Exception Handling in JAVA

Dr. Susovan Jana
Associate Professor & Programme In-Charge
Department of Computer Science & Engineering (IoT)
Institute of Engineering & Management, Kolkata, INDIA
Email (O): susovan.jana@iem.edu.in
Email (P): jana.susovan2@gmail.com

What is exception?

- ❑ Exception is an event that disrupts the normal flow of the program.
- ❑ It is an object which is thrown at runtime.

Types of Exception

- Checked Exception
  - The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.
  - Checked exceptions are checked at compile-time.
- Unchecked Exception
  - The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
  - Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

Some Common Exceptions

- ArithmeticException occurs
  - If we divide any number by zero, there occurs an ArithmeticException.
  - int a=50/0;//ArithmeticException
- NullPointerException occurs
  - If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.
  - String s=null;
    System.out.println(s.length());//NullPointerException

Some Common Exceptions

- NumberFormatException
  - The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.
  - String s="abc";

    int i=Integer.parseInt(s);//NumberFormatException
- ArrayIndexOutOfBoundsException
  - If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException
  - int a[]=new int[5];

    a[10]=50; //ArrayIndexOutOfBoundsException

Exception Handling

- Maintain the normal flow of the application even though exception occurs.
- Exception Handling is mainly used to handle the unchecked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

```
statement 1;
statement 2;
statement 3; //exception occurs
statement 4;
statement 5;
statement 6;
```

Exception Handling Keywords

- try
- catch
- finally
- throw
- throws

Exception Handling: ***try-catch***

- ***try*** block
  - It contains the segment of code where exception may occur.
  - It must be used within the method.
  - Java try block must be followed by either catch or finally block.
- ***catch*** block
  - Java catch block is used to handle the exception.
  - It must be used after the try block only.
  - You can use multiple catch block with a single try.

Exception Handling: ***try-catch***

- Syntax of java try-catch

```
try{
//code that may throw exception
}
catch(Exception_class_Name ref){
//customized message to inform user
}
```

Code without Exception Handling

```
public class Testtrycatch{
public static void main(String args[]){
int data=50/0;//may throw exception
System.out.println("rest of the code...");
}
}
```

Output: Exception in thread main java.lang.ArithmeticException:/ by zero

Code with Exception Handling

```
public class Testtrycatch{
public static void main(String args[]){
try{
int data=50/0;//may throw exception
}
catch(ArithmaticException e){
System.out.println(e);
}
System.out.println("rest of the code...");
}
}
```

Output:
```
Exception in thread main java.lang.ArithmeticException:/ by zero
        rest of the code...
```

Multiple ***catch***

- At a time only one Exception is occured and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception otherwiseit will show Compile-time error

Multiple ***catch***

```
public class TestMultipleCatchBlock{
public static void main(String args[]){
try{
int a[]=new int[5];
a[5]=30/0;
}

catch(ArithmeticException e){
System.out.println("Arithmetic Exception");}
catch(ArrayIndexOutOfBoundsException e){
System.out.println("Array Index Out Of Bounds Exception");}

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

System.out.println("rest of the code...");
} }
```

Output:
```
Array Index Out Of Bounds Exception
rest of the code...
```

Nested *try*

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
class NestExcep
{
public static void main(String args[]){
try{
            try{int b =39/0;}
            catch(ArithmeticException e){System.out.println(e);}
            try{ int a[]=new int[5]; a[5]=4;}
            catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
            System.out.println("other statement);
}
catch(Exception e){System.out.println("handeled");}
System.out.println("normal flow..");
} }
```

Output:

```
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
other statement
normal flow..
```

finally block

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block must be followed by try or catch block.
- For each try block there can be one or more catch blocks, but only one finally block.
- The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

finally: exception doesn't occur

```
class TestFinallyBlock{
public static void main(String args[]){
try{
int data=25/5;
System.out.println(data); }
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
} }
```

Output:

```
5
finally block is always executed
rest of the code...
```

finally: exception occurs and not caught

```
class TestFinallyBlock{
public static void main(String args[]){
try{
int data=25/0;
System.out.println(data); }
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
} }
```

Output:

```
finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero
```

finally: exception occurs and caught

```
class TestFinallyBlock{
public static void main(String args[]){
try{
int data=25/0;
System.out.println(data); }
catch(ArithmeticException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
} }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...
```

Use of ***throw***

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.

throw new IOException("sorry device error”);

throw keyword example

```
public class TestThrow{
static void validate(int age){
if(age<18)
throw new ArithmeticException("not valid");
else
System.out.println("welcome to vote"); }
public static void main(String args[]){
validate(13);
System.out.println("rest of the code...");
} }
```

Output:
```
Exception in thread main java.lang.ArithmeticException:not valid
```

Use of ***throws***

- The Java throws keyword is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

```
<return_type> <method_name>() throws <exception_class>{
//method code }
```

throws keyword example

```
import java.io.IOException;
class Testthrows{
void m() throws IOException{
throw new IOException("device error");
}//checked exception
void n()throws IOException{
m();
}
void p(){
try{
n();}
catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[]){
Testthrows obj=new Testthrows();
obj.p();
System.out.println("normal flow...");
} }
```

Output:

```
exception handled
normal flow...
```

Facts of using ***throws***

- You declare the exception-
  - In case you declare the exception, if exception does not occur, the code will be executed fine.
  - In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

Declared exception but not occurred

```
import java.io.*;
class M{
void method()throws IOException{
System.out.println(“Exception is not thrown");} }
class Testthrows{
public static void main(String args[])throws IOException{//declare exception
M m=new M();
m.method();
System.out.println("normal flow...");
} }
```

Output:

```
Exception is not thrown
normal fow…
```

Declared exception and it occurred

```
import java.io.*;
class M{
void method()throws IOException{
throw new IOException("device error");
} }
class Testthrows{
public static void main(String args[])throws IOException{//declare exception
M m=new M();
m.method();
System.out.println("normal flow...");
} }
```

Output:
```
Exception in thread "main" java.io.IOException: device error
```

Create Your Own Exception

```
class InvalidAgeException extends Exception{
InvalidAgeException(String s){
super(s);
}
}
class TestCustomException{
static void validate(int age)throws InvalidAgeException{
if(age<18)
throw new InvalidAgeException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[]){
try{
validate(13);
}
catch(Exception m){
System.out.println("Exception occured: "+m);
}
System.out.println("rest of the code...");
}
}
```

Output:

```
Exception occured: InvalidAgeException:not valid
rest of the code...
```