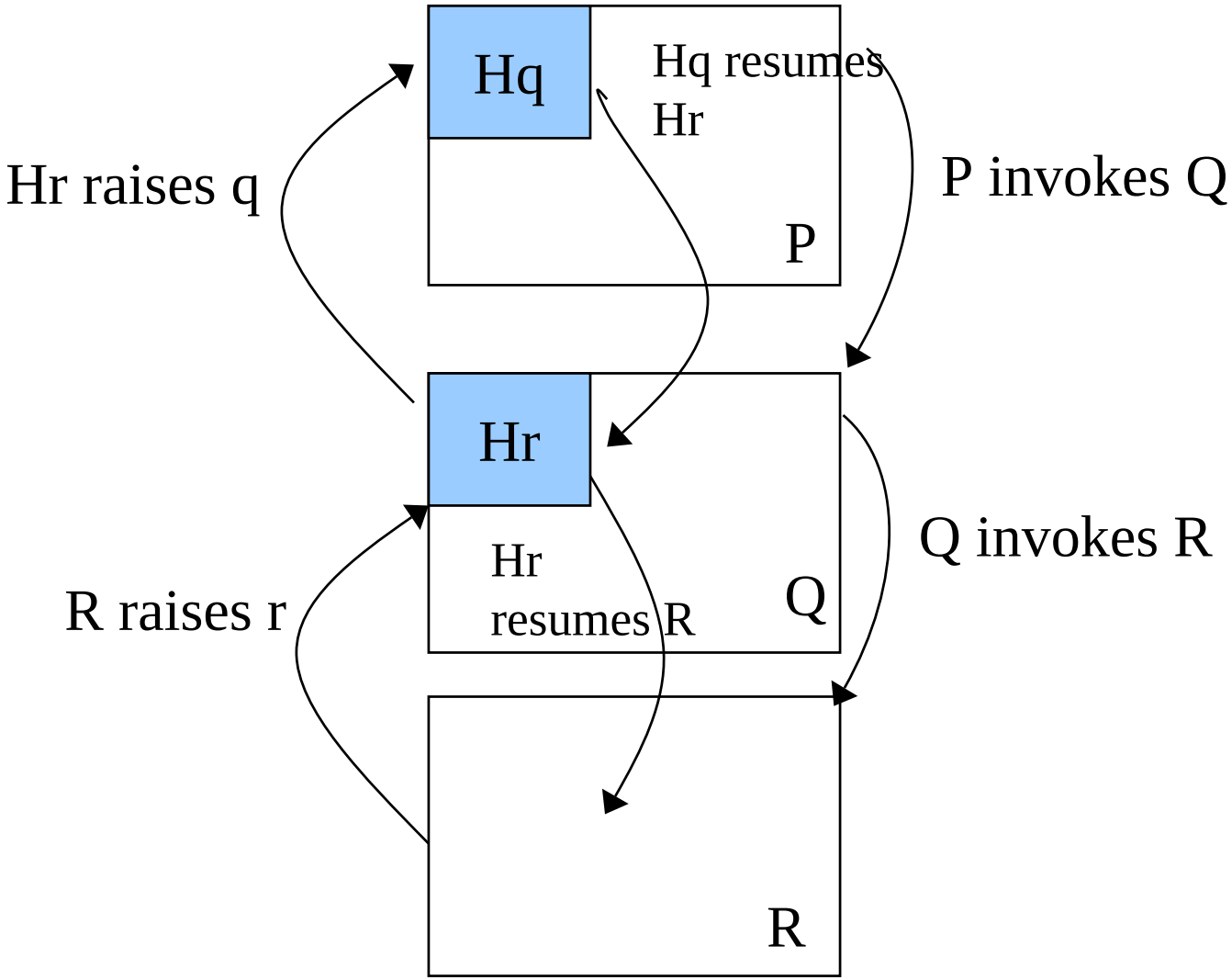
Resumption versus termination model

- Should the invoker of the exception continue its execution after the exception has been handled
- If the invoker can continue, then it may be possible for the handler to cure the problem that caused the exception to be raised and for the invoker to resume as if nothing has happened
- This is referred to as the resumption or notify model
- The model where control is not returned to the invoker is called termination or escape
- Clearly it is possible to have a model in which the handler can decide whether to resume the operation which caused the exception, or to terminate the operation; this is called the hybrid model

