Exception Handling

Objectives

- Introduce exception handling in Java
- To ensure error handling is correctly done

Contents

- Prerequisite Thinking
- Exception
- Exception Usage
- Traditional Error Handling vs. Exception
- Create Exceptions
- Throw Exceptions
- Guidelines

References

[1] Sun tutorial group, The Java Tutorial, *A practical guide for programmers*, `java.sun.com`

[2] Eckel B., Thinking in Java, Prentice Hall PTR, 1998, ISBN 0-13-659723-8

[3] Oser P. (POS), Exception Handling Guidelines, Internal Leaf project, 2000.

Prerequisite Thinking

- Errors occur in software programs.
- What really matters is what happens after the errors occur.
- Errors must be handled by developers.
- Errors should be reported to application users and the system administrators.
- Application should recover after errors. It should not just die.

Exception

- Definition: An exception is an event that occurs during the execution of a program that stops the normal flow of instructions.[1]
- Concept: Exceptions are the customary way in Java to indicate to a calling method that an abnormal condition has occurred.[3] An error, in a program, is an abnormal condition[1].
- Thus they should be shown up as exceptions

Exception

- Throwing Exception: when you want to stop the flow of instructions because of an abnormal condition, you can *throw* an exception. This exception is sent from the context of the method to that of its invokers.
- Catching Exception: When you are in the context of the invoker of a method that throws an exception and you want to handle it, you *catch* it. This exception is no more thrown up to your invokers/callers.

Exception Usage

- The calling method can do one of three things below with exceptions:[3]
  - It can call the method from within a `try { } catch {}` block and catch the exception to handle it.
  - It can decide not to handle the exception and declare that it throws itself the same exception via the `throws <ExceptionName>` clause.
  - It can catch the exception, do possibly some partial exception handling and then re-throw the same exception or throw another exception.

Exception Usage

```
Employee m_emp;
//…
public void modifyEmployee(){
  //…collect information from GUI to the object m_emp
  try{
    empService.modifyEmployee(m_emp);
  }catch(EmployeeModifiedByAnotherException emae){
    //Shows message to the user to reload the employee data
  }
}
```

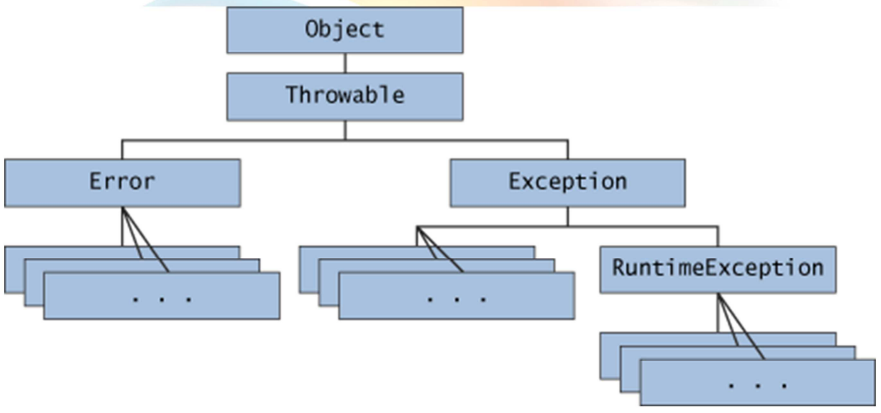
```
public void modifyEmployee(Employee emp) throws EmployeeModifiedByAnotherException{
  empDAO.updateEmployee(emp);
}
```

```
public void modifyEmployee(Employee emp) throws EmployeeModifiedByAnotherException{
  try{
    empDAO.updateEmployee(emp);
  }catch(EmployeeModifiedByAnotherException emae){
    //log this event into a file
    throws emae;
  }
}
```

Exception Types

- Checked and Unchecked Exceptions
- Checked Exceptions must be handled in one of the three ways above otherwise Java compiler complains.
- Unchecked Exceptions (Runtime Exceptions) can also be handled in the same ways. However the compiler does not require their handling.

Exception Hierarchy[3]



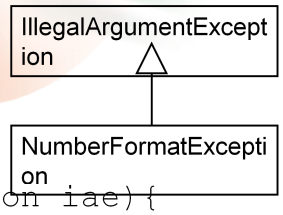
Error Class

- `java.lang.Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch.
  - **Example:** The `OutOfMemoryError` error shows no more memory for JVM to run
- A method is not required to declare in its `throws` clause any subclasses of `Error` that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

Catching Exceptions

- `catch(ExceptionName e){}` block is an exception handler. It allows developers to write code that handles the exception.
- One `try` can have many `catch` blocks.
- One thrown exception can match multiple `catch` blocks. However the first matching `catch` block is the only handler according to the order of the appearance of `catch` blocks in the code.

