

Core Java

Lesson 09 : Exception Handling



Lesson Objectives

- On completion of this lesson, you will be able to:
 - Explain the concept of Exception
 - Describe types of Exceptions
 - Handle Exception in Java
 - Create your own Exceptions
 - Exception chaining
 - New Features of Java 7
 - Try-with-resources and Autocloseable
 - Handling Suppressed Exceptions
 - Catch Block Handling Multiple Exceptions
 - State best practices on Exception





Why is exception handling used?

- No matter how well-designed a program is, there is always a chance that some kind of error will arise during its execution, for example:
 - Attempting to divide by 0
 - Attempting to read from a file which does not exist
 - Referring to non-existing item in array
- An exception is an event that occurs during the execution of a program that disrupt its normal course.





Exception Handling

- Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions:
 - Eg: Hard disk crash, Out of bounds array access, Divide by zero etc
- When an exception occurs, the executing method creates an Exception object and hands it to the runtime system —“throwing an exception”
- The runtime system searches the runtime call stack for a method with an appropriate handler, to handle/catch the exception.





Exception Handling

- The general form of exception handling block:

```
try {  
    //code to be monitored.  
}  
catch (Exception1 e1 ) {  
    //exception handler for Type Exception1  
}  
catch (Exception2 e2 ) {  
    //exception handler for Type Exception2  
}  
finally {  
    // code that must be executed.  
}
```



Demo

- Execute the DefaultDemo.java program

```
class DefaultDemo {  
    public static void main(String a[]) {  
        String str = null;  
        str.equals("Hello");  
    }  
}
```

Output:
Exception in thread "main"
java.lang.NullPointerException at
 com.igatepatni.lesson5.DefaultDemo.main(DefaultDemo.
 java:6)





Advantages of Exceptions


➤ Separating Error-Handling Code from "Regular" Code:

Code without Exception handling

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

Code with Exception handling

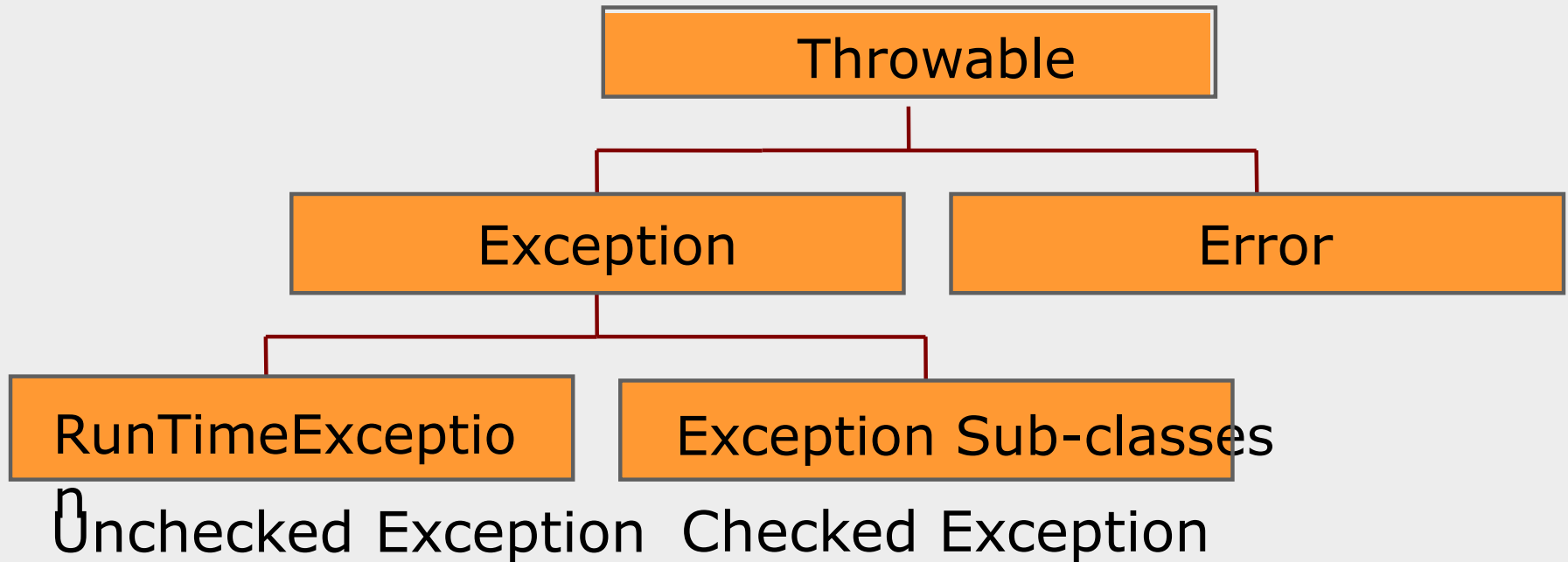
```
readfile() {  
    try {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    } catch (fileOpenFailed) { doSomething; }  
    catch (sizeDeterminationFailed) { doSomething; }  
    catch (memoryAllocationFailed)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



Note that error handling code and regular code are separate



Hierarchy of Exception Classes





Error

- An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.
- Most such errors are abnormal conditions.
- Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
 - Stack overflow is an example of such an error.



Exception

- The Exception class and its subclasses are a form of Throwables. They indicate conditions, which a reasonable application may want to catch.
- Two Types:
 - Checked Exception
 - UnChecked Exception



Checked/Compile Time Exceptions

➤ Characteristics of Checked Exceptions:

- They are checked by the compiler at the time of compilation.
- They are inherited from the core Java class Exception.
- They represent exceptions that are frequently considered “non-fatal” to program execution.
- They must be handled in your code, or passed to parent classes for handling.
- Some examples of Checked exceptions include: IOException, SQLException, ClassNotFoundException



Unchecked/Runtime Exceptions

- Unchecked exceptions represent error conditions that are considered “fatal” to program execution.
 - Runtime exceptions are exceptions which are not detected at the time of Compilation.
 - They are encountered only when the program is in execution.
 - It is called unchecked exception because the compiler does not check to see if a method handles or throws these exceptions.



9.3: Handling Exceptions

Keywords for handling Exceptions

- **try** : This marks the start of a block associated with a set of exception handlers.
- **catch** : The control moves here if an exceptions is generated.
- **finally** :This is called irrespective of whether an exception has occurred or not.
- **throws** : This describes the exceptions which can be raised by a method.
- **throw** : This raises an exception to the first available handler in the call stack, unwinding the stack along the way.



9.3: Handling Exceptions

Why to handle exceptions?

➤ Without Exception handling

```
class WithoutException {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
        System.out.println("Will not be printed"); } }
```

➤ With Exception handling

```
class WithExceptionHandling {  
    public static void main(String args[]) {  
        int d=0, a;  
        try {  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) {  
            System.out.println("Division by zero.");  
        }  
        System.out.println("This will get printed"); } }
```



9.3: Handling Exceptions

Try and Catch

- The try structure has three parts:
 - The try block : Code in which exceptions are thrown
 - One or more catch blocks : Respond to different Exceptions
 - An optional finally block : Contains code that will be executed regardless of exception occurring or not
- The catch Block:
 - If exception occurs in try block, program flow jumps to the catch blocks.
 - Any catch block matching the caught exception is executed.



9.3: Handling Exceptions

Using Try and Catch

- Execute the TryCatchDemo.java program



Multiple Catch Blocks

- If you include multiple catch blocks, the order is important.