The Resumption Model

- Consider three procedures P, Q and R.
- P invokes Q which in turn invokes R.
- R raises an exception r which is handled by Q assuming there is no local handler in R.
- The handler for r is Hr.
- In the course of handling r, Hr raises exception q which is handled by Hq in procedure P (the caller of Q).
- Once this has been handled Hr continues its execution and when finished R continues
- Most easily understood by viewing the handler as an implicit procedure which is called when the exception is raised

The Resumption Model



The Resumption Model

- Problem: it is difficult to repair errors raised by the RTS
- Eg, an arithmetic overflow in the middle of a sequence of complex expressions results in registers containing partial evaluations; calling the handler overwrites these registers
- Pearl & Mesa support the resumption and termination models
- Implementing a strict resumption model is difficult, a compromise is to re-execute the block associated with the exception handler; Eiffel provides such a facility.
- Note that for such a scheme to work, the local variables of the block must not be re-initialised on a retry
- The advantage of the resumption model comes when the exception has been raised asynchronously and, therefore, has little to do with the current process execution

The Termination Model

- In the termination model, when an exception has been raised and the handler has been called, control does not return to the point where the exception occurred
- Instead the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure
- An invoked procedure, therefore, may terminate in one of a number of conditions
- One of these is the normal condition, while the others are exception conditions
- When the handler is inside a block, control is given to the first statement following the block after the exception has been handled

The Termination Model

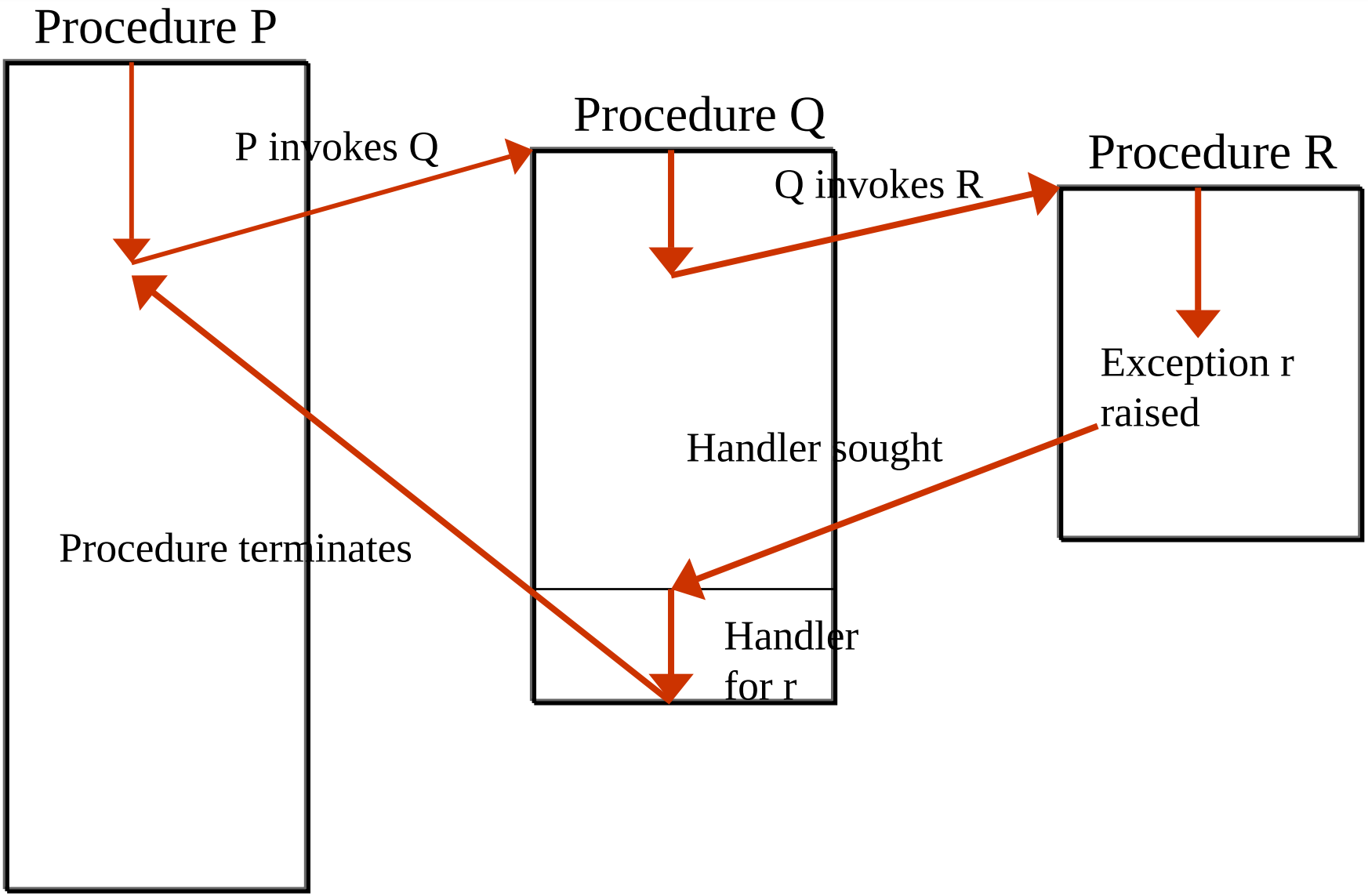
```
declare

  subtype Temperature is Integer range 0 .. 100;

begin
  ...
  begin
    -- read temperature sensor and calculate its value,
    -- may result in an exception being raised
  exception
    -- handler for Constraint_Error for temperature,
    -- once handled this block terminates
  end;
  -- code here executed when block exits normally
  -- or when an exception has been raised and handled.
exception
     -- handler for other possible exceptions
end;
```