Catching Exceptions




```
try{
  Integer.partInt("XX");
}
catch(IllegalArgumentException iae){
  //Handle iae
}
catch(NumberFormatException  nfe){
  //This code is never called
}
```

Catching Exceptions

- Remarks:
  - Catch the most precisely specialized exception rather than its superclass.
  - `catch` block must not be empty
  - Order `catch` blocks according to the predicted number of the exception occurrence. The greatest is first. This is not applied to parent exceptions over their children.
  - Do not use `try catch` to perform the flow control.

```
Iterator i = empCollection.iterator();
Employee emp;
for(;;){
  try(
    emp = i.next();
    emp.program();
  }catch(NoSuchElementException e){
    break;
  }
}
```

```
Iterator I = empCollection.iterator();
Employee emp;
while(i.hasNext()){
  emp = i.next();
  emp.program();
}
```

Stack Trace

- Stack trace is the list of methods that consecutive called down to the one that throws the exception
- Stack trace can be visualizable
- Useful for debug and logging

Read Stack Trace

```
public class MyApp {
   public static void main(String[] args) {
        try {
            doSomething();
        } catch (MyNumberFormatException e) {
            e.printStackTrace();
        }
    }
    private static void doSomething() throws MyNumberFormatException{
        try {
            Integer.parseInt("xxx");
        } catch (NumberFormatException e) {
            throw new MyNumberFormatException("Not an integer", e);
        }
    }
}
class MyNumberFormatException extends Exception{
    public MyNumberFormatException(String msg, Throwable orgEx){
        super(msg, orgEx);
    }
}
```

Read Stack Trace

```
test.MyNumberFormatException: Not an integer
  at test.MyApp.doSomething(MyApp.java:28)
  at test.MyApp.main(MyApp.java:18)
Caused by: java.lang.NumberFormatException: For input string: "xxx"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:468)
  at java.lang.Integer.parseInt(Integer.java:518)
  at test.MyApp.doSomething(MyApp.java:26)
  ... 1 more
```

Exception vs. Error Code

- Separating Error Handling Code from "Regular" Code
- Propagating Errors Up the Call Stack
- Grouping Error Types and Error Differentiation

Error Handling Example[1]

- In pseudo-code, your function might look something like this:

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

- This code seems nice and enough.

Error Handling Example

- What happens if the file can't be opened? Ie. File not found.
- What happens if the length of the file can't be determined? Ie.File description is lost.
- What happens if enough memory can't be allocated? Ie.File is too big
- What happens if the read fails? Ie.File corrupted
- What happens if the file can't be closed? Ie. IO

Traditional Error Handling

```
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;
}
```

Exception Handling

```
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;
    }
}
```

- Error Handling Code & "Regular" Code are separated

Exception Handling

- Avoid code:

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

Example

- Consider the following method call stack:

```
method1 {
     call method2;
}
method2 {
     call method3;
}
method3 {
     call readFile;
}
```

Traditional Error Handling

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;//Error Handling Code
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;//Error Propagating Code
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;//Error Propagating Code
    else
        proceed;
}
```

Exception Handling

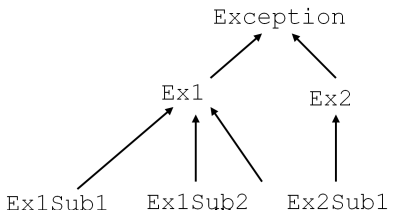
```
method1 {
    try {
        call method2;
    } catch (AnException ae) {
        doErrorProcessing;
    }
}
method2 throws AnException{//Propagating declaration
    call method3;
}
method3 throws AnException{//Propagating declaration
    call readFile;//the original readFile
}
```

- Declaration can propagate exception handling.

Exception Handling

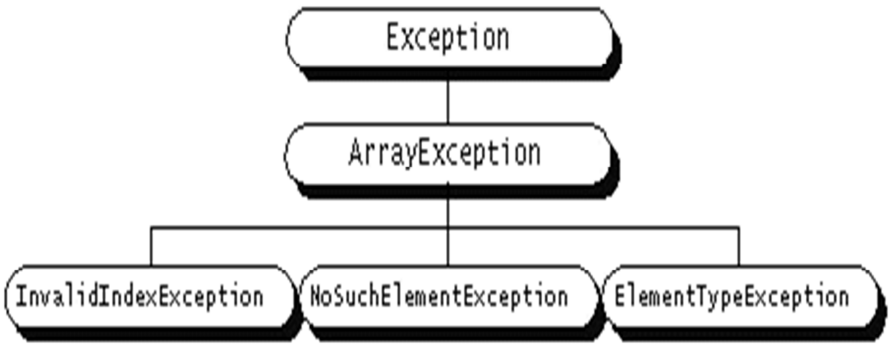
- Remarks:
  - Never declare a method to throw directly `Exception` or `Throwable`.
  - Declare to throw the most-derived exceptions.
  - Example:

