



Java Exception Handling

Banu Prakash

Objectives

- Understand the Exception handling mechanism.
- Understand the difference between an error and exception.
- Why Exception handling is important?
- Understand different types of exception.
- Understand how to funnel exceptions.
- How to write custom exception classes and the need for it?

What is an Exception?

- Exception is an abnormal condition that arises when executing a program
- In Java an exception is an object that describes an exceptional condition (error) that has occurred when executing a program.
- Effective exception handling will make your programs more robust and easier to debug. They help answer these three questions:
 - What went wrong?
 - Answered by the type of exception thrown.
 - Where did it go wrong?
 - Answered by exception stack trace.
 - Why did it go wrong?
 - Answered by exception message

- Exception handling involves the following:
 - When an error/exception occurs, an object (exception) representing this error is created and thrown in the method that caused it.
 - That method may choose to handle the exception itself or pass it on to the invoking method.
 - Either way, at some point, the exception is caught and processed
- Exceptions sources can be:
 - Generated by Java run-time system
 - Manually generated by programmer's code.

Example

- Video : Exception_Handling_1.swf
 - Illustrates abnormal program termination in scenarios where there is no exception handling.

Exception Handling traditional approach

- Method returns error code.
 - Problem: Forget to check for error code
 - Failure notification may go undetected
- Problem: Calling method may not be able to do anything about failure
 - Program must fail too and let its caller worry about it
 - Many method calls would need to be checked
- Instead of programming for success
 `object.doSomething()`
you would always be programming for failure:
 `if (!object.doSomething()) return false;`

Exception handling constructs

Five constructs are used in exception handling:

- **try** - a block surrounding program statements to monitor for exceptions
- **catch** - together with try, catches specific kinds of exceptions and handles them in some way
- **finally** - specifies any code that absolutely must be executed whether or not an exception occurs
- **throw** - used to throw a specific exception from the program
- **throws** - specifies which exceptions a given method can throw (duck)

The try, catch and finally block

```
try {  
    /*  
    * some codes to test here  
    */  
} catch (SQLException sx) {  
    /*  
    * handle Exception1 here  
    */  
} catch (IOException ix) {  
    /*  
    * handle Exception2 here  
    */  
} catch (Exception ex) {  
    /*  
    * handle Exception3 here  
    */  
} finally {  
    /*  
    * always execute codes here  
    */  
}
```

try block encloses the context where a possible exception can be thrown

each `catch()` block is an exception handler and can appear several times

An optional `finally` block is always executed before exiting the `try` statement.

Example:

- Video
 - Refer: Exception_Handling_2.swf
 - Illustrates basic constructs of exception handling.
- Code Example:
 - Refer: Exception_1.zip
 - Example used for video presentation

Why use exception handling mechanism?

Traditional Exception Handling

Code

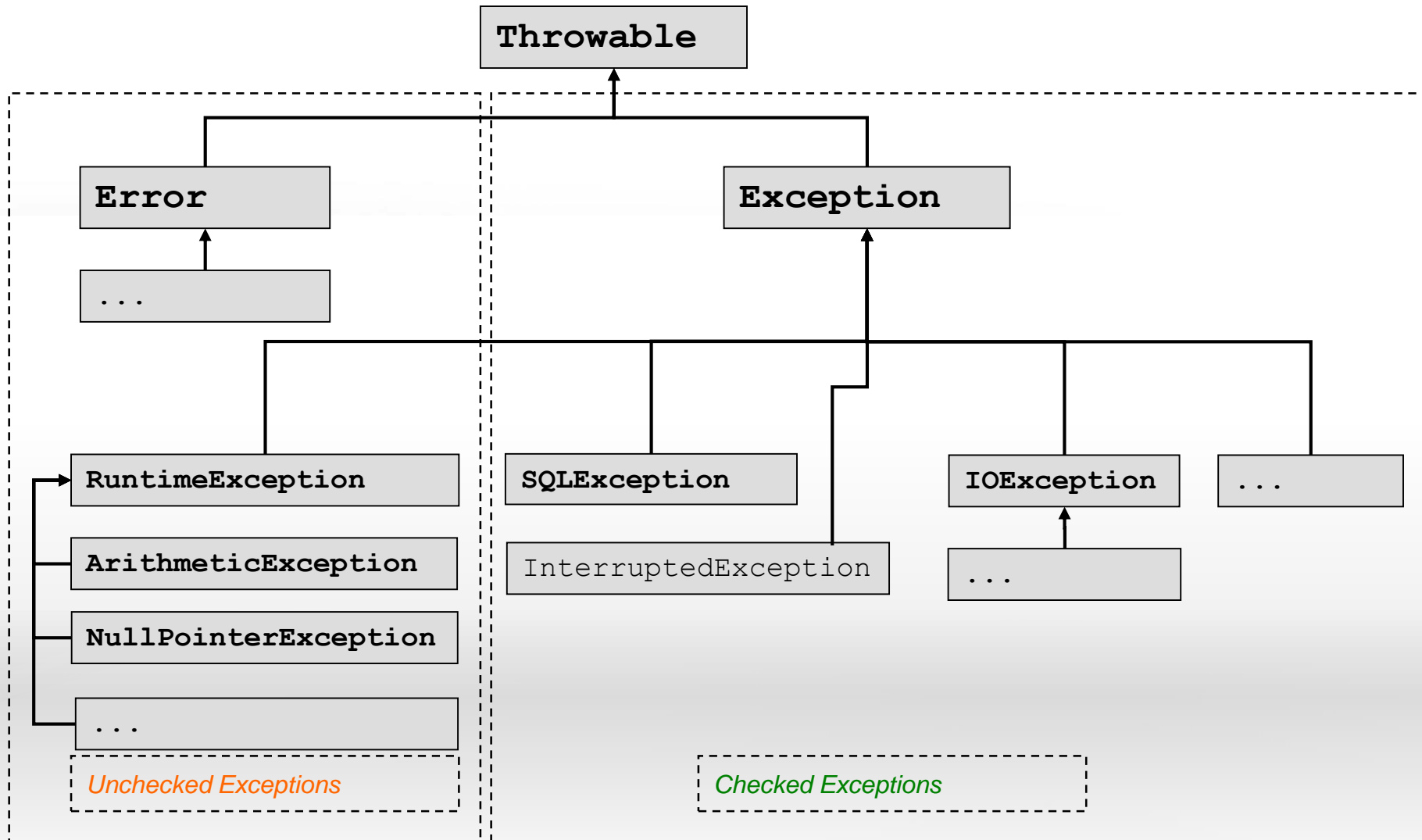
```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
      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

Java 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;
    }
}
```