- With procedures, as opposed to blocks, the flow of control can quite dramatically change
- Ada and Java support the termination model

The Termination Model



The Hybrid Model

- With the hybrid model, it is up to the handler to decide if the error is recoverable
- If it is, the handler can return a value and the semantics are the same as in the resumption model
- If the error is not recoverable, the invoker is terminated
- The signal mechanisms of Mesa provides such a facility
- Eiffel also supports the restricted `retry' model

Exception Handling and Operating Systems

- Languages like Ada or Java will usually be executed on top of an operating system
- These systems will detect certain synchronous error conditions, eg, memory violation or illegal instruction
- This will usually result in the process being terminated; however, many systems allow error recovery
- POSIX allows handlers to be called when these exceptions are detected (called signals in POSIX)
- Once the signal is handled, the process is resumed at the point where it was “interrupted” — hence POSIX supports the resumption model
- If a language supports the termination model, the RTSS must catch the error and manipulate the program state so that the program can use the termination model

Exception Handling in Ada

- Ada supports: explicit exception declaration, the termination model, propagation of unhandled exceptions, and a limited form of exception parameters.
- Exception declared: either by language keyword:

```
stuck_valve : exception;
```

- or by the predefined package `Ada.Exceptions` which defines a private type called `Exception_Id`
- Every exception declared by keyword has an associated `Exception_Id` which can be obtained using the pre-defined attribute `Identity`

```
package Ada.Exceptions is
  type Exception_Id is private;
  Null_Id : constant Exception_Id;

  function Exception_Name(Id : Exception_Id) return String;

  type Exception_Occurrence is limited private;
    Null_Occurrence : constant Exception_Occurrence;

  procedure Raise_Exception(E : in Exception_Id;
                            Message : in String := "");
  function Exception_Message(X :
                             Exception_Occurrence) return String;
  procedure Reraise_Occurrence(X : in
                             Exception_Occurrence);
  function Exception_Identity(X : Exception_Occurrence)
                             return Exception_Id;
  function Exception_Name(X : Exception_Occurrence)
                            return String;
  function Exception_Information(X :
                             Exception_Occurrence) return String;
  ....

private
  ... -- not specified by the language
end Ada.Exceptions;
```

Exception Declaration

```
with Ada.Exceptions;
with Valves;
package My_Exceptions is
   Id : Ada.Exceptions.Exception_Id :=
        Valves.Stuck_Valve'Identity;
end My_Exceptions;
```

- Applying the function Exception_Name to Id will now return the string `My_Exceptions.Id` and not `Stuck_Valve`
- An exception can be declared in the same place as any other declaration and, like every other declaration, it has scope

Raising an Exception

Exceptions may be raised explicitly

```
begin
   ...
   -- statements which request a device to
   -- perform some I/O
   if IO_Device_In_Error then
      raise IO_Error;
   end if; -- no else,as no return from raise
   ...
end;
```

