Exception Handling

Examples

Exception Handling

- Exception handling is a flow control mechanism that alters the normal operation of a program.
  - Control can be passed to code in the same module or unit, passed up the call stack, or passed to an entirely different module.
  - By design, it is used as an error handling mechanism.
- Most people agree that exception handling should be used primarily for error handling and not as a flow control mechanism.
- Why not use it for control flow?
  - The usual reasons involving readability, writability, and difficulty of maintenance.

Robust Programs

- Applications are robust when they continue to operate correctly under all conceivable error conditions.
  - We don’t want the computer controlling the engine in a car to stop operating with a cryptic error message when an error occurs.
  - We also don’t want it to accelerate to 100mph and disable the brakes when an error occurs.

Hardware Errors

- Hardware errors:
  - Disk read failure
  - Storage full
  - Memory parity error
  - Network device failure
  - Page fault
  - Null pointer access (if detected by hardware)
  - Illegal memory access
- Errors such as these are detected and handled by low-level OS code that checks device status registers or responds to hardware signals.
- The errors are handled by the operating system but passed back to the application in some form.

Example of Non-Robust Code

(* Pascal *)
reset(file, name); (* open *)
sum := 0.0;
count := 0;
while (not eof(file)) do begin
  read(file, number);
  sum := sum + number;
count := count + 1;
end;
avg := sum / count;
• What can go wrong?
  - If whitespace occurs after last number EOF will be false
  - Number is not in a valid format
  - Open fails
  - No data in file and count is 0
Levels of Abstraction in Errors

- **OS Level** - errors that occur when operating systems services fail
  - Some hardware errors
  - EOF on input
  - Communications failure
  - File not found
- **Programming language level**
  - Array access out of bounds, run-time type errors, data not in expected format
- **Programmer defined**
  - Stack full, pop an empty stack

Exception And Error Handling

- Without exception or error handling
  - When an exception or error occurs at the OS level, the program is terminated and the OS displays a message
  - Errors at the application level such as buffer overflow, index out of bounds, type error etc may allow the program to continue to operate erratically or crash spectacularly
- With exception handling
  - Programs can trap some exceptions and/or test for errors
  - Goal is to either fix the problem and continue or at least die gracefully
  - Essential for operating systems

Exception Handling Alternatives

- An exception is raised when its associated event occurs
- A language that does not have exception handling capabilities can still define, detect, raise, and handle exceptions (user defined, software detected)
- Alternatives are various error handling techniques:
  - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram
  - Pass a label parameter to all subprograms (error return is to the passed label)
  - Pass an exception handling subprogram to all subprograms

Error Handling Strategies

- Many languages do not support exception handling
- A number of strategies for handling errors:
  - Return illegal value from function as error indicator (example: IndexOf for array returns -1)
  - Write functions that return boolean success indicators
  - Add error parameter to a void function
  - Try to anticipate possible errors and add conditional guards to statement blocks

Built-in Exception Handling

- Error detection code is tedious to write and it clutters the program
- Exception handling encourages programmers to consider many different possible errors
- Exception propagation allows a high level of reuse of exception handling code
- Exceptions can be handled in some other unit of dynamic or static scope so that error handling is centralized and encapsulated rather than being strewn throughout the code

Design Issues

- How are user-defined exceptions specified?
- Should there be default exception handlers for programs that do not provide their own?
- Can built-in exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that can be handled?
- Are there any built-in exceptions?
- How can exceptions be disabled, if at all?
Design Issues

• How and where are exception handlers specified and what is their scope?
• How is an exception occurrence bound to an exception handler?
• Can information about the exception be passed to the handler?
• Where does execution continue, if at all, after an exception handler completes its execution? (terminate or resume?)
• Is some form of finalization provided?

Exception Handling Control Flow

Terminate or Resume?

• Resumption model transfers control back to faulting code after exception has been handled
• Termination model transfers control back to caller
• Either may be appropriate depending on type of exception and resolution of problem.
• Languages such as Ada, Java, C++ all use the termination model
  - If resumption model is needed then an alternate error handling mechanism must be used

Implementation Complexity

• Both terminate and resume models have complex issues
  - Resume: have to restore state of program at time exception was called
  - Terminate: what happens to resources (memory, open files, etc) belonging to the terminated code?

