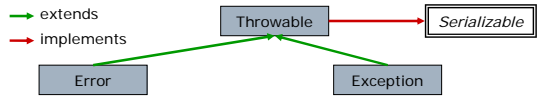


Exceptions and Assertions

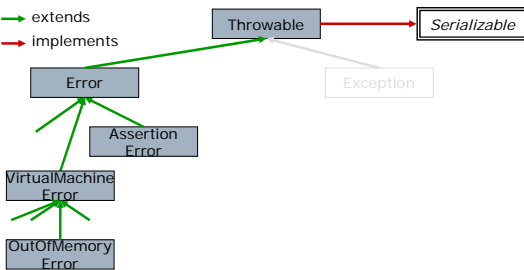
Lecture 7

Throwable Hierarchy

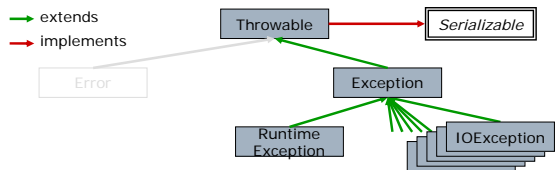


- Error
 - Internal problems or resource exhaustion within VM
 - Thrown by Java SDK methods or VM itself
 - "unrecoverable"
 - Beyond the program's ability to control or handle
 - Little you can do: abort the program
- Exceptions
 - Problems within the application
 - Thrown by Java SDK or programmer application
 - "recoverable" (maybe)
 - Corrective actions within program may be possible

Error Hierarchy

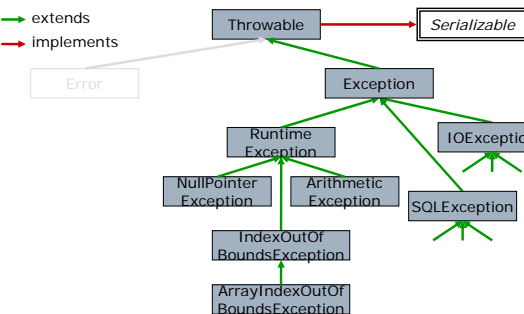


Exception Hierarchy

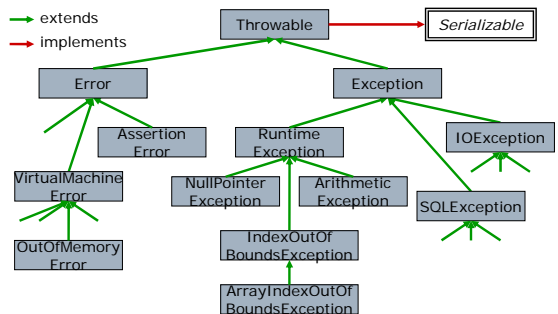


- Exceptions derived from RuntimeException
 - Examples: bad cast, out-of-bounds array access
 - Happen because an error exists in your program
 - "Your fault"
- Exceptions that do not derive from RuntimeException
 - Example: trying to open a malformed URL
 - Happen because of externalities (the outside world)
 - "Not your fault"

Exception Hierarchy



Throwable (sub)Hierarchy



Best Practices: When to Use

- Reserve for “unexpected” or “unusual” behavior
 - Good: to signal file does not exist
 - Bad: to signal end of file
 - Terrible: to signal end-of-line (ie for control flow)
- Particularly appropriate when client can not guarantee the requires clause of a method
 - Example: existence of a file. First checking for the file does not help because file could be deleted after check but before method is called
- Concurrency of world with which program interacts means that some requires clauses can not be unilaterally guaranteed by client, as required by design-by-contract

Syntax of Try/Catch Blocks

- Vocabulary: Exceptions (and Errors) are
 - “thrown” by a component implementation
 - “caught” by a client
- In client, a try/catch block is used to catch


```
try {
    statements
} catch(exceptionType1 identifier1) {
    handler for type1
} catch(exceptionType2 identifier1) {
    handler for type2
} . . .
```
- If nothing is thrown during execution of the statements in the try clause:
 - Try clause finishes successfully
 - All catch clauses are ignored

Catching a Throwable

- If something is thrown during execution of the statements in the try clause:
 1. The rest of the code in the try block is skipped
 2. The catch clauses are examined top to bottom for the first matching catch (based on type compatibility)
 - catch (SomeException e) matches subtypes of SomeException
 3. If an appropriate catch clause is found:
 - Body of catch clause is executed
 - Remaining catch clauses are skipped
 4. If no such a catch clause is found:
 - The exception is thrown to outer block, which is either
 - A try block (that potentially handles it, in same manner)
 - A method body (resulting in it being thrown to its client)
- Consequence:
 - A catch clause for a *subclass* of SomeException cannot follow a catch clause for SomeException

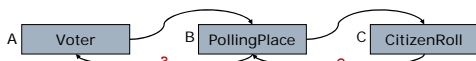
Best Practice: Specific Catching

- Can be tempting to bundle all exception catching into one clause


```
try {
    . . .
} catch (Exception e) {
    . . .
}
```
- Usually, however, properly handling an exception is more type-specific
- Therefore, catch each (relevant) exception type separately
- Similar concern as “coding to the interface”
 - An exception's static type should be specific enough to provide information needed by the client for recovery, but not more