- Separation of exception code from actual code

Exception class hierarchy



- Errors represent critical errors that should not occur and that the application is not expected to recover from.
- Errors are typically generated from mistakes in program logic or design and should be rectified through correction of design or code
- Examples: `OutOfMemoryError`, `StackOverflowError`, `UnsatisfiedLinkerError`,...

- Code Example:
 - Refer: Exception_2.zip
 - Contains ErrorsExample.java
 - Illustrates how errors can be caught but cannot be handled.

Un Checked Exceptions

- All exceptions inherited from RuntimeException.
- Runtime exceptions usually occur due to a bug in a code.
- Because they are unpredictable and could be anywhere in the application code, the compiler does not enforce their handling.
- Examples: ArithmeticException, NullPointerException,...
- “Runtime Exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or handle them in anyway”

Checked Exceptions

- Checked exceptions are used for problems predictable at design time.
- The compiler enforces handling of these exceptions.
- Software components either have to catch those exceptions, or declare that they are not catching it.
- If the method is declared saying that it is not catching then the calling code is aware that there is something to be done about potential problem.
- Examples : SQLException, IOException, ClassNotFoundException

Note: Checked exceptions are part of the method signature and are therefore integral part of the contract the class or interface has with the clients.

Important methods of Throwable class

- **String getMessage()**
String [getLocalizedMessage\(\)](#)

- Gets the detail message, or a message adjusted for this particular locale.

- **void printStackTrace()**
void printStackTrace(PrintStream)
void [printStackTrace\(PrintWriter\)](#)

- Prints the Throwable and the Throwable's call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown

The throws keyword

- If an Checked exception occurs we have two choices:
 - Handle the exception using try and catch blocks
 - Specify to the compiler that the method is ducking the exception and the exception should be caught by the calling code
 - Use throws specifier to duck the checked type of exception

```
public void uploadCsvFile(String fileName)  
    throws SQLException, FileNotFoundException{
```

Example: Checked and Unchecked exceptions

- Code Example:

- Refer:Exception_3.zip

- contains CheckedUncheckedExample.java
 - Illustrates difference between checked and unchecked type of exceptions.

- Video:

- Refer: Exception_Handling_3.swf

Exception Wrapping

- Exception wrapping is like repositioning the problem in a context that is easily interpretable by higher level application layers.
- Instead of dealing with the problem, the component creates a new Exception that contains the original one, plus some extra info, and the re-throws it.

Exception Wrapping explained

- Imagine you are writing an application that you would like to sell to more than one customer.
 - The business logic is the same, but some of your customers want to persist data in a relational database, some in LDAP, ..
 - We do not want to modify the business layer because of the change in data access layer

Solution :

- Your DAO (data access layer) can catch the SQLException, wrap them in a PersistenceException and re-throw them.
- Business layer becomes unaware of underlying persistence mechanics.

```
public void addEmployee(Employee employee) throws  
    PersistenceException;
```

This method does not indicate how persistence is achieved namely, rdbms or files.

Exceptions and abstraction

- Without Wrapping if register method ducks original exception
- This exception can be understood only by a person who knows database

ORA-00001: unique constraint violated ...

