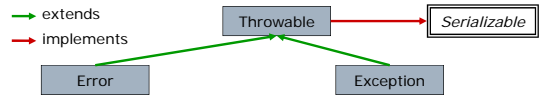


Exceptions

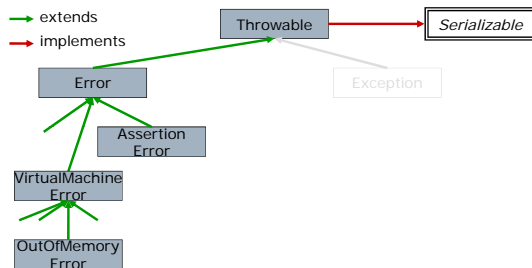
Lecture 15

Throwable Hierarchy

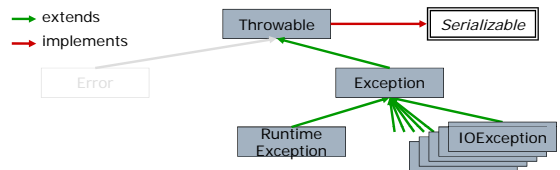


- Error
 - Internal problems or resource exhaustion within VM
 - Thrown by Java SDK methods or VM itself
 - "unrecoverable"
 - Beyond the program's ability to control or handle
 - Little you can do: abort the program
- Exceptions
 - Problems within the application
 - Thrown by Java SDK or programmer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

Error Hierarchy

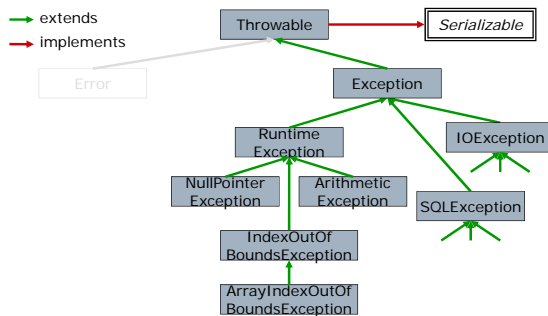


Exception Hierarchy

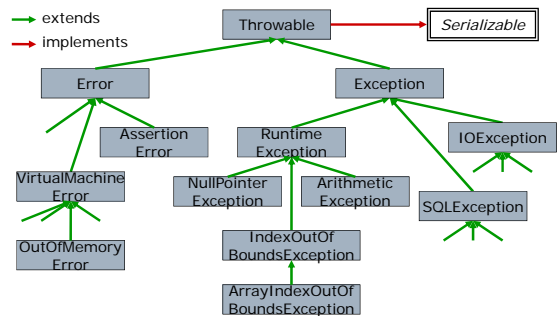


- Exceptions derived from RuntimeException
 - Examples: bad cast, out-of-bounds array access, dereferencing a null pointer
 - Happen because an error exists in your program
 - "Your fault"
- Exceptions that do not derive from RuntimeException
 - Example: trying to open a malformed URL
 - Happen because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

Exception Hierarchy



Throwable (sub)Hierarchy



Good Practice: 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
- More on this in a future lecture...

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

Example

```
String filename = ...
try {
    //Create the file
    File f = new File(filename);
    if (f.createNewFile()) {
        ... //file creation succeeded
    }
    else {
        ... //file already exists
    }
    ... //either way, can use f here
} catch (IOException e) {
    //deal with IO problem (eg disk full)
} catch (SecurityException e) {
    //some permission problem
}
```

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

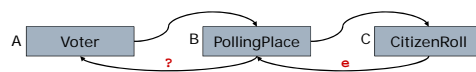
Good Practice: Specific Catching

- 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 declared 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
 - But e might not make sense to A, which does not even know about C!
 - Create and throw a new throwable, e2, for A
 - Exception *chaining* can link e2 to its cause (e)

Good Practice: Never Suppress

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


```
try {
    . . .
} catch (IOException e) { }
```
 - There are *very 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

(Poor) Example

```
String filename = "/nosuchdir/somefile";
try {
    //Create the file
    new File(filename).createNewFile();
} catch (IOException e) {
    //Display the exception that occurred
    System.out.println("Unable to create "
        + filename + ": " + e.getMessage());
}
```

- ❑ Output


```
Unable to create /nosuchdir/somefile:
The system cannot find the path
specified
```

Finally Clause

- ❑ Some actions should be performed whether or not exceptions occur
 - Example: releasing resources such as database or network connections
- ❑ Syntax

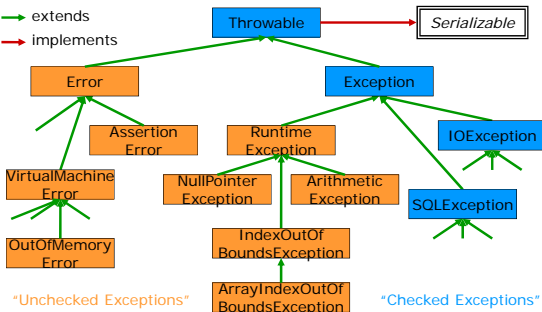

```
try {
    . . .
} catch (IOException e) {
    . . .
} finally {
    . . . //always executed
}
```
- ❑ The finally clause is executed:
 - After the try clause in the case of normal execution
 - After the catch clause in the case of an exception

Checked vs Unchecked Exceptions

- ❑ Unchecked Exceptions
 - Ubiquity: Possible sources (statements) are common
 - ❑ Examples: any dereference could be of null, any memory allocation could exhaust memory
 - Condition could arise in practically any method
 - Consequently, all methods allowed to throw them
- ❑ Checked exceptions
 - Operation-specific
 - ❑ Examples: working with file system or network
 - Conditions could arise in limited number of methods
 - A method can only throw (checked) exceptions explicitly declared in its signature


```
Image load() throws IOException, EOFException {...}
```
 - A client must catch all *declared* checked exceptions
 - These last two requirements are checked by compiler

Distinguishing Checked & Unchecked



Rule

- ❑ Unchecked exceptions are:
 - Subclasses of Error, or
 - Subclasses of RuntimeException
- ❑ The rest are checked exceptions

Throwing Throwables

- To signal an exceptional situation
 - Component creates a new throwable object
 - Component throws it with throws keyword
- Syntax

```
String readData(Scanner in) throws EOFException {
    ...
    while(. . .) {
        if (!in.hasNext()) { //EndOfFile encountered
            if(n < len) {
                throw (new EOFException("File too short"));
            }
        }
        ...
    }
    return s;
}
```
- For checked exceptions: Dynamic type of the thrown exception must be subclass of an exception type declared in method signature

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

Good Practice: 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

- Body of a catch clause often creates and throws a new throwable
 - Used to change the type of exception
 - Map failure to a 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();
}
```

Summary

- Throwable hierarchy
 - Errors, exceptions (& run-time exceptions)
 - Checked vs unchecked throwables
- Mechanics
 - Try/catch block
 - Declaration in method signatures
 - Exception chaining