```
f() throws Ex1Sub1, Ex1Sub2, Ex2Sub1
```



Grouping Exceptions

Use Class hierarchy



Create your own Exceptions

- Checked Exception: generally used for business errors
  - Your new Exception Class extends `java.lang.Exception` or its subclasses except `java.lang.RuntimeException`
- Unchecked Exception: generally used for technical errors
  - Your new Exception Class extends `java.lang.RuntimeException` or its subclasses.
  - A new runtime exception requirement is less used generally.

Create your own Exception

- Your exception should shows the necessary context information
- The exception name must be clear about the reason (nature) of the exception

```
public class InsufficientCreditException extends Exception{
  private int m_orderNum;
  private int accountNum;
  private String m_ownerName;
  public InsufficientCreditException(String msg, int orderNum,
                                     int accountNum, String ownerName){
    super(msg);//Message msg should be used for development only
    m_orderNum = orderNum;
    m_accountNum = accountNum;
    m_ownerName = ownerName;
  }
}
```

Create your own Exception

- The following exception creation is not good:

```
class MyException extends Exception{
   String m_reason;
   public MyException(){}
   public MyException(String reason){
       super(reason);
       m_reason = reason;
   }
}
throw new MyException("Number format is wrong");
throw new MyException("1234:Connection lost");
throw new MyException("1234");
```

- Good creation:

```
class <Reason>Exception extends Exception{//…}
```

Throw Exception

- Throw:

```
if (accout.getBalance < withdrawnAmmount)
  throw new InsufficientCreditException(
     orgEx.getMessage(),
     account.getOrderNum()
     account.getAccountNum(),
     account.getOwnerName());
```

- Re-throw:

```
catch (InsufficientCreditException ice) {
  //do some handling such as logging
  throw ice;
}
```

Translate Exception

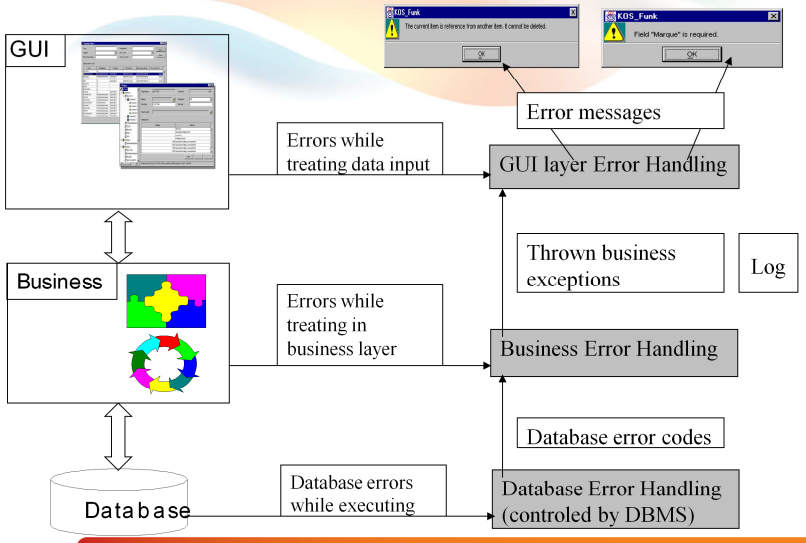
```
class MyException extends Exception{
   public MyException(String msg, Throwable t){
        super(msg, t);
   }
}

try{
   //do something that can raise YourException
}
catch (YourException e) {
   throw new MyException("my own message", e);
}
```

Exception Layers

- Like application layers, in a good design application, for each layer there should be some related exceptions.
  - Presentation Exceptions
  - Business Exceptions
  - Data Exceptions
  - Technical Exceptions

  } Your own Exceptions
- Normally, all exceptions should be thrown up to presentation layer. They are handled here to report to application users.
- Any uncaught exception, to the Application Layer, must be caught and handled here.

Handling error (Recall)



Exception Precautions[2]

- Once overriding method you can throw only exceptions thrown by the method in the base-class, or subclasses of the exception.
- Stop instruction flow right away. This leads to non-released resources such as leak of network connections, and leak of opened file. Example:

```
try{
   //open resources except memory
   //do something that raises MyException
   //release resources except memory
}
catch(MyException me) {
   System.out.prinln("My Exception occurs");
   //there is no more code to release resources
}
```

`Finally` Clause

- Block of code executes whether or not an exception occurs in a try block. For instance the code to solve the second precaution of the exception in the previous slide.
- Java provides a keyword `finally` along with `try` and `catch`.

