Software Development (16302180)

Mustafa MISIR

mmisir@nuaa.edu.cn

http://mustafamisir.github.io/sd.html

Office 230 @ CCST (No specific office hours)

WeChat Software Development discussion group

Software Development

Week 5: High-Quality Code Development

Defensive Programming (continue)

Outline

- Invalid Inputs
- Assertions
- Error-Handling
- Exceptions

Error-Handling

- Assertions are used to handle errors that should never occur in the code. How do you handle errors that you do expect to occur?
- Depending on the specific circumstances, you might want to
  - return a neutral value,
  - substitute the next piece of valid data,
  - return the same answer as the previous time,
  - substitute the closest legal value,
  - log a warning message to a file,
  - return an error code,
  - call an error-processing routine
  - or object, display an error message, or shut down—or you might want to use a combination of these responses.

Error-Handling

- *Return a neutral value*
- Sometimes the best response to bad data is to continue operating and simply return a value that's known to be harmless.
  - A numeric computation might return 0.
  - A string operation might return an empty string, or a pointer operation might return an empty pointer.
  - A drawing routine that gets a bad input value for color in a video game might use the default background or foreground color.
  - A drawing routine that displays x-ray data for cancer patients, a neutral value?
    - however, would not want to display a "neutral value."
    - In that case, you'd be better off shutting down the program than displaying incorrect patient data.

Error-Handling

- *Substitute the next piece of valid data*
- When processing a stream of data, some circumstances call for simply returning the next valid data.
- If you're reading records from a database and encounter a corrupted record, you might simply continue reading until you find a valid record.
- If you're taking readings from a thermometer 100 times per second and you don't get a valid reading one time, you might simply wait another 1/100th of a second and take the next reading.

Error-Handling

- *Return the same answer as the previous time*
- If the thermometer-reading software doesn't get a reading one time, it might simply return the same value as last time.
  - Depending on the application, temperatures might not be very likely to change much in 1/100th of a second
- In a video game, if you detect a request to paint part of the screen an invalid color, you might simply return the same color used previously.
- But if you're authorizing transactions at a cash machine, you probably wouldn't want to use the "same answer as last time"—that would be the previous user's bank account number!

Error-Handling

- *Substitute the closest legal value*
- In some cases, you might choose to return the closest legal value. This is often a reasonable approach when taking readings from a calibrated instrument
- The thermometer might be calibrated between 0 and 100 degrees Celsius, for example. If you detect a reading less than 0, you can substitute 0, which is the closest legal value.
- Cars use this approach to error handling whenever going back. Since a speedometer doesn't show negative speeds, when it simply shows a speed of 0—the closest legal value.

Error-Handling

- *Log a warning message to a file (a must-have approach)*
- When bad data is detected, you might choose to log a warning message to a file and then continue on.
- This approach can be used in conjunction with other techniques like substituting the closest legal value or substituting the next piece of valid data.
- If you use a log, consider whether you can safely make it publicly available or whether you need to encrypt it or protect it some other way.
  - Define different levels: e.g. debug, test … (information hiding)

