Exception Handling

Prepared by
Rubyeat Islam
Lec, CSE Dept, MIST
rubyeat88@gmail.com

What is Exception?

- An exception is an event, occurs during the execution of a program but disrupts the normal flow of program's instruction.
- **Code:**

```
public class demo {
    public static void main(String[] args) {
        int a= 100/0;     //this is exception /zero

      System.out.println("Hello CSE-19");
    }
}
```

- **Output:**

```
C:\Users\MohammadYusuf\.jdks\openjdk-14.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBra
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at try_catch.demo.main(demo.java:5)

Process finished with exit code 1
```

Try_Catch Block

- Exception Handling

In 1 try_catch block, there will be only one try and can be single or multiple catch block;

```
Try{

} catch (Name of exception) {



} catch (Name of exception) {

}
```

Try_Catch

Code:

```
public class demo {
    public static void main(String[] args) {
        try {
            int a= 100/0;
        } catch (ArithmeticException e ){


        }
        System.out.println("Hello CSE-19");
    }
}
```

Output:

```
C:\Users\MohammadYusuf\.jdks\openjdk-14.0.1\bi
Hello CSE-19
```

Try_Catch with Exception Name

Code:

```
public class demo {
   public static void main(String[] args) {
      try {
         int a= 100/0;
      } catch (ArithmeticException e ){
    System.out.println(e);      //Print the exception
      }
      System.out.println("Hello CSE-19");
   }
}
```

Name of Exception

Output:

```
C:\Users\MohammadYusuf\.jdks\openjdk-14.0.1\bin\java.exe "-javaag
java.lang.ArithmeticException: / by zero
Hello CSE-19

Process finished with exit code 0
```

Try_Catch with General Exception

```
public class demo {
    public static void main(String[] args) {
        try {
            int a= 100/0;
        } catch (ArithmeticException e ){
            System.out.println(e);

        } catch (Exception e){
            System.out.println(e);
        }

        System.out.println("Hello CSE-19");
    }
}
```

When you don't know the actual name of exception Generalize it

Which exception Called?

```
C:\Users\MohammadYusuf\.jdks\openjdk-14.0.1\bin\java.exe "-javaag
java.lang.ArithmeticException: / by zero
Hello CSE-19

Process finished with exit code 0
```

Always Remember

General exception has greater priority than any other exception in try_catch block.

Another Exception

```
public class demo {
    public static void main(String[] args) {
        int[] b= new int[2];
        try {
            System.out.println("Value of b = " + b[3]);
        } catch (ArithmeticException e ){
            System.out.println(e);
        } catch (Exception e){
            System.out.println(e);
        }
        System.out.println("Hello CSE-19");
    }
}
```

Size of array
Is 2. Exception
is no value
b[3] is there

Which exception
Will be called?

```
C:\Users\MohammadYusuf\.jdks\openjdk-14.0.1\bin\java.exe "-javaagent:C:\Program Fi
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 2
Hello CSE-19
```

Will this code compile?

```
public class demo {
    public static void main(String[] args) {
        int[] b= new int[2];
        try {
            System.out.println("Value of b = " + b[3]);
        } catch (ArithmeticException e ){
            System.out.println(e);
        } catch (Exception e){
            System.out.println(e);
        } catch (ArrayIndexOutOfBoundsException e){
            System.out.println(e);
        }

        System.out.println("Hello CSE-19");
    }
}
```

Error but why?

Try_Catch_Finally

- The finally block always executes when try block exists.
- This ensures that the finally block always executes whether exception occurs or not. That doesn't matter.

```
public class demo {
    public static void main(String[] args) {
        try {
            int a = 100 / 0;
         } catch (ArithmeticException e) {
            System.out.println("Catch Called");
            System.out.println(e);
        } finally {
            System.out.println("finally called");
        }
    }
}
```

```
Catch Called
java.lang.ArithmeticException: / by zero
finally called

Process finished with exit code 0
```

Finally Block

```
public class demo {
   public static int retint(){
      int a=100;
      try {
      System.out.println(“Try Called");
     return a;
      } catch (ArithmeticException e) {
         System.out.println("Catch Called");
         System.out.println(e);
         return a;
      } finally {
     System.out.println("finally called");
     }
   }

   public static void main(String[] args) {
      System.out.println(retint());
   }
}
```

First, try executes
But don’t return anything

Second,
finally executes

```
try Called
finally called
100

Process finished with exit code 0
```

Last, 100 returns from try.

