Chapter 13

In a language without exception handling:
When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated

In a language with exception handling:
Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing

Many languages allow programs to trap input/ output errors (including EOF)

Def: An *exception* is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing

Def: The special processing that may be required after the detection of an exception is called *exception handling*

Def: The exception handling code unit is called an *exception handler*

Chapter 13

Def: 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

- *Alternatives:*

1. **Send an auxiliary parameter or use the return value to indicate the return status of a subprogram**
   - e.g., C standard library functions

2. **Pass a label parameter to all subprograms (error return is to the passed label)**
   - e.g., FORTRAN

3. **Pass an exception handling subprogram to all subprograms**

Advantages of Built-in Exception Handling:

1. **Error detection code is tedious to write and it clutters the program**
2. **Exception propagation allows a high level of reuse of exception handling code**

Chapter 13

Design Issues for Exception Handling:

1. How and where are exception handlers specified and what is their scope?

2. How is an exception occurrence bound to an exception handler?

3. Where does execution continue, if at all, after an exception handler completes its execution?

4. How are user-defined exceptions specified?

5. Should there be default exception handlers for programs that do not provide their own?

6. Can built-in exceptions be explicitly raised?

7. Are hardware-detectable errors treated as exceptions that can be handled?

8. Are there any built-in exceptions?

9. How can exceptions be disabled, if at all?

Chapter 13

PL/I Exception Handling

***- Exception handler form*:**

```
ON condition [SNAP]
   BEGIN;
   ...
   END;
```

- condition is the name of the associated exception

- SNAP causes the production of a dynamic trace to the point of the exception

- Binding exceptions to handlers

- It is dynamic--binding is to the most recently executed ON statement

- Continuation

- Some built-in exceptions return control to the statement where the exception was raised
- Others cause program termination
- User-defined exceptions can be designed to go to any place in the program that is labeled

Chapter 13

- Other design choices:

- User-defined exceptions are defined with:
  `CONDITION` **exception_name**

- Exceptions can be explicitly raised with:
  `SIGNAL CONDITION (`**exception_name**`)`

- Built-in exceptions were designed into three categories:
  a. Those that are enabled by default but could be disabled by user code
  b. Those that are disabled by default but could be enabled by user code
  c. Those that are always enabled

`--->` **SHOW program listing (p. 543)**

- Evaluation

- The design is powerful and flexible, but has the following problems:
  a. Dynamic binding of exceptions to handlers makes programs difficult to write and to read
  b. The continuation rules are difficult to implement and they make programs hard to read

Chapter 13

Ada Exception Handling

Def: 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

- *Handler form:*

```
exception
   when exception_name {| exception_name} =>
    statement_sequence
   ...
   when ...
   ...
   [when others =>
    statement_sequence ]
```

- Handlers are placed at the end of the block or unit in which they occur

Chapter 13

- 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

  1. Procedures - propagate it to the caller

  2. Blocks - propagate it to the scope in which it appears

  3. Package body - propagate it to the declaration part of the unit that declared the package (if it is a library unit (no static parent), the program is terminated)

  4. 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)

Chapter 13

- User-defined Exceptions:

exception_name_list `: exception;`

`raise` **[exception_name]**

(the exception name is not required if it is in a handler--in this case, it propagates the same exception)

- Exception conditions can be disabled with:

`pragma SUPPRESS(`**exception_list**`)`

- Predefined Exceptions:

`CONSTRAINT_ERROR` **- index constraints, range constraints, etc.**

`NUMERIC_ERROR` **- numeric operation cannot return a correct value, 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**

---> SHOW program (pp. 549-550)

- Evaluation

- The Ada design for exception handling embodies the state-of-the-art in language design in 1980
- A significant advance over PL/I
- Ada was the only widely used language with exception handling until it was added to C++

C++ Exception Handling

- Added to C++ in 1990

- Design is based on that of CLU, Ada, and ML

Chapter 13

- Exception Handlers

- Form:

```
try {
 -- code that is expected to raise an exception
}
catch (formal parameter) {
 -- handler code
}
...
catch (formal parameter) {
 -- handler code
}
```

- `catch` is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique

- The formal parameter need not have a variable
  - It can be simply a type name to distinguish the handler it is in from others

- The formal parameter can be used to transfer information to the handler

- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

Chapter 13

- Binding Exceptions to Handlers

- Exceptions are all raised explicitly by the statement:

  `throw` **[expression];**

- The brackets are metasymbols

- A `throw` without an operand can only appear in a handler; when it appears, it simply reraises the exception, which is then handled elsewhere

- The type of the expression disambiguates the intended handler

- Unhandled exceptions are 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 program is terminated

Chapter 13

- *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

  - Functions can list the exceptions they may raise
    - Without a specification, a function can raise any exception

---> SHOW program listing (pp. 553-554)

- *Evaluation*

  - It is odd that exceptions are not named and that hardware- and system software-detectable exceptions cannot be handled

  - Binding exceptions to handlers through the type of the parameter certainly does not promote readability

Chapter 13

Java Exception Handling

- Based on that of C++, but more in line with OOP philosophy

- All exceptions are objects of classes that are descendants of the `Throwable` **class**

- The Java library includes two subclasses of `Throwable`**:**

1. `Error`

- Thrown by the Java interpreter for events such as heap underflow

- Never handled by user programs

2. `Exception`

- User-defined exceptions are usually subclasses of this

- Has two predefined subclasses, `IOException` **and** `RuntimeException` **(e.g.,** `ArrayIndexOutOfBoundsException` **and** `NullPointerException`

Chapter 13

- Java Exception Handlers

- **Like those of C++, except every** `catch` **requires a named parameter and all parameters must be descendants of** `Throwable`

- **Syntax of** `try` **clause is exactly that of C++**

- **Exceptions are thrown with** `throw`**, as in C++, but often the** `throw` **includes the** `new` **operator to create the object, as in:**

```
throw new MyException();
```

- **Binding an exception to a handler is simpler in Java than it is in 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**

- **An exception can be handled and rethrown by including a** `throw` **in the handler (a handler could also throw a different exception)**

Chapter 13

- Continuation

- If no handler is found in the `try` construct, the search is continued in the nearest enclosing `try` construct, etc.

- If no handler is found in the method, the exception is propagated to the method s caller

- If no handler is found (all the way to `main`), the program is terminated

- To insure that all exceptions are caught, a handler can be included in any try construct that catches all exceptions

  - Simply use an `Exception` class parameter

    - Of course, it must be the last in the try construct

- Other Design Choices

- The Java `throws` clause is quite different from the `throw` clause of C++

Chapter 13

- **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:**

  1. **Listed in the `throws` clause, or**

  2. **Handled in the method**

- **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:**

  1. **Catch and handle the exception**

  2. **Catch the exception and throw an exception that is listed in its own `throws` clause**

  3. **Declare it in its `throws` clause and do not handle it**

Chapter 13

---> SHOW Example program (pp. 558-559)

- *The* `finally` *Clause*

- Can appear at the end of a `try` construct

- Form:

```
finally {
...
}
```

- *Purpose*: To specify code that is to be executed, regardless of what happens in the `try` construct

- A `try` construct with a finally clause can be used outside exception handling

```
try {
  for (index = 0; index < 100; index++) {

    if ( ) {
      return;
    }  //** end of if
}  //** end of try clause
finally {

}  //** end of try construct
```

Chapter 13

- Evaluation

- The types of exceptions makes more sense than in the case of C++

- The `throws` clause is better than that of C++ (The `throw` clause in C++ says little to the programmer)

- The `finally` clause is often useful

- The Java interpreter throws a variety of exceptions that can be handled by user programs