Markers | Properties | Data Source Explorer | Snippets | Console

Tomcat v6.0 Server at localhost [Apache Tomcat] C:₩Program Files₩Java₩jre6₩bin₩javaw.exe (2009. 9. 13. 오후 2:07:28)

```
[DEBUG] (SqlMapClientTemplate.java) - Opened SqlMapSession [com.ibatis.sqlmap.engine.impl.SqlMapSessionImpl@15bfdbd] for iBATIS operation
[DEBUG] (JakartaCommonsLoggingImpl.java) - {conn-100002} Connection
[DEBUG] (DataSourceUtils.java) - Fetching JDBC Connection from DataSource
[DEBUG] (SqlMapClientTemplate.java) - Obtained JDBC Connection [jdbc:oracle:thin:@localhost:1521:XE, UserName=USER01, Oracle JDBC driver] for iBATIS (
[DEBUG] (JakartaCommonsLoggingImpl.java) - {conn-100002} Preparing Statement:    SELECT COUNT(0)    FROM MY_BOARD
[DEBUG] (JakartaCommonsLoggingImpl.java) - {pstm-100003} Executing Statement:    SELECT COUNT(0)    FROM MY_BOARD
[DEBUG] (JakartaCommonsLoggingImpl.java) - {pstm-100003} Parameters: []
[DEBUG] (JakartaCommonsLoggingImpl.java) - {pstm-100003} Types: []
[DEBUG] (JakartaCommonsLoggingImpl.java) - {rset-100004} ResultSet
[DEBUG] (JakartaCommonsLoggingImpl.java) - {rset-100004} Header: [COUNT(0)]
[DEBUG] (JakartaCommonsLoggingImpl.java) - {rset-100004} Result: [1]
[DEBUG] (DataSourceUtils.java) - Returning JDBC Connection to DataSource
[DEBUG] (SqlMapClientTemplate.java) - Opened SqlMapSession [com.ibatis.sqlmap.engine.impl.SqlMapSessionImpl@61cd2] for iBATIS operation
[DEBUG] (JakartaCommonsLoggingImpl.java) - {conn-100005} Connection
[DEBUG] (DataSourceUtils.java) - Fetching JDBC Connection from DataSource
[DEBUG] (SqlMapClientTemplate.java) - Obtained JDBC Connection [jdbc:oracle:thin:@localhost:1521:XE, UserName=USER01, Oracle JDBC driver] for iBATIS (
[DEBUG] (JakartaCommonsLoggingImpl.java) - {conn-100005} Preparing Statement:    select    *    from    (    select *    from    ( select * fr
[DEBUG] (JakartaCommonsLoggingImpl.java) - {pstm-100006} Executing Statement:    select    *    from    (    select *    from    ( select * fr
[DEBUG] (JakartaCommonsLoggingImpl.java) - {pstm-100006} Parameters: [1, 10]
[DEBUG] (JakartaCommonsLoggingImpl.java) - {pstm-100006} Types: [java.lang.Integer, java.lang.Integer]
[DEBUG] (JakartaCommonsLoggingImpl.java) - {rset-100007} ResultSet
[DEBUG] (JakartaCommonsLoggingImpl.java) - {rset-100007} Header: [SEQ, CATEGORY, TITLE, READ_COUNT, WRITER, REG_DATE]
[DEBUG] (JakartaCommonsLoggingImpl.java) - {rset-100007} Result: [28, 질문, 와우, 2, 쪼쪼, 2009-09-12 23:44:55.0]
[DEBUG] (DataSourceUtils.java) - Returning JDBC Connection to DataSource
[DEBUG] (AbstractCachingViewResolver.java) - Cached view [boardList]
[DEBUG] (DispatcherServlet.java) - Rendering view [org.springframework.web.servlet.view.InternalResourceView: name 'boardList'; URL [/WEB-INF/view/boa
[DEBUG] (AbstractView.java) - Rendering view with name 'boardList' with model {boardList=[kr.co.springboard.dto.BoardDTO@166bfd8], dto=kr.co.springboa
[DEBUG] (AbstractView.java) - Added model object 'dto' of type [kr.co.springboard.dto.BoardDTO] to request in view with name 'boardList'
[DEBUG] (AbstractView.java) - Added model object 'boardList' of type [java.util.ArrayList] to request in view with name 'boardList'
[DEBUG] (InternalResourceView.java) - Forwarding to resource [/WEB-INF/view/boardList.jsp] in InternalResourceView 'boardList'
[DEBUG] (DispatcherServlet.java) - Cleared thread-bound request context: org.apache.catalina.connector.RequestFacade@1226a77
[DEBUG] (FrameworkServlet.java) - Successfully completed request
[DEBUG] (AbstractApplicationContext.java) - Publishing event in context [org.springframework.web.context.support.XmlWebApplicationContext@19b46dc]: Se
[DEBUG] (AbstractApplicationContext.java) - Publishing event in context [org.springframework.web.context.support.XmlWebApplicationContext@15253d5]: Se
```

Error-Handling

- *Return an error code*
- You could decide that only certain parts of a system will handle errors. Other parts will not handle errors locally; they will simply report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error.

Error-Handling

- *Call an error-processing routine/object*
- Another approach is to centralize error handling in a global error-handling routine or error-handling object.
  - Debugging is easier
  - Can be re-used in other projects

Error-Handling

- *Display an error message wherever the error is encountered*
- This approach minimizes error-handling overhead; however, it does have the potential to spread user interface messages through the entire application-how to separate UI. *Tight coupling*
- Also, beware of telling a potential attacker of the system too much. Attackers sometimes use error messages to discover how to attack a system.

Error-Handling

- *Shut down*
- Some systems shut down whenever they detect an error.
- This approach is useful in safety-critical applications.
- For example, if the software that controls radiation equipment for treating cancer patients receives bad input data for the radiation dosage, what is its best error-handling response?
  - Should it use the same value as last time?
  - Should it use the closest legal value?
  - Should it use a neutral value?
  - In this case, shutting down is the best option. We'd much prefer to reboot the machine than to run the risk of delivering the wrong dosage.