```
public void divide(int x,int y)
{
    int ans=0;
    try{
        ans=x/y;
    }catch(Exception e) {
        //handle }
    catch(ArithmeticException f) {
        //handle}
```



You must catch subclasses before their ancestors





Nested Try Catch Block

```
try {
    int a = arg.length;
    int b = 10 / a;
    System.out.println("a = " + a);
    try {
        if(a==1)
            a = a/(a-a);
        if(a==2) {
            int c[] = { 1 };
            c[42] = 99;
        }
    } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e); }
} catch(ArithmeticException e) {
    System.out.println("Divide by 0: " + e); }
```



The Finally Clause

- The finally block is optional.
- It is executed whether or not exception occurs.

```
public void divide(int x,int y)
{
    int ans;
    try{
        ans=x/y;
    }
    catch(Exception e) {
        ans=0; }
    finally{
        return ans; // This is always executed }
}
```



Demo: The Finally Clause

- Execute the FinallyDemo.java program

Output of this program:

```
inside procA  
procA's finally  
Exception caught  
inside procB  
procB's finally  
inside procC  
procC's finally
```



Throwing an Exception

- You can throw your own runtime errors:
 - To enforce restrictions on use of a method
 - To "disable" an inherited method
 - To indicate a specific runtime problem
- To throw an error, use the throw Statement
 - throw ThrowableInstance
 where ThrowableInstance is any Throwable Object



Throwing an Exception

```
class ThrowDemo {  
    void proc() {  
        try {  
            throw new FileNotFoundException ("From Exception");  
        } catch (FileNotFoundException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e; // rethrow the exception  
        }  
    }  
    public static void main(String args[]) {  
        ThrowDemo t=new ThrowDemo();  
        try {  
            t.proc();  
        } catch (FileNotFoundException e) {  
            System.out.println("Recought: " + e);  
        }  
    }  
}
```



Using The Throws Clause

- If a method might throw an exception, you may declare the method as “throws” that exception and avoid handling the exception yourself.

```
public class ThrowsDemo {  
    public static void main(String[] args) {  
        try {  
            fileOpen();  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
            System.out.println("File name specified does not exist "  
                               + e.getMessage());  
        }  
  
        static void fileOpen() throws FileNotFoundException {  
            FileReader fileReader = new FileReader("test.txt");  
        } } }
```



User specific Exception

➤ To create exceptions:

- Write a class that extends(indirectly) Throwable.
- What Superclass to extend?
 - For unchecked exceptions: RuntimeException
 - For checked exceptions: Any other Exception subclass or the Exception itself

```
class AgeException extends Exception {  
    private int age;  
    AgeException(int a) {  
        age = a;  
    }  
    public String toString() {  
        return age+" is an invalid age";  
    }  
}
```




User specific Exception

➤ To create exceptions:

- Write a class that extends(indirectly) Throwable.
- Which Superclass to extend?
 - For unchecked exceptions: RuntimeException
 - For checked exceptions: Any other Exception subclass or the exception itself

```
class AgeException extends Exception {  
    private int age;  
    AgeException(int a) {  
        age = a;  
    }  
    public String toString() {  
        return age+" is an invalid age"; } } }
```



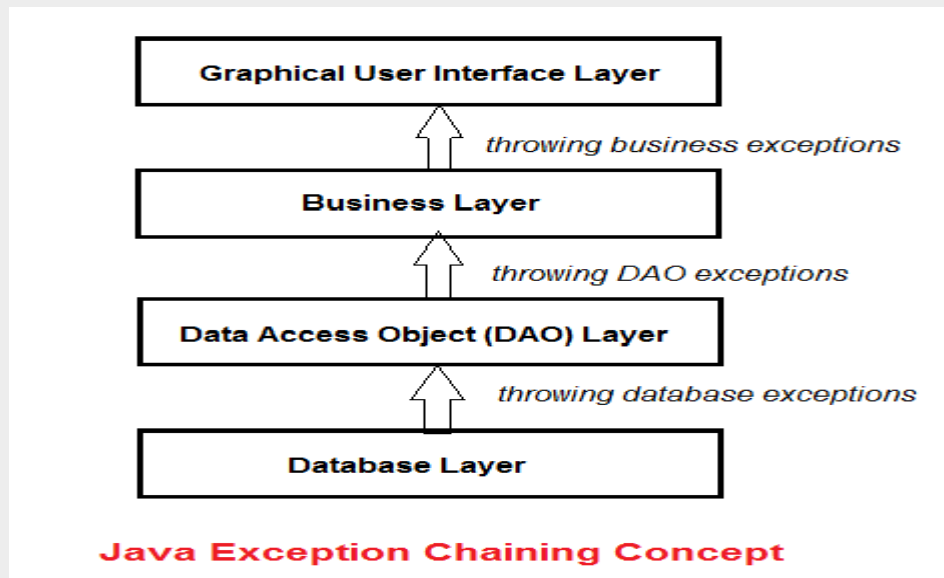
Demo: User Specific Exception

- Execute UserException.java program



Exception chaining

- Exception chaining is the process of re-throwing multiple exceptions across different abstraction layers of a program.
- The key principle here is that, these exceptions are chained together to maintain the stack trace from the exception at the lowest layer to the one at the highest layer.
- The following picture illustrates this concept visually:





Try-with-resources

- Resources used by java programs like file or database connection needs to be closed properly
- To close resource automatically in exception handling, Java 7 has added try-with-resources:

```
try (resource1; resource 2; ..... resource n) {  
    //resource related work  
}  
catch (Exception e) {  
    //handle exception  
}
```

- It can be used to close resources that implement `java.lang.AutoCloseable`



Handling Suppressed Exceptions

- At any point of time only one exception can be thrown by a method per execution. In cases when more than one exception would be thrown from the method, one of them will be suppressed and not thrown by the compiler.
- This exception is known as the suppressed exceptions in Java.
- In Java 7, the most common scenario for getting the suppressed exception is when using the try-with-resource statement
- When using the try with resources statement, first exception will occur in the try block and then probably encounters another exception while closing the resource.
- The second exception will be implicitly thrown hence that will be suppressed.
- Since multiple exceptions are thrown while closing the resources, additional exceptions are attached to the primary exception as the suppressed exceptions.



Handling Suppressed Exceptions

- Java 7 have added a new constructor and two new methods in the Throwable class as below:

```
Throwable.getSupressed(); // Returns  
Throwable[]  
Throwable.addSupressed(aThrowable);
```



Multi-Catch Blocks

- Starting from Java 7, a single catch block can be used to catch multiple exceptions
- Multi-catch block separates handled exceptions by vertical bar (|)

```
try {  
    }  
catch (exception 1 | exception 2 | .....| exception n) {  
    }
```

- Alternatives in multi-catch block is not allowed



Multiple Catch Blocks

- Execute the MultiCatch.java class



9.3: Exception Handling

Lab : Exception

➤ Lab 7: Exception Handling





The Best Practices

- Avoid empty catch blocks.
- Avoid throwing and catching a generic exception class.
- Pass all the pertinent data to exceptions.
- Use the finally block to release the resources



Best Practices

- Avoid throwing unnecessary exceptions.
- Finalize method is not reliable.
- Exception thrown by finalizers are ignored.
- While using method calls, always handle the exceptions in the method where they occur.
- Do not use loops for exception handling.



Best Practices

- When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"



Summary

- Exceptions are powerful error handling mechanisms.
- A program can catch exceptions by using a combination of the try, catch, and finally blocks:
 - The try block identifies a block of code in which an exception can occur.
 - The catch block identifies a block of code, known as an exception handler.
 - The finally block identifies a block of code that is guaranteed to execute.
 - Try-with-resources
 - Multi-catch blocks
- Throw is used to throw an exception by the user.





Review – Match the Following format

1. CheckedException	A. Compulsory to use if a method throws a checked exception and doesn't handle it
2. finally	B. Inherited from RuntimeException
3. throws	C. Can have any number of catch blocks
4. Unchecked Exception	D. Used to avoid "resource leak"
5. try	E. Inherited from Exception

