

Exception handling

Amer Diwan

What is exception handling

- A civilized way of dealing with exceptional situations
 - When a code detects an error, it raises an exception
 - Code that knows how to handle the exception can declare “handlers” for the exception

An example in Psuedo-Modula-3

```
EXCEPTION explosion, error
PROCEDURE foo() =
  TRY
    baz();
  EXCEPT
    explosion => PRINT "nice knowing you"
    | error => ...
  END
PROCEDURE baz() =
  bar();
PROCEDURE bar() =
  RAISE explosion;
```

Important features of exception handling

- It is a non-local jump
 - An exception may transfer control to code that is not even within the enclosing procedure
- Code that doesn't know how to handle an error, does not need to explicitly propagate the error condition
 - Contrast with "return success" codes
- Can be used as "structured" gotos
 - but it is discouraged

Variations in exception handling: How are exceptions raised

- **Explicitly** (E.g., Java, Modula-3, ...)
 - User throws exceptions explicitly by having a “raise” statement
- **Implicitly** (E.g., Java, PL/1)
 - E.g., on a divide-by-zero error, the language defines that an exception will be raised
 - + **Uniform: all errors map to exceptions**
 - **May degrade performance**
 - **Need checked/unchecked distinction**

Variations in exception handling: How exceptions are declared

- Exceptions can be a **special type** (e.g., Modula-3)
- Exceptions can be a **subclass of a designated type** (e.g., Java, C++)
 - + **More uniform**
 - **An exception may be handled by multiple catch statements: need a resolution mechanism**

Variations in exception handling: How are raised exceptions bound to a handler?

- Handlers have a **static scope** (Modula-3, Java, C++...)
 - An exception is handled by a handler enclosing it, or
 - it is handled by a handler enclosing the call to the routine in which the exception is raised
- Handlers have a **dynamic scope** (e.g., PL/1)

Example of exception handler with dynamic scope

- **ON CONDITION** explosion **PRINT** “nice knowing you”
BEGIN
 IF b is true **THEN**
 ON CONDITION explosion **PRINT** “gotcha”
 SIGNAL explosion;
 END
 SIGNAL explosion
- **Advantages: very powerful**
- **Disadvantages: hard to understand**

Variations in exception handling: What happens after a handler executes

- **Termination model** (Java, Modula-3, C++...)
 - Execution continues after the handler
- **Resumption model** (PL/1)
 - Execution continues after the RAISE
 - Complex
 - Perhaps more directly supports error recovery

Exception handling in Java

- Exceptions may be raised implicitly or explicitly
- Exceptions are objects
- Exception handlers have “static” scope
- Termination model
- Combines **TRY-CATCH** with **TRY-FINALLY**
- Distinction between checked and unchecked exceptions

Checked and unchecked exceptions

- The language guarantees at compile time that either
 - there is an enclosing handler for every throw of a checked exception, or
 - the enclosing procedure names all potentially raised checked expressions (or their supertypes) in its “throws” clause

Example 1

- Class aCheckedExc extends Exception { ... }
- ```
int try1() {
 try {
 throw aCheckedExc;
 }
 catch (aCheckedExc e) { ... }
}
```

## Example 2

- Class aCheckedExc extends Exception { ... }
- int try1() {  
    try {  
        throw aCheckedExc;  
    }  
    catch (Exception e) { ... }  
}

## Example 3

- Class aCheckedExc extends Exception { ... }
- Class anotherChExc extends Exception { ... }
- int try1() {  
    try {  
        throw aCheckedExc;  
    }  
    catch (anotherChExc e) { ... }  
}

## Example 4

- Class aCheckedExc extends Exception { ... }
- Class anotherChExc extends Exception { ... }
- int try1() throws aCheckedExc {  
    try {  
        throw aCheckedExc;  
    }  
    catch (anotherChExc e) { ... }  
}

## Example 5

- Class aCheckedExc extends Exception { ... }
- Class anotherChExc extends Exception { ... }
- int try1() throws aCheckedExc {  
    try {  
        throw aCheckedExc;  
    }  
    catch (anotherChExc e) { ... }  
}
- int try2() {  
    try1();  
}

## Why the distinction between checked and unchecked

- In Java, many operations may raise exceptions
  - E.g., arithmetic, pointer dereferences, I/O
  - It would be very bulky to require the programmer to put all of them in the throws clause or declare handlers for all of them
    - E.g., errors throw unchecked exceptions
  - Programmers can define unchecked exceptions too, but are encourage to use only checked exceptions

## Exception handling in Modula-3

- Exceptions are raised explicitly
- Exceptions are a separate “second class” type
- Exception handlers have “static” scope
- Termination model
- Separate TRY-EXCEPT and TRY-FINALLY
- All exceptions are checked

## What is a TRY-FINALLY and why is it there?

- Java combines TRY-FINALLY into TRY-CATCH
- **Semantics:**
  - Execute the try block
  - Execute the finally block. If finally block does not raise an exception, reraise the exception raised by the try block
- TRY-FINALLY is a way to allow programmers to give “**finalization**” or “**cleanup**” code

## Example of TRY-FINALLY

- TRY  
  f\_input := OPEN\_FILE("input");  
  ...  
  FINALLY  
  CLOSE(f\_input)  
  END
- **The file is closed even if the try block raises an exception**

## Next lecture: implications of exceptions

- How to implement exception handling
- What are the implementation implications for how languages incorporate exceptions