- If `IO_Error` was of type `Exception_Id`, it would have been necessary to use `Ada.Exceptions.Raise_Exception`; this would have allowed a textual string to be passed as a parameter to the exception.
- Each individual raising of an exception is called an exception occurrence
- The handler can find the value of the Exception_Occurrence and used it to determine more information about the cause of the exception

Exception Handling

- Optional exception handlers can be declared at the end of the block (or subprogram, accept statement or task)
- Each handler is a sequence of statements

```
declare
   Sensor_High, Sensor_Low, Sensor_Dead : exception;
begin
   -- statements which may cause the exceptions
exception
   when E: Sensor_High | Sensor_Low =>
     -- Take some corrective action
     -- if either sensor_high or sensor_low is raised.
     -- E contains the exception occurrence
   when Sensor_Dead =>
     -- sound an alarm if the exception
     -- sensor_dead is raised
end;
```

Exception Handling

- **when others** is used to avoid enumerating all possible exception names
- Only allowed as the last choice and stands for all exceptions not previously listed

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead: exception;
begin
  -- statements which may cause exceptions
exception
  when Sensor_High | Sensor_Low =>
    -- take some corrective action
  when E: others  =>
    Put(Exception_Name(E));
    Put_Line(" caught. Information is available is ");
    Put_Line(Exception_Information(E));
    -- sound an alarm
end;
```

Exception Handling

- An exception raised in an handler cannot be handled by that handler or other handlers in the same block (or procedure)
- Instead, the block is terminated and a handler sought in the surrounding block or at the point of call for a subprogram

Exception propagation

- If there is no handler in the enclosing block/subprogram/ accept statement, the exception is propagated
- For a block, the exception is raised in the enclosing block, or subprogram.
- For a subprogram, it is raised at its point of call
- For an accept statement, it is raised in both the called and the calling task
- Exception handlers provided in the initialisation section of packages WILL NOT handle exceptions that are raised in the execution of their nested subprograms

```
package Temperature_Control is
  subtype Temperature is Integer range 0 .. 100;
  Sensor_Dead, Actuator_Dead : exception;
  procedure Set_Temperature(New_Temp: Temperature);
end Temperature_Control;

package body Temperature_Control is
  procedure Set_Temperature(...) is
  begin ... raise; end Set_Temperature;

begin
  -- initialisation of package
  Set_Temperature(Initial_Reading);
exception
  when Actuator_Dead =>
    -- take some corrective action
end Temperature_Control;
```

Last wishes

- Often the significance of an exception is unknown to the handler which needs to clean up any partial resource allocation
- Consider a procedure which allocates several devices.

