14. Exception Handling

14.1 Intro to Exception Handling

- 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)

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

- The special processing that may be required after detection of an exception is called exception handling

- The exception handling code unit is called an exception handler
14.1 Intro to Exception Handling (continued)

- 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:
  - 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
- Advantages of Built-in Exception Handling:
  - Error detection code is tedious to write and it clutters the program
  - Exception propagation allows a high level of reuse of exception handling code

14.1 Intro to Exception Handling (continued)

- Design Issues for Exception Handling:
  - How and where are exception handlers specified and what is their scope?
  - How is an exception occurrence bound to an exception handler?
  - Where does execution continue, if at all, after an exception handler completes its execution?
  - 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?
14.1 Intro to Exception Handling (continued)

- Exception handling-control flow

![Diagram of exception handling control flow]

14.4 Exception Handling in C++

- Added to C++ in 1990
- Design is based on that of CLU, Ada, and ML
- Exception Handlers
  - Form:
    ```cpp
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
14.4 Exception Handling in C++ (continued)

- 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

- 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 (the `throw` clause)

- See program listing (pp. 577-578)

- 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
14.5 Exception Handling in Java

- 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`
  - `Error`
    - Thrown by the Java interpreter for events such as heap underflow
    - Never handled by user programs
  - `Exception`
    - User-defined exceptions are usually subclasses of this
    - Has two predefined subclasses, `IOException` and `RuntimeException` (e.g., `ArrayIndexOutOfBoundsException` and `NullPointerException`)

14.5 Exception Handling in Java (continued)

- 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:
    ```java
type new MineException();
```
  - 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)
14.5 Exception Handling in Java (continued)

- **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

14.5 Exception Handling in Java (continued)

- **Other Design Choices**
  - The Java `throws` clause is quite different from the `throw` clause of C++
  - 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
  - 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
14.5 Exception Handling in Java (continued)

- See Example program (pp. 582-583)
- The `finally` Clause
  - Can appear at the end of a `try` construct
  - 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
- 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