Propagation of exceptions

• Languages that support exceptions associate exceptions with a scope or block of code
  - Java, C++, C# use the Try block
  - Ada allows exception handlers to be defined at both block and subprogram levels of scope
• If no handler is found within current scope, the exception is propagated up the call stack
• If no handler is found then typically the language runtime will handle the exception by terminating the program

Exception Handling Evolution

• COBOL provided very limited exception handling
• PL/I generalized the idea, first language with real exception handling (1970’s)
• Ada is usually considered to be the first language with well defined an usable exception handling (1983)
  - But most academic authors fail to mention BASIC with its ON ERROR GOTO statement (early 1980s)
• Mainstream adoption came in 1990’s with C++ compilers
Exception Handling in Ada

- The Ada design for exception handling embodies the state-of-the-art in language design in 1980
  - Ada was the only widely used language with exception handling until it was added to C++
  - There are a few technical problems with Ada exceptions especially with object oriented extensions
- The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block
- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters

Ada Exception Handlers

- Handler form:
  ```ada
  when exception_choice =>
  statement_sequence
  ...
  when others =>
  statement_sequence
  ```
- exception_choice form:
  ```ada
  exception_name | others
  ```
- Handlers are placed at the end of the block or unit in which they occur

Example

```ada
with Ada.Exceptions, Ada.Text_IO;

procedure Foo is
  Some_Error : exception;
begin
  Do_Something_Interesting;
  exception
    when Constraint_Error => -- Start of exception handlers
    when Storage_Error => -- Propagate Storage_Error as a different exception
      raise Some_Error with "Out of memory";
    when Error : others => -- Handle all others
      Ada.Text_IO.Put("Exception: ");
      Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
  end Foo;
```

Binding Exceptions to Handlers

- If the block or unit in which an exception is raised does not have a handler for that exception, the exception is propagated elsewhere to be handled
  - Procedures - propagate it to the caller
  - Blocks - propagate it to the scope in which it appears
  - Package body - propagate it to the declaration part of the unit that declared the package (if it is a library unit, the program is terminated)
  - Task - no propagation; if it has a handler, execute it; in either case, mark it "completed"

Continuation

- The block or unit that raises an exception but does not handle it is always terminated (also any block or unit to which it is propagated that does not handle it)
- Control never returns to the block or unit that raised the exception

Example

```ada
package Directory_Enquiries is
  procedure Insert (New_Name : in Name;
                   New_Number : in Number);
  procedure Lookup (Given_Name : in Name;
                  Corr_Number : out Number);
  Name_Duplicated : exception;
  Name_Absent : exception;
  Directory_Full : exception;
end Directory_Enquiries;
```

```ada
package Directory_Enquiries is
  procedure Insert (New_Name : in Name;
                   New_Number : in Number);
  procedure Lookup (Given_Name : in Name;
                  Corr_Number : out Number);
  Name_Duplicated : exception;
  Name_Absent : exception;
  Directory_Full : exception;
end Directory_Enquiries;
```
Example

```vhdl
package body Directory_Enquiries is

procedure Insert (New_Name in Name;
                 New_Number in Number)
is
  begin
    if New_Name = old_Name then
      raise Name_Duplicated;
    end if;

  end Insert;

procedure Lookup (Given_Name in Name;
                 Corr_Number out Number)
is
  begin
    if not Found then
      raise Name_Absent;
    end if;

end Lookup;
end
```

Other Design Choices

- User-defined Exceptions form:
  ```vhdl```
```vhdl```
exception_name_list : exception;
```vhdl```
- Raising Exceptions form:
  ```vhdl```
```vhdl```
raise [exception_name]
```vhdl```
- (the exception name is not required if it is in a handler--in this case, it propagates the same exception)
```vhdl```
- Exception conditions can be disabled with:
  ```vhdl```
```vhdl```
pragma SUPPRESS(exception_list)
```vhdl```

