Chapter 3: Exceptions and Exception Handling

Aims

- To illustrate the various model of exception handing
- To consider the Ada and Java models of exception handling in detail and to show how exception handling can be used as a framework for implementing fault-tolerant systems

INTRODUCTION
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

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
}
```

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

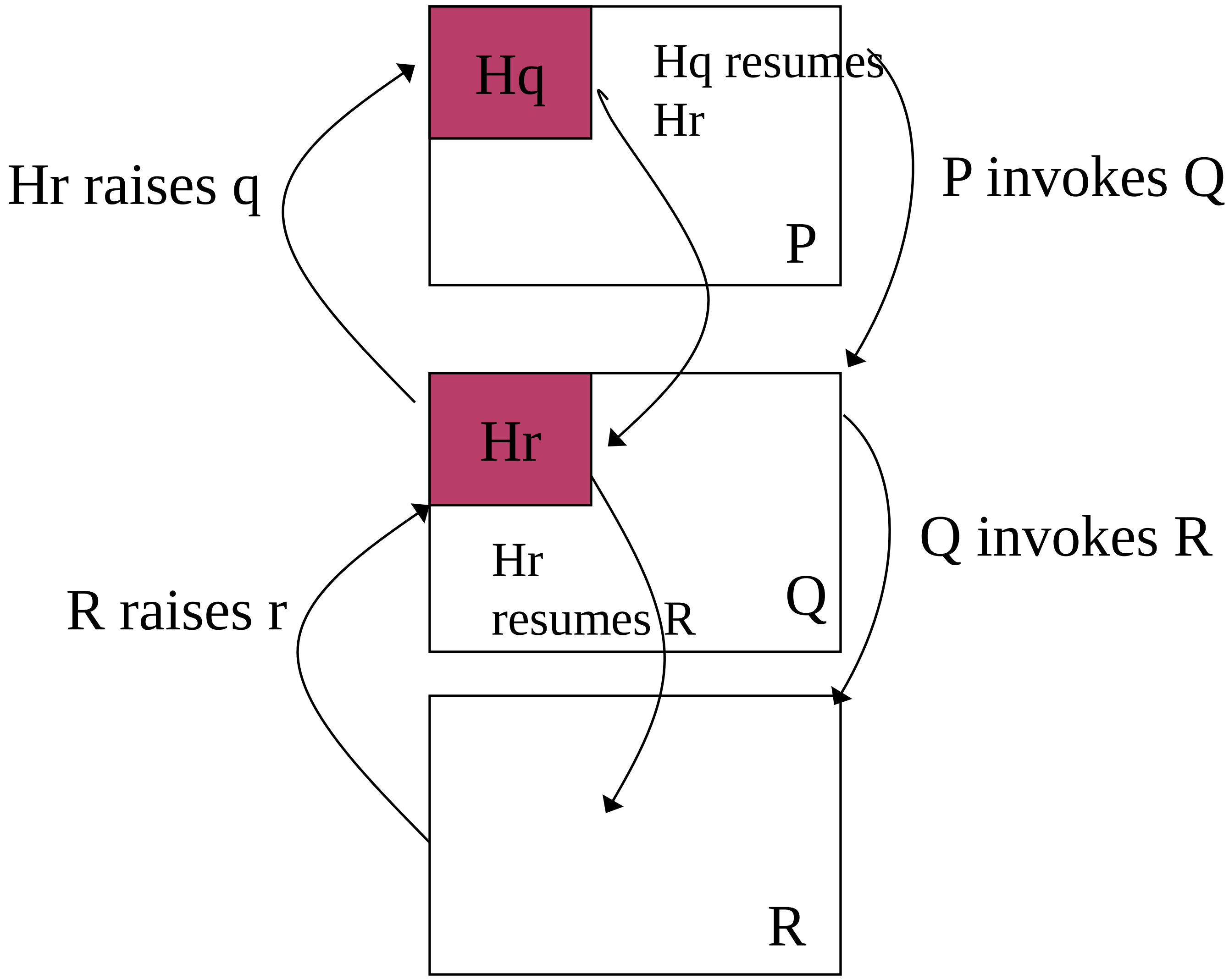
ALTERNATIVE APPROACH

- 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



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

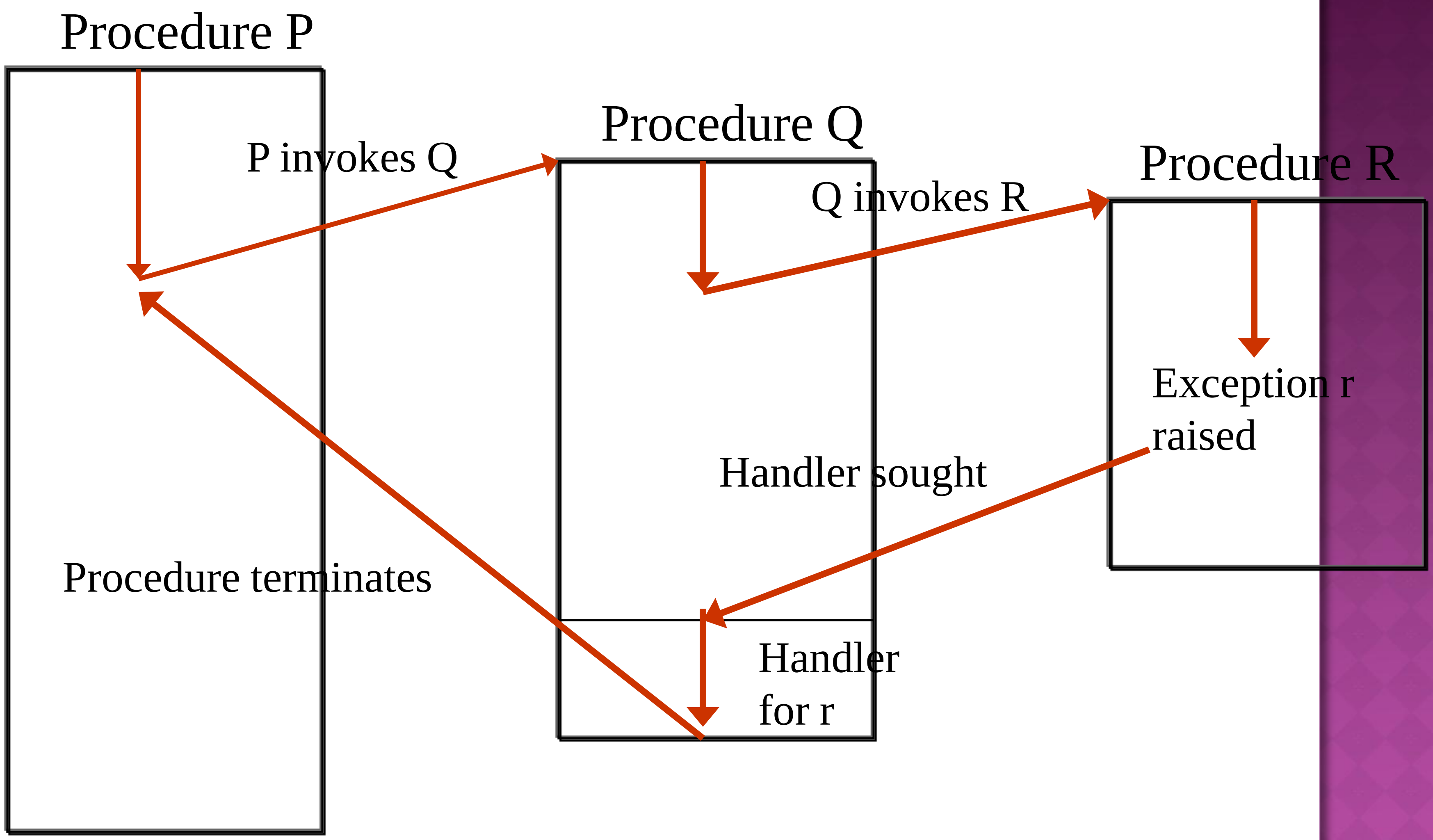
```
declare

  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



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`

```ada
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;
```