```
procedure Allocate (Number :  Devices) is
begin
   -- request each device be allocated in turn
   -- noting which requests are granted
exception
  when others  =>
     -- deallocate those devices allocated
    raise; -- re-raise the exception
end Allocate;
```

- Used in this way, the procedure can be considered to implement the failure atomicity property of an atomic action; all the resources are allocated or none are

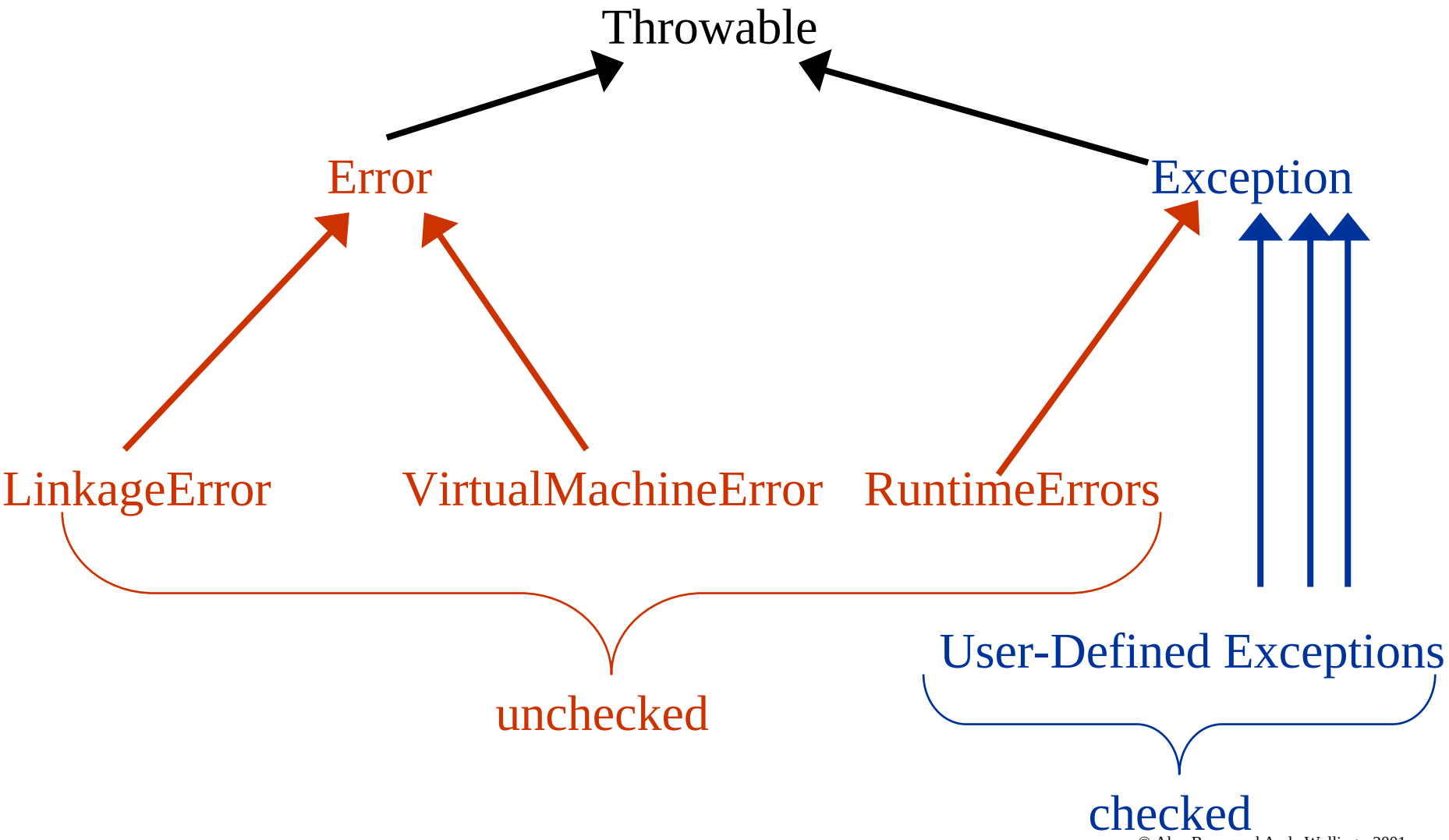
Difficulties with the Ada model of Exceptions

- Exceptions and packages
  - Exceptions which are raised a package are declared its specification
  - It is not known which subprograms can raise which exceptions
  - The programmer must resort to enumerating all possible exceptions every time a subprogram is called, or use of when others
  - Writers of packages should indicate which subprograms can raise which exceptions using comments
- Parameter passing
  - Ada only allows strings to be passed to handlers
- Scope and propagation
  - Exceptions can be propagated outside the scope of their declaration
  - Such exception can only be trapped by when others
  - They may go back into scope again when propagated further up the dynamic chain; this is probably inevitable when using a block structured language and exception propagation

Java Exceptions

- Java is similar to Ada in that it supports a termination model of exception handling
- However, the Java model is integrated into the OO model
- In Java, all exceptions are subclasses of the predefined class `java.lang.Throwable`
- The language also defines other classes, for example: `Error`, `Exception`, and `RuntimeException`

The Throwable Class Hierarchy



Example

```
public class IntegerConstraintError extends Exception
{
  private int lowerRange, upperRange, value;

  public IntegerConstraintError(int L, int U, int V)
  {
    super(); // call constructor on parent class
    lowerRange = L;
    upperRange = U;
    value = V;
  }

  public String getMessage()
  {
    return ("Integer Constraint Error: Lower Range " +
    java.lang.Integer.toString(lowerRange) + " Upper Range " +
    java.lang.Integer.toString(upperRange) + " Found " +
    java.lang.Integer.toString(value));
  }
}
```

```
import exceptionLibrary.IntegerConstraintError;

public class Temperature
{
  private int T;

  public Temperature(int initial) throws IntegerConstraintError
    // constructor
  {
    ...;
  }
  public void setValue(int V) throws IntegerConstraintError
  {
    ...;
  };
  public int readValue()
  {
    return T;
  };
  // both the constructor and setValue can throw an
  // IntegerConstraintError
};
```