Predefined Exceptions

- CONSTRAINT_ERROR - index constraints, range constraints, etc.
- NUMERIC_ERROR - numeric operation cannot return a correct value (overflow, division by zero, etc.)
- PROGRAM_ERROR - call to a subprogram whose body has not been elaborated
- STORAGE_ERROR - system runs out of heap
- TASKING_ERROR - an error associated with tasks

Exception Handling in C++

- Added to C++ ANSI Standard in 1990
  - Design is based on that of CLU, Ada, and ML
  - But C++ has no standard exceptions: all are user/library defined
- Scope of C++ exception handler is a try block rather than a program unit

C++ Exception Handlers

- Try block
  ```vhdl```
```vhdl```
try {
  /* code that is expected to raise an exception */
} catch (formal parameter) {
  // handler code
  ...
  catch (formal parameter) {
    // handler code
  } catch (...) {
    // catch anything handler code
  }
```vhdl```

The Catch Function

- `catch` is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter can be a bare type name instead of a variable
  - Provides a signature to distinguish the handler from others
  - With a variable name the formal parameter can be used to transfer information to the handler
- An exception raised in a try block causes immediate transfer of control to catch handlers
  - Catch blocks are searched sequentially so the convention is to place more specific handlers at the top of the catch block and more generic ones below
  - The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled and guarantees that all exceptions will be caught
C++ Catch Clause

- From a C++ manual:

```cpp
catch(T)
catch(const T)
catch(T&)
catch(const T&)
```

Such handlers can catch exception objects of type E if:

1. T and E are the same type, or
2. T is an accessible base class of E at the throw point, or
3. T and E are pointer types and there exists a standard pointer conversion from E to T at the throw point. T is an accessible base class of E if there is an inheritance path from E to T with all derivations public.

Throwing Exceptions

- Exceptions are all raised explicitly by the statement:
  ```cpp
  throw expression;
  ```
- `throw` without an operand can only appear in a handler
  - when it appears, it simply re-raises the exception, which then must be handled elsewhere
- The type of the expression disambiguates the intended handler

Unhandled Exceptions

- An unhandled exception is propagated to the caller of the function in which it is raised
- This propagation continues to the main function
- If no handler is found, the default handler is called

Continuation

- After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element
- Other design choices
  - All exceptions are user-defined
  - Exceptions are neither specified nor declared
  - The default handler, `unexpected`, simply terminates the program; `unexpected` can be redefined by the user
  - Functions can list the exceptions they may raise
  - Without a specification, a function can raise any exception (the `throw` clause)

C++ Example 1

```cpp
#include <iostream.h>
int main () {
  char A[10];
  cin >> n;
  try {
    for (int i=0; i<n; i++)
      if (i>9) throw "array index error";
    A[i]=getchar();
  }
  catch (char* s)
    { cout << "Exception: " << s << endl; }
  return 0;
}
```

Example

```cpp
/* this example computes a frequency distribution for a set of grades read from cin and terminated by the appearance of a grade < 0 */
#include <iostream>
int main() {
  int new_grade, index, limit1, limit2;
  int freq[10] = {0,0,0,0,0,0,0,0,0,0};
  class NegativeInputException()
    public:
      NegativeInputException() { // constructor
        cout << "End of input data reached << endl; }
    }  
```
Example

```cpp
try {
    while (true) {
        cout << "please enter a grade" << endl;
        if ((cin >> newgrade) < 0) // end of data
            throw NegativeInputException();
        index = new_grade / 10;
        try {
            if (index > 9)
                throw new_grade;
            freq[index]++;
        } catch (int grade) {
            if (grade == 100)
                freq[9]++;
            else
                cout << "Error: new grade " << grade << " out of range." << endl;
        }
    }
}
```
Java Exception Class Hierarchy

Binding Exceptions to Handlers

• When an exception is thrown an instance of the exception class is the operand of the throw statement
  ```java
class MyException extends Exception {
  public MyException() {}
  public MyException (String msg){
    super (msg);
  }
  ...  
  throw new MyException();
  throw new MyException("A weird error occurred.");
}
```

• Binding an exception to a handler is similar to C++
  - An exception is bound to the first handler with a parameter is the same class as the thrown object or an ancestor of it
  - The generic exception handler that handles anything is
    ```java
catch(Exception genericObj){
}
```
  - An exception can be handled and rethrown by including a throw in the handler (a handler could also throw a different exception)