Handling a Throwable

- Implementations are layered
 - Objects are both *clients* and *components*



- How should B handle the throwable e from C?
- Three choices for body of catch clause in B
 - Handle the exceptional situation, effectively masking the issue from A
 - Pass e on to A
 - Create and throw a new throwable, e2, for A
 - e might not make sense to A, which doesn't know about C
 - Exception chaining can link e2 to its cause (e)

Best Practices: Never Suppress

- An empty catch clause is a red flag
 - Usually indicates laziness


```
try {
    . . .
} catch (IOException e) { }
```
 - There are rare instances where “no action” actually does properly handle the situation
 - If so, document code with clear justification
- More subtle: catch clause that logs


```
try {
    . . .
} catch (IOException e) {
    e.printStackTrace();
}
```

 - This also effectively hides the exception without actually having handled it

Creating new Exception Classes

- ❑ Throwable hierarchy can be subclassed to create new application-specific exception types

```
class TemperatureException extends Exception { ... }
```
- ❑ Inherit Throwable's String for informal description

```
t = new TemperatureException("Engine overheated");
throw (new EOFException("File too long"));
```
- ❑ Why create new exception types?
 - New class can declare new fields and methods
 - ❑ Can provide more structured information to client
 - ❑ Eg TemperatureException includes value of temperature that triggered the exception
 - Client catch clause is determined by exception type
 - ❑ Can distinguish a problem for which distinct handling logic will (likely) be required on client's side
 - ❑ Eg TemperatureExceptions will require modifying the engine's temperature before repeating the operation

Best Practices: New Exceptions

- ❑ Use standard exceptions if possible
 - Good litmus test: are particular methods needed to aid in recovery?
- ❑ Prefer checked exceptions
 - Extend Exception, not Error or RuntimeException
- ❑ Naming convention
 - Class name ends in "Exception" (see SDK)

Catching Checked Exceptions

- ❑ Choices for body of catch clause corresponding to checked exception e
 - Mask the problem by handling the exceptional situation
 - Rethrow e on up to client and *declare exception type in signature (throws)*
 - Create and throw a new throwable, e2, on up to client
 - ❑ e2 could be checked, in which case it must be declared in signature
 - ❑ e2 could be unchecked, in which case it should not be declared in signature

Exception Chaining

- ❑ Creating and throwing a new throwable in the body of a catch clause
 - Used to change the type of exception
 - Maps failure to mode that makes sense to client
- ❑ Original exception, however, might still be useful
 - Example: debugging by looking at the trail of cascading exceptions
- ❑ Chaining: a Throwable has a cause (another throwable)

```
catch (SQLException e) {
    ServletException se = new ServletException();
    se.setCause(e);
    throw se;
}
```

 - At client (or higher), original exception can be retrieved

```
catch (ServletException e) {
    Throwable cause = e.getCause();
}
```

Assertions

- ❑ An assertion is a statement that should always evaluate to true
- ❑ Keyword: assert
 - `assert eval-expr [: detail-expr];`
`assert tail.next == null : "No list end";`
- ❑ If the eval-expr does not evaluate to true, an AssertionError is thrown
 - An error since an assertion violation is unrecoverable
 - detail-expr can be either
 - ❑ A String (becomes the informal description)
 - ❑ A Throwable (gets chained as the cause)

Roles of Assertions

- ❑ Checking representation invariants
 - At the end of the constructor
 - At the end of every method
- ❑ Checking requires
 - Defensive programming: check assumptions
- ❑ Checking ensures
 - Verify implementation has delivered promised behavior
- ❑ Checking flow-of-control
 - Example: assert false at a point that should never be reached
 - Throwing the AssertionError directly usually preferred to assert false
- ❑ Checking loop invariants

Turning Assertions On and Off

Computer Science and Engineering @ The Ohio State University

- Assertions can be enabled or disabled
 - Default: disabled
 - Class-level and package-level control
- Command-line arguments
 - -ea (-da) to enable (disable) all assertions
 - -ea:edu.osu.Tester to enable only in class Tester
 - -ea:edu.osu... to enable only in package edu.osu
- Never have side-effects
 - Example: `assert i++ < max;`
 - Program behavior changes if assertions are on/off
- (Poor) argument for disabling assertions: performance
 - Benefit is likely to be negligible
 - And robustness always outweighs speed

Best Practice: Public Methods

Computer Science and Engineering @ The Ohio State University

- Widely-accepted Java convention:
 - Never use `assert` to check *requires* of *public* methods
 - Prefer a `RuntimeException` (eg `IllegalArgumentException`)
 - OK for *requires* of private methods (and all *ensures*)
- But a violation of *requires* clause is not recoverable (by client), so it should be an `Error`, not an `Exception`
 - Really, these contract checks belong in a separate component (a checking wrapper)
 - But without better linguistic support for such things, assertions will have to do
- Contrary to Sun recommendations, use `asserts` liberally, even for public methods