```
class ActuatorDead extends Exception
{
  public String getMessage()
  { return ("Actuator Dead");}
};
class TemperatureController
{
  public TemperatureController(int T)
         throws IntegerConstraintError
  {
    currentTemperature = new Temperature(T);
  };
  Temperature currentTemperature;
  public void setTemperature(int T)
         throws ActuatorDead, IntegerConstraintError
  { currentTemperature.setValue(T); };

  int readTemperature()
  {
    return currentTemperature.readValue();
  }
};
```

Declaration

- In general, each function must specify a list of throwable checked exceptions **`throw`** `A, B, C`
  - in which case the function may throw any exception in this list and any of the unchecked exceptions.
- `A`, `B` and `C` must be subclasses of `Exception`
- If a function attempts to throw an exception which is not allowed by its throws list, then a compilation error occurs

Throwing an Exception

```
import exceptionLibrary.IntegerConstraintError;
class Temperature
{
  int T;

  void check(int value) throws IntegerConstraintError
  {
    if(value  > 100 || value  < 0) {
      throw new IntegerConstraintError(0, 100, value);
    };
  }

  public Temperature(int initial) throws IntegerConstraintError
    // constructor
  { check(initial); T = initial; }

  public void setValue(int V) throws IntegerConstraintError
  { check(V); T = V; };

  public int readValue()
  { return T; };
};
```

Exception Handling

```
// given TemperatureController TC

try {
  TemperatureController TC = new TemperatureController(20);

  TC.setTemperature(100);
  // statements which manipulate the temperature
}
catch (IntegerConstraintError error) {
  // exception caught, print error message on
  // the standard output
  System.out.println(error.getMessage());
}
catch (ActuatorDead error) {
  System.out.println(error.getMessage());
}
```

The `catch` *Statement*

- The **`catch`** statement is like a function declaration, the parameter of which identifies the exception type to be caught
- Inside the handler, the object name behaves like a local variable
- A handler with parameter type `T` will catch a thrown object of type `E` if:
  - `T` and `E` are the same type, or
  - `T` is a parent (super) class of `E` at the throw point
- It is this last point that makes the Java exception handling facility very powerful
- In the last example, two exceptions are derived from the `Exception` class: `IntegerConstraintError` and `ActuatorDead`

Catching All

```
try {
  // statements which might raise the exception
  // IntegerConstraintError or ActuatorDead
}
catch(Exception E) {
  // handler will catch all exceptions of
  // type exception and any derived type;
  // but from within the handler only the
  // methods of Exception are accessible
}
```

- A call to `E.getMessage` will dispatch to the appropriate routine for the type of object thrown
- **`catch`**`(Exception E)` is equivalent to Ada's **`when others`**

Finally

- Java supports a **`finally`** clause as part of a try statement
- The code attached to this clause is guaranteed to execute whatever happens in the try statement irrespective of whether exceptions are thrown, caught, propagated or, indeed, even if there are no exceptions thrown at all

```
try
{
   ...
}
catch(..)
{
...
}
finally
{
 // code executed under all circumstances
}
```

C Exceptions

- C does not define any exception handling facilities
- This clearly limits its in the structured programming of reliable systems
- However, it is possible to provide some form of exception handling mechanism by using the C macro facility
- To implement a termination model, it is necessary to save the status of a program's registers etc. on entry to an exception domain and then restore them if an exception occurs.
- The POSIX facilities of **setjmp** and **longjmp** can be used for this purpose

Setjmp and Longjmp

- **setjump** saves the program status and returns a 0
- **longjmp** restores the program status and results in the program abandoning its current execution and restarting from the position where **setjump** was called
- This time **setjump** returns the values passed by **longjmp**

```
/* begin exception domain */

typedef char *exception;
       /* a pointer type to a character string */
exception error ="error";
        /* the representation of an exception named "error" */

if(current_exception =  (exception) setjmp(save_area) == 0) {
  /* save the registers and so on in save_area */
  /* 0 is returned */

  /* the guarded region */

  /* when an exception "error" is identified */
  longjmp(save_area, (int) error);
  /* no return */

}
else {
  if(current_exception == error) {
    /* handler for "error" */
  }
  else {
    /* re-raise exception in surrounding domain */
  }
}
```

C Macros