Error-Handling: Correctness vs. Robustness

- *Correctness* means never returning an inaccurate result; returning no result is better than returning an inaccurate result.
- *Robustness* means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes.
- Safety-critical applications tend to favor correctness to robustness. It is better to return no result than to return a wrong result. e.g. the radiation machine
- Consumer applications tend to favor robustness to correctness. Any result whatsoever is usually better than the software shutting down. achine is a good example of this principle. e.g. MS Word

Outline

- Invalid Inputs
- Assertions
- Error-Handling
- Exceptions

Exceptions

- *An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions*.
- If code in one routine encounters an unexpected condition that it doesn't know how to handle, it throws an exception, essentially throwing up its hands and yelling,
  - "I don't know what to do about this—I sure hope somebody else knows how to handle it!"
- Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

```java
// demonstrates throwing an exception when a divide-by-zero occurs
public static int quotient( int numerator, int denominator )
   throws ArithmeticException
{
   return numerator / denominator; // possible division by zero
} // end method quotient
```

```java
catch ( InputMismatchException inputMismatchException )
{
   System.err.printf( "\nException: %s\n",
      inputMismatchException );
   scanner.nextLine(); // discard input so user can try again
   System.out.println(
      "You must enter integers. Please try again.\n" );
}
catch ( ArithmeticException arithmeticException )
{
   System.err.printf( "\nException: %s\n", arithmeticException );
   System.out.println(
      "Zero is an invalid denominator. Please try again.\n" );
} // end catch
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

JAVA

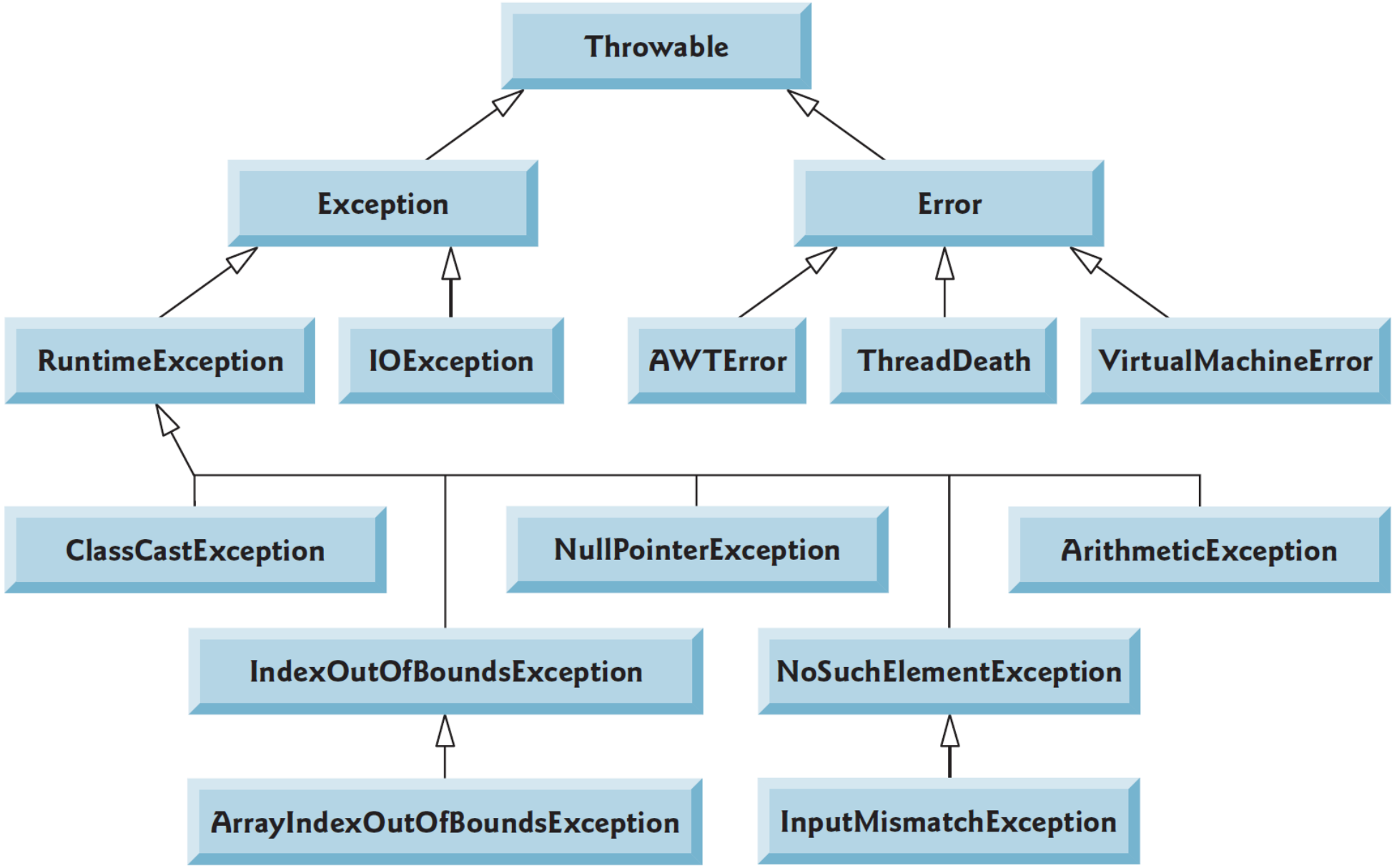

Fig. 11.3 | Portion of class `Throwable`'s inheritance hierarchy.

Exceptions

- ***Use exceptions to notify other parts of the program about errors that should not be ignored***
- The benefit of exceptions is their ability to signal error conditions in such a way that they cannot be ignored (Meyers 1996).
- Other approaches to handling errors create the possibility that an error condition can propagate through a code base undetected. Exceptions eliminate that possibility.

Exceptions

- **Separating Error-Handling Code from "Regular" Code**
- Exceptions provide the means to separate the details of what to do when something out of the ordinary.
- Error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Exceptions

- **Separating Error-Handling Code from "Regular" Code**
- At first glance, this function seems simple enough. Any issues you can think of?
  - What happens if the file can't be opened?
  - What happens if the length of the file can't be determined?
  - What happens if enough memory can't be allocated?
  - What happens if the read fails?
  - What happens if the file can't be closed?

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Exceptions

- To handle such cases, the readFile function must have more code to do error detection, reporting, and handling
  - Any comments?

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Exceptions

- There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter.
- Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory?
- It's even more difficult to say that the code continues to do the right thing when you modify the method three months after writing it.
- Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions

- Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere.
- If the readFile function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exceptions

- **Propagating Errors Up the Call Stack**
- The ability to propagate error reporting up the call stack of methods.
- Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program:
  - method1 calls method2, which calls method3, which finally calls readFile.