Finally Block cont…

Now if we add exception in try block, what happens? Let's see

```
public class demo {
   public static int retint(){
      int a=100;
      try {
      a= a/100;
       System.out.println("Try Called");
      return a;
      } catch (ArithmeticException e) {
         System.out.println("Catch Called");
         System.out.println(e);
         return a;
      } finally {
      System.out.println("finally called");
    }
   }

   public static void main(String[] args) {
      System.out.println(retint());
   }
}
```

Which block return 100?
Try or catch??

```
Catch Called
java.lang.ArithmeticException: / by zero
finally called
100
```

Finally Block cont…

Finally block also overrides. Let's see

```
public class demo {
   public static int retint(){
      int a=100;
      try {
       a= a/100;
        System.out.println("Try Called");
       return a;
      } catch (ArithmeticException e) {
         System.out.println("Catch Called");
         System.out.println(e);
         return a;
      } finally {
       a= 5000;
       System.out.println("finally called");
       return a;
     }
   }

   public static void main(String[] args) {
      System.out.println(retint());
   }
}
```

Overrides 100?
5000 returns from finally.

```
Catch Called
java.lang.ArithmeticException: / by zero
finally called
5000
```

Exception in Inheritance

Rules for exception handling with method overriding in Java

- If **super class** method has not declared any exception using **throws clause** then subclass overridden method **can't declare any checked exception** but it **can declare unchecked exception** with the throws clause.
- If **super class** method has declared a checked exception using throws clause then subclass overridden method can do **one of the three things.**
- subclass can declare the same exception as declared in the superclass method
- Subclass can declare the subtype exception the exception declared in the super class method. But subclass method can not declare any exception that is up in the hierarchy than the exception declared in the super class method.
- Subclass method can choose not to declare any exception at all

Let’s see this example

inheritException.java

```java
package try_catch;

public class inheritException {
    public static void main(String[] args) {
        A a= new A();
        B b = new B();
        b.abc();
    }

}
class A{
    public void abc(){
        System.out.println("Inside Class A");
    }
}

class B{
    public void abc() {
        System.out.println("Inside Class B");
    }
}
```

Output

```
Inside Class B
```

Rule no 1

inheritException.java

```java
package try_catch;


public class inheritException {
    public static void main(String[] args)  {
        A a= new A();
        B b = new B();
        b.abc();
    }

}
class A{
    public void abc(){
        System.out.println("Inside Class A");
    }
}

class B extends A{
    public void abc() throws NullPointerException{
        System.out.println("Inside Class B");
    }
}
```

Super class A has No exception
But Sub class B has An exception.
This is called unchecked exception

Output

```
Inside Class B
```

Rule no 2.1

inheritException.java

```java
package try_catch;
import java.io.IOException;

public class inheritException {
    public static void main(String[] args) throws IOException {
        A a= new A();
        B b = new B();
        b.abc();
    }

}
class A{
    public void abc() throws IOException {
        System.out.println("Inside Class A");
    }
}

class B extends A{
    public void abc() throws IOException {
        System.out.println("Inside Class B");
    }
}
```

IOException must be Thrown by a method

IOException is not run Time exception so it should Be thrown by super class abc() method.

Output

```
Inside Class B
```

Rule no 2.2

inheritException.java

```java
package try_catch;
import java.io.IOException;

public class inheritException {
    public static void main(String[] args) throws IOException {
        A a= new A();
        B b = new B();
        b.abc();
    }

}
class A{
    public void abc() throws Exception {
        System.out.println("Inside Class A");
    }
}

class B extends A{
    public void abc() throws IOException {
        System.out.println("Inside Class B");
    }
}
```

A throws general Exception

B throws IOException Which is a subtype Exception of Exception class

Rule 2.2 gives error

If subclass throws general exception where superclass throws subtype exception

inheritException.java

```java
package try_catch;
import java.io.IOException;

public class inheritException {
    public static void main(String[] args) throws IOException {
        A a= new A();
        B b = new B();
        b.abc();
    }

}
class A{
    public void abc() throws IOException {
        System.out.println("Inside Class A");
    }
}

class B extends A{
    public void abc() throws Exception {
        System.out.println("Inside Class B");
    }
}
```

Error!! Why?

Rule 2.3

inheritException.java

```java
package try_catch;
import java.io.IOException;

public class inheritException {
    public static void main(String[] args) throws IOException {
        A a= new A();
        B b = new B();
        b.abc();
    }

}
class A{
    public void abc() throws IOException {
        System.out.println("Inside Class A");
    }
}

class B extends A{
    public void abc() {
        System.out.println("Inside Class B");
    }
}
```

Sub class throws no exception Whether Super class throws Exception.

Create Own Exception Class

Why?