Get started with Gmail

First name:

Last name:

Desired Login Name: @gmail.com
Examples: JSmith, John.Smith

Exceptions and abstraction

- After Wrapping in register method and re-throwing
- This message can be understood by everyone.

banuprakash@gmail.com is not available.

Get started with Gmail

First name:	<input type="text" value="Banu"/>
Last name:	<input type="text" value="Prakash"/>
Desired Login Name:	<input type="text" value="banuprakash"/> @gmail.com

Examples: JSmith, John.Smith

Example: Exception Wrapping

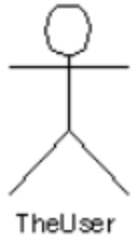
- Video:

- Exception_Handling_4.swf
- Exception_Handling_4a.swf
- Exception_Handling_4b.swf
 - Illustrates the need for custom user defined exception.

- Code Example:

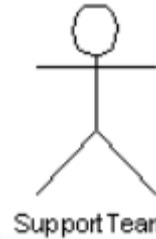
- Refer:Exception_4.zip
 - Code examples used for video presentation

Exception handling the big picture



TheUser

The user does not need to know the details, and probably would not understand anyway. An apologetic message is displayed asking him to retry



SupportTeam

Support time is alerted about the problem. They know that the persistence mechanism used is JDBC, they look into log file and try to fix it.

Service Facade Layer

This layer catches all `ApplicationException`, logs it, rolls back the transaction and alerts the right person. This layer is also not aware of persistence mechanism used.

Business Logic Layer

This layer ignores `PersistenceException` because it is concerned about only business logic. This layer is not aware of persistence mechanism used.

Data Access Layer

An `SQLException` occurs here due to some database related issues, the layer catches `SQLException`, wraps it with `PersistenceException` and re-throws it

Things to Remember !!!

1. Do not suppress the exception

```
try{  
  
    // File reading code here  
  
}catch(IOException ioEx){  
    // do Nothing  
}
```



- Often we use try catch block in our code when compiler/IDE force as to handle the checked exceptions, we do write try catch block and just leave the catch block as Empty.
- We must handle these checked exceptions, that's the reason it has been forced by compilers, by leaving empty catch block, we defeat the purpose of these checked exceptions

Things to Remember !!!

2. Do not throw the irrelevant / unknown exceptions to the caller

For example, a method is trying to connect to Database and it throws Connection Could not established exception, then code should not throw this to handler, the method must handle and do some recoveries like re try or try to connect to the alternate Database.

The caller may not know what to do with this exception, so this must be handled then and there



3. Do not handle the irrelevant exception by own.

The code must not handle the exceptions which are not relevant to that piece, it should throw the exception to the caller



4. Do Wrap-up the exception

- Log the exceptions
- Wrap the exception and pass it to the above layer

5. Naming Exceptions

- The name of the exception should try to describe what went wrong and should not imply any notion of gravity.
- Having an exception called FatalException does not make sense: only the handler of the exception knows how fatal the exception is, not the thrower.



Things to remember !!!

6. Never return from finally block


- Even if exception occurs the finally block is executed and will return normally, as if nothing happened. The exception is lost.

7. Exceptions carry information, they do not do things.

- Bad Code:

```
public class MyException extends Exception {  
    public MyException(Exception rootCause) {  
        super(rootCause);  
        logger.error(rootException);  
    }  
}
```





Quiz!

Some quiz questions to reinforce the classroom learning

1. Which of the listed exceptions are of checked type?
 - ArithmeticException
 - ClassCastException
 - FileNotFoundException
 - NullPointerException
2. The throws keyword compulsorily required to be if used if the method does not handle.
 - a) Checked type of exceptions
 - b) Unchecked type of exceptions
3. Does the finally block get executed if an exception occurs.

4. Can a catch block exist without an try block.
5. Is there anything wrong with the following exception handler as written? Will this code compile?

```
try {  
    } catch (Exception e) { }  
    catch (ArithmeticException a) { }
```

6. Is the following code legal?

```
try { }  
finally { }
```




Explore More!!

Never let your curiosity die!

- Assertions
 - Using assert statements
 - Enabling assertions
 - Disabling assertions
 - <http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html>
- Exception handling and performance
- The fillStackTrace() of Throwable class
- Exception Handling Templates in Java
 - <http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html>
- Logging Exceptions: Where to Log Exceptions?
 - <http://www.java-tips.org/java-se-tips/java.util.logging/how-to-log-an-exception.html>



References

Contains the reference that will supplement the self learning and will be needed for completing the assignments & practice questions

- Recommended Book
 - Thinking in Java
 - Chapter 10 :Error Handling with Exceptions
- Java Exception Handling Tutorial <http://goo.gl/HMVxy>
 - <http://goo.gl/HMVxy>
- Java Tutorial from oracle.com
 - <http://goo.gl/sezZU>