Java throws Clause

• Specifies what exceptions might be raised
  ```java
  public void methodA() throws SomeException,
  AnotherException {
    //methodbody
  }
  public void methodB() throws CustomException {
    //Methodbody
  }
  public void methodC() {
    try
      methodB();
      methodA();
    catch (Exception e){
      ...
    }
    finally {
      //clean up ...
    }
  }
```

Checked and Unchecked Exceptions

• Exceptions of class Error and RuntimeException and all of their descendants are called unchecked exceptions
• All other exceptions are called checked exceptions
• Checked exceptions that may be thrown by a method must be either:
  - Listed in the throws clause, or
  - Handled in the method
• Checked at compile time
Other Design Choices

- A method cannot declare more exceptions in its `throws` clause than the method it overrides.
- A method that calls a method that lists a particular checked exception in its `throws` clause has three alternatives for dealing with that exception:
  - Catch and handle the exception
  - Catch the exception and throw an exception that is listed in its own `throws` clause
  - Declare it in its `throws` clause and do not handle it

The finally Clause

- The finally clause can appear at the end of a `try` construct
- Form:
  ```java
  finally {
      ...
  }
  ```
- Specifies code that is to be executed, regardless of what happens in the `try` construct
  - Typical uses are freeing resources such as an open file or database connection that must be closed regardless of error conditions

The Finally Block

- Java and C# support the finally block - a major omission in C++
  ```java
  try {
      // The guarded region:
      // Errors might throw A, B, or C
      catch (A a1) {
          // Handle A
      }
      catch (B b1) {
          // Handle B
      }
      catch (C c1) {
          // Handle C
      }
      finally {
          // Executed whether or not there was an exception
      }
  }
  ```

Example

- A try construct with a finally clause can be used outside exception handling
- The finally clause will be executed even if the return statements terminates the loop
  ```java
  try {
      for (index = 0; index < 100; index++) {
          ...
      }
  } finally {
      ...
  }
  ```

Assertions

- Primarily used during program development
- Statements in the program declaring a boolean expression regarding the current state of the computation
  - When evaluated to true nothing happens
  - When evaluated to false an `AssertionError` exception is thrown
  - Can be disabled during runtime without program modification or recompilation
- Two forms
  - `assert` condition;
  - `assert condition; expression;`

Assertion or Exception?

- From Programming with Assertions
  http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html
  - By convention, preconditions on public methods are enforced by explicit checks that throw particular, specified exceptions. For example:
    ```java
    /** Sets the refresh rate. * @param rate refresh rate, in frames per second. * @throws IllegalArgumentException if rate <= 0 or * rate > MAX_REFRESH_RATE. */
    public void setRefreshRate(int rate) {
        // Enforce specified precondition in public method
        if (rate <= 0 || rate > MAX_REFRESH_RATE) {
            throw new IllegalArgumentException("Illegal rate: "+ rate);
        }
        setRefreshInterval(1000/rate);
    }
    ```
  - This convention is unaffected by the addition of the `assert` construct.
  - Do not use assertions to check the parameters of a public method.
  - An assert is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, the assert construct does not throw an exception of the specified type, it can throw only an `AssertionError`. 
Assertion or Exception?

- You can, however, use an assertion to test a nonpublic method's precondition that you believe will be true no matter what a client does with the class. For example, an assertion is appropriate in the following 'helper method' that is invoked by the previous method:

```java
/**
 * Sets the refresh interval (which must correspond
 * to a legal frame rate).
 * @param interval refresh interval in milliseconds.*/
 private void setRefreshInterval(int interval) {
    // Confirm adherence to precondition
    // in nonpublic method
    assert interval > 0 &&
    interval <= 1000/MAX_REFRESH_RATE : interval;
    ... // Set the refresh interval
}
```

Exceptions in Scripting Languages

- Most modern scripting languages implement some form of try ... catch block even though they use dynamic typing

JavaScript

```javascript
try {
    // run some code here
    catch(error) {
        // a javascript error object with
        // properties such as error.message
    }
    finally {
        // Statements that execute afterward either way
    }
    // JavaScript also has a throw statement
    // Many error object properties are vendor specific
}
```