- How do we know which exception is relevant, which is not?
- By looking at the names of the exceptions to see if its meaning is appropriate or not. For example,
  the **IllegalArgumentException** is appropriate to throw when checking parameters of a method; the **IOException** is appropriate to throw when reading/writing files.
- From my experience, most of the cases we need custom exceptions for representing business exceptions which are, at a level higher than technical exceptions defined by JDK.

For example: **InvalidAge**Exception, **LowScore**Exception,

TooManyStudentsException, etc.

Create Own Exception Class

How?

- Create a new class whose name should end with Exception like **ClassName**Exception. This is a convention to differentiate an exception class from regular ones.
- Make the class **extends** one of the exceptions which are subtypes of the **java.lang.Exception** class. Generally, a custom exception class **always extends** directly from the Exception class.
- Create a **constructor** with a String parameter which is the detail message of the exception. In this constructor, simply **call the super constructor** and **pass** the message.

Let's See

- The following is a custom exception class which is created by following the above steps:

StudentNotFoundException.java | StudentManager.java | Student.java

```java
package Own_Exception_class;

public class StudentNotFoundException extends Exception {
    public StudentNotFoundException(String message){
        super(message);
    }
}
```

- And the following example shows the way a custom exception is used is nothing different than built-in exception:

StudentNotFoundException.java | StudentManager.java | Student.java

```java
package Own_Exception_class;

public class StudentManager {
    public Student find(String studentID) throws StudentNotFoundException {
        if (studentID.equals("123456")) {
            return new Student();
        } else {
            throw new StudentNotFoundException("Could not find student with ID " + studentID);
        }
    }
}
```

Own Exception class cont..

- And the following test program handles that exception:

```
StudentNotFoundException.java   StudentManager.java   Student.java
1   package Own_Exception_class;
2
3   public class Student {
4       public static void main(String[] args) {
5           StudentManager manager = new StudentManager();
6
7           try {
8
9               Student student = manager.find( studentID: "0000001");
10
11          } catch (StudentNotFoundException ex) {
12              System.out.println(ex);
13          }
14      }
```

- Run this program and you will see this output:

```
C:\Users\MohammadYusuf\.jdks\openjdk-14.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrai
Own_Exception_class.StudentNotFoundException: Could not find student with ID 0000001

Process finished with exit code 0
```

Re-throw Custom Exception

- It's a common practice for catching a built-in exception and re-throwing it via a custom exception. To do so, let add a new constructor to our custom exception class. This constructor **takes two parameters: the detail message and the cause of the exception**. This constructor is implemented in the Exception class as following:

  public Exception(String message, Throwable cause)

- Besides the detail message, this constructor takes a Throwable's subclass which is the origin (cause) of the current exception.

Re-throw Custom Exception cont..

- For example, create the **StudentStore**Exception class as following:

```java
package Own_Exception_class;

public class StudentStoreException extends Exception{
    public StudentStoreException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

- And the following example shows where the **StudentStoreException** gets thrown:

```java
public class StudentManager {
    public Student find(String studentID) throws StudentNotFoundException {
        if (studentID.equals("123456")) {
            return new Student();
        } else {
            throw new StudentNotFoundException("Could not find student with ID " + studentID);
        }
    }
    int [] b= {1,2,3};
    public void save(Student student) throws StudentStoreException {
        try {
            int a= b[3];
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new StudentStoreException("Failed to save student", ex);
        }
    }
}
```

Handle Re-throw Exception

- And the following code demonstrates handling the StudentStoreException above:

```
StudentNotFoundException.java   StudentManager.java   Student.java
            Student student = manager.find( studentID: "0000001");

        } catch (StudentNotFoundException ex) {
            System.out.println(ex);
        }

        StudentManager manager1 = new StudentManager();

        try {

            manager1.save(new Student());

        } catch (StudentStoreException ex) {
            System.out.println(ex);
        }
    }
}
```

Why Re-throw?

- Here,

suppose that the **save()** method stores the specified student information into a database using JDBC. The code can throw **ArrayIndexOutOfBoundsException**. We catch this exception and throw a new **StudentStoreException** which wraps the **ArrayIndexOutOfBoundsException** as its cause. And it's obvious that the **save()** method declares to throw **StudentStoreException** instead of A**rrayIndexOutOfBoundsException .**

So what is the benefit of re-throwing exception like this?

- Why not leave the original exception to be thrown?
- Well, the main benefit of re-throwing exception by this manner is adding a higher abstraction layer for the exception handling, which results in more meaningful and readable API. Do you see StudentStoreException is more meaningful than ArrayIndexOutOfBoundsException, don't you?
- However, remember to include the original exception in order to preserve the cause so we won't lose the trace when debugging the program when the exception occurred.

Keep Practicing