```
method1 {
     call method2;
}

method2 {
     call method3;
}

method3 {
     call readFile;
}
```

Exceptions

- *Suppose also that method1 is the only method interested in the errors that might occur within readFile*.
- Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Exceptions

- Only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

Exceptions

- **Grouping and Differentiating Error Types**
- Because all exceptions thrown within a program are objects (referring to OOP), the grouping or categorizing of exceptions is a natural outcome of the class hierarchy
- An example of a group of related exception classes in the Java platform are those defined in java.io — *IOException* and its descendants, e.g. FileNotFoundException.

Exceptions

- IOException vs FileNotFoundException?
- Exception handlers are to be as specific as possible.
- The main eason is a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy.

```
catch (FileNotFoundException e) {
    ...
}

catch (IOException e) {
    ...
}
```

Exceptions

- ***Throw exceptions at the right level of abstraction***
- A routine should present a consistent abstraction in its interface, and so should a class.
- *The exceptions thrown are part of the routine interface, just like specific data types are*.

Exceptions

- Anything wrong?
- it exposes some details about how it's implemented by passing the lower-level exception to its caller, *encapsulation is broken*

```
class Employee {
   ...
   public TaxId GetTaxId() throws EOFException {
      ...
   }
   ...
}
```

```
class Employee {
   ...
   public TaxId GetTaxId() throws EmployeeDataNotAvailable {
      ...
   }
   ...
}
```

Exceptions

- ***Include in the exception message all information that led to the exception***
- Be sure the message contains the information needed to understand why the exception was thrown.
- If the exception was thrown because of an array index error, be sure the exception message includes the upper and lower array limits and the value of the illegal index.

Exceptions

- ***Avoid empty* catch *blocks***
- Sometimes it's tempting to pass off an exception that you don't know what to do with, like this
  - either the code within the *try* block is wrong *because it raises an exception for no reason*, or the code within the *catch* block is wrong *because it doesn't handle a valid exception*.

Bad Java Example of Ignoring an Exception

```
try {
   ...
   // lots of code
  ...
} catch ( AnException exception ) {
}
```