```
#define NEW_EXCEPTION(name) ...
      /* code for declaring an exception */
#define BEGIN ...
      /* code for entering an exception domain */
#define EXCEPTION ...
      /* code for beginning exception handlers */
#define END ...
      /* code for leaving an exception domain */
#define RAISE(name) ...
      /* code for raising an exception */
#define WHEN(name) ...
      /* code for handler */
#define OTHERS ...
      /* code for catch all exception handler */
```

Termination Model

```
  NEW_EXCEPTION(sensor_high);

  NEW_EXCEPTION(sensor_low);

  NEW_EXCEPTION(sensor_dead);
  /* other declarations */

BEGIN
  /* statements which may cause the above */
  /* exceptions to be raised; for example */
  RAISE(sensor_high);

EXCEPTION
  WHEN(sensor_high)
      /* take some corrective action */
  WHEN(sensor_low)
      /* take some corrective action */
  WHEN(OTHERS)
            /* sound an alarm */
END;
```

Recovery Blocks and Exceptions

- Remember:

```
ensure  <acceptance test>
by
   <primary module>
else by
   <alternative module>
else by
   <alternative module>
...
else by
   <alternative module>
else error
```

- Error detection is provided by the acceptance test; this is simply the negation of a test which would raise an exception
- The only problem is the implementation of state saving and state restoration

A Recovery Cache

- Consider

```
package Recovery_Cache is
   procedure Save; -- save volatile state
   procedure Restore; --restore state
end Recovery_Cache;
```

- The body may require support from the run-time system and possibly even hardware support for the recovery cache
- Also, this may not be the most efficient way to perform state restoration
- It may be more desirable to provide more basic primitives, and to allow the program to use its knowledge of the application to optimise the amount of information saved

Recovery Blocks in Ada

```
procedure Recovery_Block is
  Primary_Failure, Secondary_Failure,
  Tertiary_Failure: exception;
  Recovery_Block_Failure : exception;
  type Module is (Primary, Secondary, Tertiary);

  function Acceptance_Test return Boolean is
  begin
    -- code for acceptance test
  end Acceptance_Test;
```

```
procedure Primary is
begin
  -- code for primary algorithm
  if not Acceptance_Test then
    raise Primary_Failure;
  end if;
exception
  when Primary_Failure  =>
    -- forward recovery to return environment
    -- to the required state
    raise;
  when others  =>
    -- unexpected error
    -- forward recovery to return environment
    -- to the required state
    raise Primary_Failure;
end Primary;
-- similarly for Secondary and Tertiary
```

```
  begin
    Recovery_Cache.Save;
     for Try in Module loop
     begin
       case Try is
         when Primary => Primary; exit;
         when Secondary => Secondary; exit;
         when Tertiary => Tertiary;
       end case;
     exception
       when Primary_Failure =>
         Recovery_Cache.Restore;
       when Secondary_Failure =>
         Recovery_Cache.Restore;
       when Tertiary_Failure =>
         Recovery_Cache.Restore;
         raise Recovery_Block_Failure;
       when others =>
         Recovery_Cache.Restore;
         raise Recovery_Block_Failure;
     end;
   end loop;
end Recovery_Block;
```

Summary

- All exception handling models address the following issues
  - Exception representation: an exception may, or may not, be explicitly represented in a language
  - The domain of an exception handler: associated with each handler is a domain which specifies the region of computation during which, if an exception occurs, the handler will be activated
  - Exception propagation: when an exception is raised and there is no exception handler in the enclosing domain, either the exception can be propagated to the next outer level enclosing domain, or it can be considered to be a programmer error
  - Resumption or termination model: this determines the action to be taken after an exception has been handled.

Summary

- With the resumption model, the invoker of the exception is resumed at the statement after the one at which the exception was invoked
- With the termination model, the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure.
- The hybrid model enables the handler to choose whether to resume or to terminate
- Parameter passing to the handler -- may or may not be allowed

Summary

Language	Domain	Propagation	Model	Parameters
Ada	Block	Yes	Termination	Limited
Java	Block	Yes	Termination	Yes
C++	Block	Yes	Termination	Yes
CHILL	Statement	No	Termination	No
CLU	Statement	No	Termination	Yes
Mesa	Block	yes	Hybrid	Yes

Summary

- It is not unanimously accepted that exception handling facilities should be provided in a language
- The C and the occam2 languages, for example, have none
- To sceptics, an exception is a GOTO where the destination is undeterminable and the source is unknown!
- They can, therefore, be considered to be the antithesis of structured programming
- This is not the view taken here!