PHP

```php
<?php
function inverse($x) {
    if (!$x) {
        throw new Exception('Division by zero.');
    }
    else return 1/$x;
}
try {
    echo inverse(5) . 
    echo inverse(0) . 
} catch (Exception $e) {
    echo 'Caught exception: ', $e->getMessage(), 
    echo 'Hello World';
    // PHP also has a throw statement
}
```

Python

```python
f = None
try:
    f = file("afilename")
    f.write(could_make_error())
except IOError:
    print "Unable to open file"
except: # catch all exceptions
    print "Unexpected error"
else:
    # executed if no exceptions are raised
    print "File write completed successfully"
finally:
    # cleanup actions, always executed
    if f:
        f.close()
```

Ruby

```ruby
begin
    # Do something nifty
    raise SomeError, "This is the error message!" # Uh-oh!
    rescue SomeError
        # This is executed when a SomeError exception
        # is raised
    rescue AnotherError => error
        # Here, the exception object is referenced from the
        # 'error' variable
        rescue
            # This catches all exceptions derived from StandardError
        end
    else
        # This is executed only if no exceptions were raised
        ensure
            # This is always executed, exception or not and
```
Abusing Exception Handling

- From http://leedumond.com/blog/the-greatest-exception-handling-wtf-of-all-time

```csharp
public static class NumberHelpers
{
    public static ApplicationException EvenOrOdd(int integer)
    {
        if (integer % 2 == 0)
        {
            return new ApplicationException("The integer is even.");
        }
        else
        {
            return new ApplicationException("The integer is odd.");
        }
    }
}
```

Usage

```csharp
protected void btnTest_Click(object sender, EventArgs e)
{
    try
    {
        throw NumberHelpers.EvenOrOdd(Convert.ToInt32(txtIntToTest.Text));
    }
    catch (ApplicationException ex)
    {
        litResult.Text = ex.Message;
    }
}
```

A light-hearted look at exceptions

- Shamelessly copied from http://www.theregister.co.uk/2006/01/11/exception_handling/page2.html

- What to do if you get an exception. Faced with this question, some writers indulge in hand waving. Fortunately I am able to offer specific guidelines, based on standard practice as observed in commercial software, and some informed guesswork.

- If you are running as some sort of web service, the standard approach on handling an exception is to send the user a page of ODBC diagnostics, preferably mashed up with a few suggestions from Apache.

A light-hearted look at exceptions

- But this approach doesn't just work with HTML output. For example some banks have also adopted it, as a novel method of telling you that you aren't going to see your card back from the ATM any time soon.

A light-hearted look at exceptions

- If you are running as a background process with no user interaction, don't just disappear silently. Be sure to do a memory dump, so gifting the user a digital turd - a binary lump of disk space of no use to man or beast. Naturally it won't help you-the-programmer find the cause of the exception, because even if the user troubles to send it to you, like everyone else you don't keep your debug symbol tables in version control, kidding yourself that you can rebuild them identically from the source.

A light-hearted look at exceptions

- (By the way, the Windows API call to create this thing, MiniDumpWriteDump(), is one of my faves: as it says in the docs, sometimes you have to throw another exception to get it to fire. Neat.)

- If you are GUI program, now's the time to pop up a modal message box. It doesn't really matter what text you put in it, because the user will ignore it.
• A refinement, especially popular with Delphi programmers, is to put up further, identical message boxes at a one half second interval, so that unless the user intervenes and starts closing them at a greater rate, the whole system will eventually die from memory exhaustion.

• If you’re running under Windows XP, consider converting the exception into a null pointer dereference in the catch handler:

```cpp
catch(...)
  i // now we're really stuffed
  int * p = 0;
  *p = 22;
```

• This has the advantage over an ordinary crash that you will get one of those special OS-supplied dialogs, that asks permission to send log details back to Microsoft. Naive users will interpret this as a Windows fault, and will direct their bile Redmondwards.

• Of course, all the above techniques can be combined in fresh and original ways. Never be afraid to experiment.

• The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user. Provided you keep this cardinal rule in mind, you can't go far wrong.