Programming Languages: Exception Handling

Ryan Stansifer

Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

5 April 2020
Calvin and Hobbes
Overview: Exception Handling

- Goals
- Blocks not resumption
- Propagation
- Features in different languages
- “Final wishes”
What are exceptions?

Bad things happen occasionally at runtime.

**arithmetic:** $\div 0$, $\sqrt{-9}$

**environmental:** no space, malformed input

**undetectable:** subscript out of range, value does not meet prescribed constraint

**others:** can’t invert a singular matrix, can’t pop an element from an empty stack

Both hardware-detectable and software-detectable conditions. Which is which may depending on language implementation, hardware, etc.
Exceptions

Exceptions are said to be *raised* or *thrown* at the moment of detection, and are said to be *handled* or *caught* at the point when normal execution resumes.

Raising an exception halts normal execution abruptly and alternative statements are sought to be executed, possibly the program terminates.

Usually exceptions can be named; there are predefined exceptions and user-defined exceptions.
Exception Handling Goals

1. Separate normal flow of control from error handling
2. Incur no execution-time penalty
Return-Code

if ((fd=open(name, O_RDONLY))==-1) {
    fprintf (stderr, "Error %d opening file", errno);
    exit();
}

See also the C program from Stevens.

main.c
A programmer can program defensively in a programming language without support for exceptions. But there are many problems with ad hoc approach:

- Easy to ignore, hence error prone
- Poor modular decomposition
- Hard to test such programs
- Inconsistency — sometimes null, sometimes -1
- No additional info

Exceptions

In general, an exception should be used when there is an inability to fulfill a specification. Why not a precondition and an assert statement? Some cases are difficult to describe or detect; some cases are too rare concern the ordinary users of the code.

Indeed, assertions are often the same things as exceptions: a runtime signal of problem stopping the program from committing some unintended actions.
Exceptions in PL

- **Exception propagation**, resumption versus block-structured
- Named exceptions, user-defined and predefined exceptions
- Exceptions as values or separate entities
- Exceptions with arguments (additional information)
- Catch or declare
- Handling exceptions out of the scope of their names, one handler for all exceptions, re-raising exceptions in a handler
- Final wishes, return and exit as exceptions

Sebesta, Chapter 14: Exception Handling, page 557.
Blocks With Handlers

In Ada, Modula-3, C++, Java, and SML a set of handlers watches over a block of code. When an exception is raised somewhere (perhaps in a subroutine call) in the block, execution stops at that point and a handler is sought. If one of the handlers is invoked after a successful search, the code of the handler is executed (naturally), and then the entire block ends as if no exception were ever raised.

TRY
   (* a section of executable statements *)
EXCEPT
   | One_Exception => (* One handler *)
   | Another_Exception => (* Another handler *)
END
Exception Propagation

There is no attempt at going back and finishing remaining actions in the block. PL/I tried implicit resumption after handling exceptions, but this is confusing. The programmer can still program any sort of resumption imaginable by careful use of block-structured exception handling,

*Exception propagation*. Modern languages all take the same approach to *exception propagation*, the search for the place to resume normal execution: follow the dynamic chain of block activations. This is obviously correct: the caller who asked for the service should hear of the failure. On the other hand, declarations of exception names follow the same static scope rules as all identifiers.
• Ada
• C++
• Modula-3
• Java
• C#
• Python
• SML

See also `setjmp/longjmp` in C and `callcc` in functional languages.

*No* exception handling in Haskell as a matter of purity.
Examples

except/Pre.java – simple example with predefined exceptions
except/main.adb – simple example with predefined exceptions
except/calls.adb – recall runtime stack
except/calls2.adb – dynamic propagation
except/propagation.adb – much the same with print statements
except/propagation2.adb – others, multiple exceptions handler
except/Value.java – exceptions are classes in Java
except/Declare.java – catch or declare; exceptions important to specifications
except/Trace.java – stack trace is useful; don’t “swallow” exceptions
Exceptions in Python are similar to Java. Like Java they are classes.

try:
    # block of code
    pass
except NameError:
    pass
except KeyError:
    pass

Exceptions can be built-in exceptions (e.g., IndexError, SyntaxError, user-defined exceptions, or user-defined classes.)
Exceptions in Python

Exceptions are like classes as in Java.

```
try
    pass
except KeyError as err:
    # Bind the exception instance to 'err'
except (AttributeError, TypeError, SyntaxError):
    # Multiple (and subclass, too!)
except:
    # wildcard; all other exceptions
    raise  # reraise the exceptoin
else:
    # no exception raised
```

A raise statement without an exception name and in a handler reraises the current exception.
Exception Occurrences

Just exactly what is an exception. What exactly is an exception occurrence? When is one generated? If two exceptions have the same name are they the same?

- `except/occur.adb` – exceptions are usually global signals
- `except/recurse.adb` – but they can be declared locally
- `except/Recurse.java` – local scope
- `except/Recurse2.java` – local scope
Exception Scope

Because exceptions propagate along the dynamic chain and their scope follows the static structure of the program, it is possible for an exception to propagate out of its scope.

- except/out_of_scope.adb
- except/madness.adb (uses packages)
Final Wishes

Even if a subprocedure cannot fix an exceptional case and must propagate the problem back up to its caller, it still may wish to undo the mess it is responsible for before relinquishing control for good. Exception handling seems like a good way to handle “final wishes.”

- except/final.adb – others clause
Final Wishes

Modula-3:

TRY
    (* executable statements *)
FINALLY
    (* final wishes *)
END

Java has one combined statement:

try {
    // executable statements
} catch (Exception e) {
    // handler for ‘e’
} finally {
    // 1. normal, 2. caught exc, 3. uncaught
    // exc, 4. break, continue, 5. return
    // final wishes
}
Java's try is Combination

Modula-3:

TRY
    TRY
        (* executable statements *)
    EXCEPT
        (* handlers *)
    END
FINALLY
    (* final wishes *)
END
It is combined in Python

```python
try:
    # executable statements
except:
    # handlers
finally:
    # final wishes
```
try:
    print(1)
    #raise NameError #1,3,6
    #raise KeyError #1,4,6
    print(2)   #1,2,5,6
except NameError:
    print(3)
except:
    print(4)
else:
    print(5)
finally:
    print(6)
print("end")
finally Clause

There is some confusion with the `finally` clause. The code in the `finally` clause ought not change the kind of control flow: normal, exceptional, return, or break. In other words, if the block is getting ready to break out of a loop it ought not to return instead. Or, if the block is getting ready to raise an exception it ought not to break out of a loop instead.

- `except/FinReturn.java` – Java warns