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  =>  // last wishes
    Put(Exception_Name(E));
    Put_Line(" caught. Information is available is ");
    Put_Line(Exception_Information(E));
    -- sound an alarm
end;
```

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

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;
```

Controlled Types

- Allow objects to have initialization and finalization routines

```
package Ada.Finalization is
  pragma Preelaborate(Finalization);

  type Controlled is abstract tagged private;
  procedure Initialize(Object : in out Controlled);
  procedure Adjust(Object : in out Controlled);
  procedure Finalize(Object : in out Controlled);

  type Limited_Controlled is abstract tagged private;
  procedure Initialize(Object : in out Limited_Controlled);
  procedure Finalize(Object : in out Limited_Controlled);
private
  ... -- not specified by the language
end Ada.Finalization;
```

LAST WISHES I

```
with Ada.Finalization; use Ada.Finalization;

package Last_Wishes is
   type Finalizer is new Controlled with private;
   procedure Finalize (O : in out Finalizer);
   procedure Got_Resource(R: Resource; O : in out Finalizer);
private
   type Finalizer is new Controlled with
   record
     Resource1_Acquired : boolean := false;

     ...
    end record;

end Last_Wishes;
```

LAST WISHES II

```
with Ada.Text_IO; use Ada.Text_IO;
package body Last_Wishes is
   procedure Finalize (O : in out Finalizer) is
   begin
     if Resource1_Acquired then
       -- return resources
     else
       ...
     end if;
   end Finalize;

   procedure Get_Resource(R: Resource; O : in out Finalizer) is
   begin
      -- set appropriate acquired field to true
   end Get_Resource;

end Last_Wishes;
```

Reliable Resource Usaged

```
with Last_Wishes; use Last_Wishes;
procedure Reliable_Resource_Usage is
   final : Finalizer;
begin
   -- get resources etc
   Get_Resource(R1, final);
   -- use resources
end Reliable_Resource_Usage;
```

LAST WISHES AND TASKS I

- Example: count the number of times two entries are called

```
with Ada.Finalization; use Ada;
package Counter is
 type Task_Last_Wishes is new
           Finalization.Limited_Controlled
    with record
      Count1, Count2 : Natural := 0;
    end record;
  procedure Finalize(Tlw : in out Task_Last_Wishes);
end Counter;
```

LAST WISHES AND TASKS II

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Text_IO; use Ada.Text_IO;
package body Counter is
  procedure Finalize(Tlw : in out Task_Last_Wishes) is
  begin
    Put("Calls on Service1:");
    Put(Tlw.Count1);
    Put(" Calls on Service2:");
    Put(Tlw.Count2);
    New_Line;
  end Finalize;
end Counter;
```

Last Wishes and Tasks III