```
try{//…}
catch(){//…}
//… other catches
finally{
  //Stuffs that happens all the time
}
```

Finally Clause

```
MyConnection con = null;
try{
  //open resources
  con = new MyConnection();
  //do something that raises MyException
}
catch(MyException me) {
  System.out.prinln("My Exception occurs");
}
finally{
  //code which is executed all the time here
  if(con!=null)
    con.close(); //release opened resources
}
```

`Finally` Clause

```
readFile {//Correct exception handling
    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;
    } finally {
       close the file;
    }
}
```

`Finally` Precautions

- Exception loss

```
MyConnection con = null;
try{
   //open resources
}
finally{
   if(con!=null)
      con.close(); //close open resources
}
```

  - **if `con.close()` raises an exception, X, then MyException is lost and X is thrown instead.**

- Must catch exceptions in finally clauses but do not throw, re-throw or translate it.
- Should not return in `finally` clause

Exception Guidelines[3]

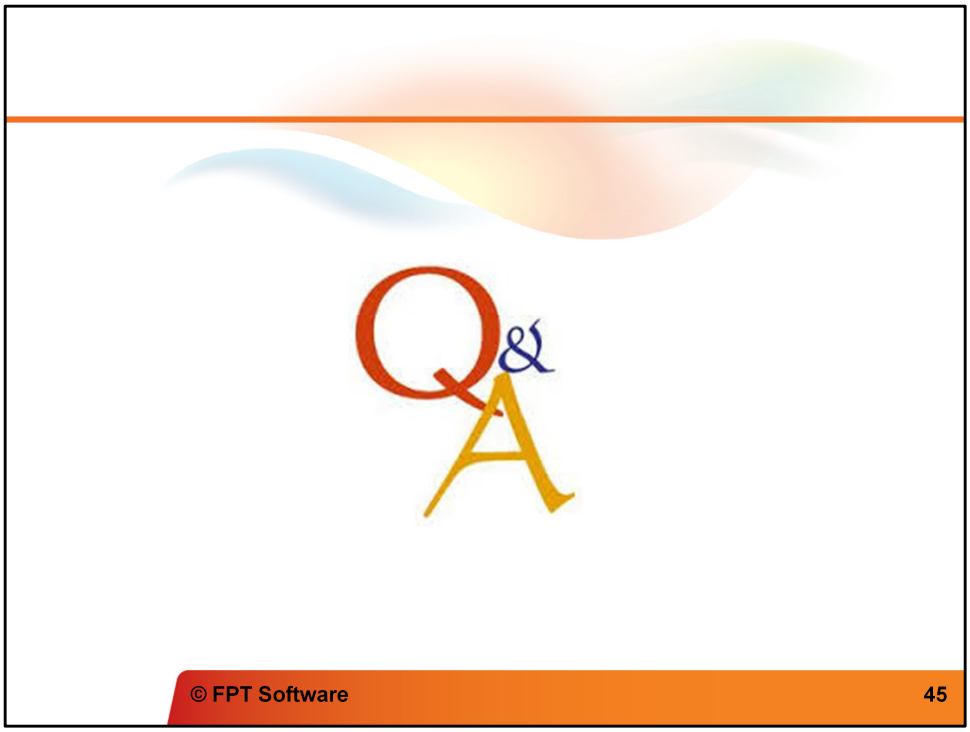
- Use Exception to
  - signal abnormal situations (exceptional situations) to the caller to react to these situations.
  - indicate that its caller has violated the method calling contract. For instance the caller gives a wrong argument list to the method.

Checked vs. Unchecked[3]

Checked exceptions	Unchecked exceptions
For exceptional alternatives to the normal ending of a method.	To indicate that a method is used improperly, for example when the input parameters are out of bound.
For reproducible and predictable situations.	For rare, hardly predictable situations whose handling would clutter the code.
When the user of a method should handle the exception directly.	To hide details particular to one implementation of an interface or to the environment.

Guidelines[3]

- Wrap Exceptions from a thrown exception (Translate Exception). This could be recursive.
- Wrapping and wrapped exceptions can be both checked or unchecked.
- Add attributes to exception classes to give more information about the exception. Ie. Name of the file that could not be opened.
- Declare even Unchecked exceptions in a throws clause
- Mention unchecked exceptions in javadoc
- Design exception hierarchy for complex error cases


Q&A
© FPT Software
45