```
task body Server is
  Last_Wishes :  Counter.Task_Last_Wishes;
begin
  -- initial housekeeping
  loop
    select
      accept Service1(...) do
        ...
      end Service1;
      Last_Wishes.Count1 :=  Last_Wishes.Count1 + 1;
    or
      accept Service2(...) do
        ...
      end Service2;
      Last_Wishes.Count2 :=  Last_Wishes.Count2 + 1;
    or
      terminate;
    end select;
    -- housekeeping
  end loop;
end Server;
```

Note, can be used to get last wishes from a procedure as well

As the task terminates the finalize procedure is executed

LAST WISHES — ADA 2005

- Ada 2005 has added a new facility
- When a task terminates, application code can be executed
- A task can have a specific `termination handler' or
- A default handler can be executed
- The handler knows the cause of termination
  - This includes unhandled exceptions

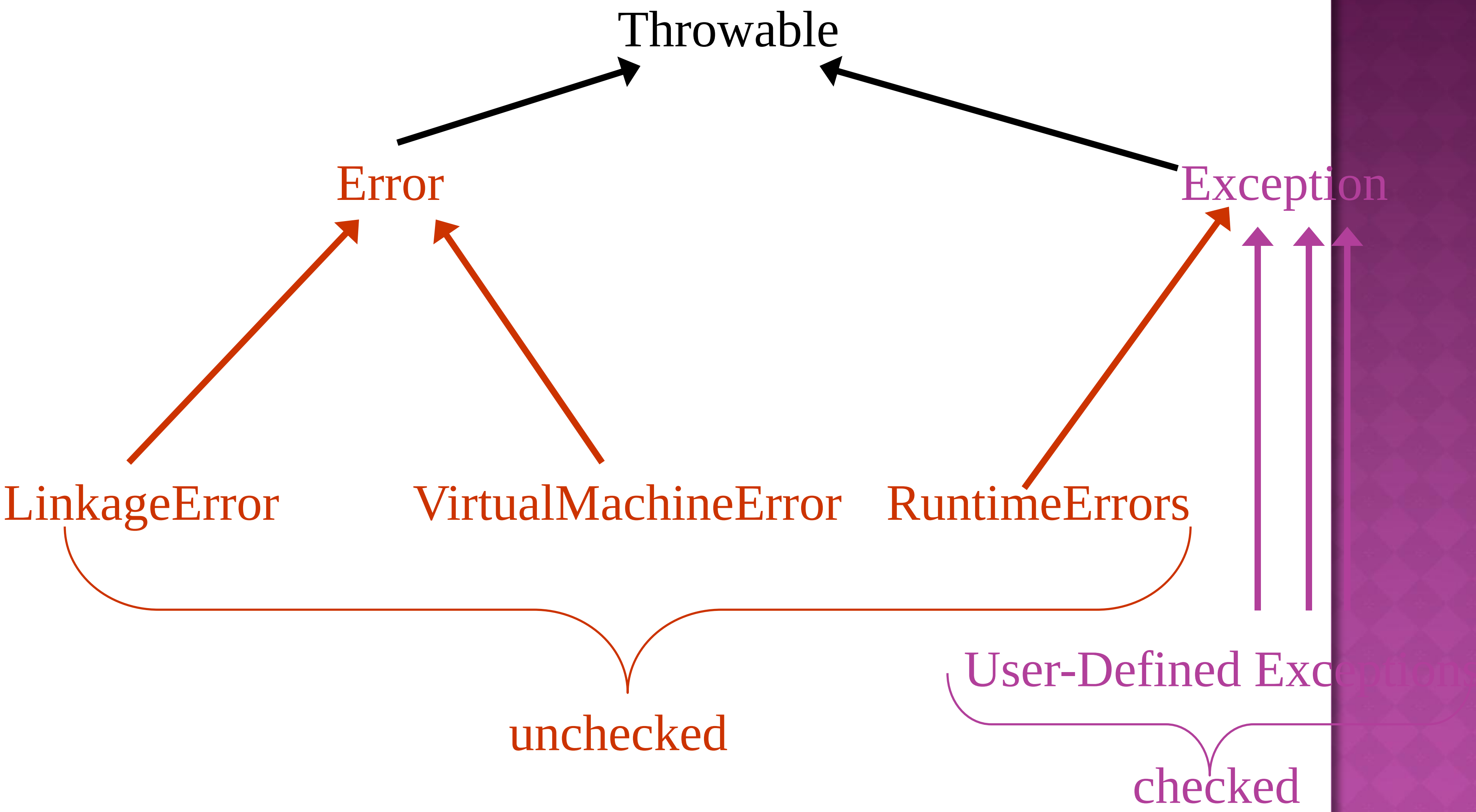
Difficulties with the Ada Model of Exceptions

- 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

```java
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));
  }
}
```

EXAMPLE CONTINUED

```
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
}
```

```java
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 {
     // check Actuator
     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
- This 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
}
```

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
  procedure Discard; -- discard the state
end Recovery_Cache;
```

- The body may require support from the run-time system and possibly even hardware support for the recovery cache

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;
         Recovery_Cache.Discard;
         raise Recovery_Block_Failure;
       when others =>
         Recovery_Cache.Restore;
         Recovery_Cache.Discard;
         raise Recovery_Block_Failure;
     end;
   end loop;
   Recovery_Cache.Discard;
 end Recovery_Block;
```

Summary I

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 II

- 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!