



CS205 Object Oriented Programming in Java

Module 3 - More features of Java (Part 7)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics



- **More features of Java :**

- ☑ **Exception Handling:**

- Multiple **catch** Clauses
 - Nested **try** Statements

Multiple catch Clauses



- There can be more than one exception in a single piece of code.
 - To handle this type of situation, we can specify two or more **catch** clauses, each catching a *different type of exception*.
- When an exception is thrown,
 - each catch statement is inspected in order, and
 - the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the other catch statements are bypassed(ignored), and execution continues after the **try/catch** block.

Multi catch-Example



```
class Multicatch {  
    public static void main(String args[]) {  
        try {  
            int a = args.length;           //number of commandline arguments  
            System.out.println("a = " + a);  
            int b = 42 / a;                 //when a is 0 this will raiseAthmeticException  
            int c[] = { 1 };  
            c[42] = 99; //size of array is 1. So c[42] leds to ArrayIndexOutOfBoundsException  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Divide by 0: " + e);  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("Array index oob: " + e);  
        }  
        System.out.println("After try/catch blocks.");  
    }  
}
```

Here the value of a is set as the number of command line arguments. If no command line arguments are there during execution

E.g. **java MultiCatch**

Here a is 0

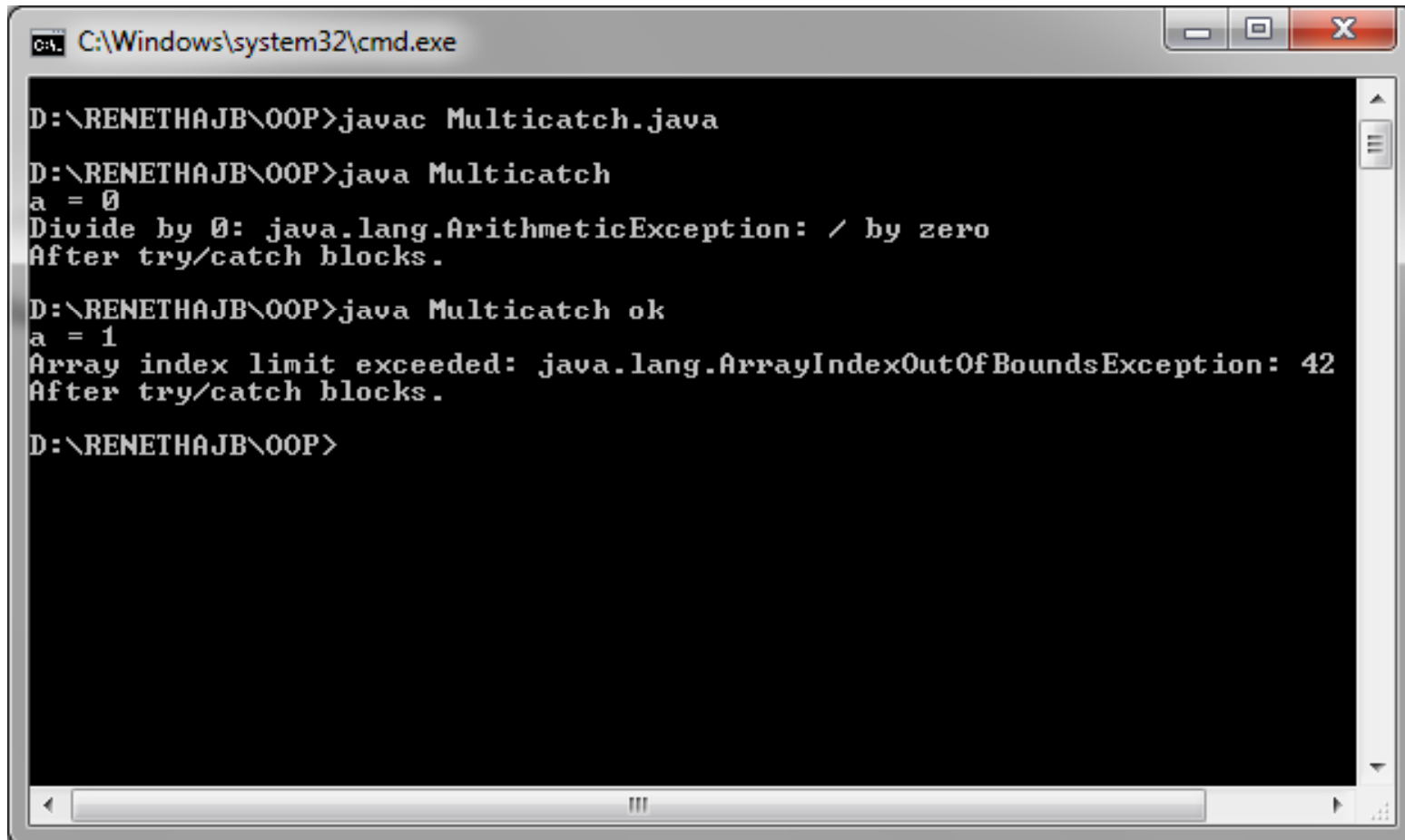
So **int b = 42 / a;** will cause ArithmeticException. and is caught by **catch(ArithmeticException e).**

If command line arguments are there ,then a is not zero. E.g. **java MultiCatch ok**
(Here a=1. So no exception occurs in int b = 42 / a)

Size of array c is 1 (only one element).

So **c[42] = 99;** will cause ArrayIndexOutOfBoundsException occurs(because position 42 is not there in this array)

- Output



```
C:\Windows\system32\cmd.exe

D:\RENETHAJB\OOP>javac Multicatch.java
D:\RENETHAJB\OOP>java Multicatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

D:\RENETHAJB\OOP>java Multicatch ok
a = 1
Array index limit exceeded: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.

D:\RENETHAJB\OOP>
```

Multi-catch (contd.)



- When we use multiple **catch statements**, it is important that **exception subclasses** must come before any of their superclasses.
- If we are using catch with superclass exception before the catch with subclass exception then catch with subclass exception will be ignored.
 - Such codes are unreachable. Unreachable code is an **ERROR**.



- E.g. Exception class is the superclass of all other exception classes like ArithmeticException, FileNotFoundException etc.

```
try
{
//statements
}
catch(Exception e)           //ALL EXCEPTIONS WIL BE CAUGHT HERE
{ //statements
}
catch(ArithmeticException ae) //This catch is never used for catching
{ //statements
}
```

Any exception that occurs in try block will be caught by the **first suitable** catch. Here all exceptions will match with **Exception** object. So even though ArithmeticException occurs inside try block, it will be caught by catch(Exception e) block. So catch(ArithmeticException ae) will never catch it.

Multi catch(ERROR) // superclassexception should not be caught before catching subclass



```
class SuperSubCatch {  
    public static void main(String args[])  
    {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        }  
        catch(Exception e)    //All exceptions are caught here  
        {System.out.println("Generic Exception catch.");  
        }  
        /* The next catch is never reached because  
        ArithmeticException is a subclass of Exception. */  
        catch(ArithmeticException e)  
        {    // ERROR - unreachable  
            System.out.println(" Arithmetic Exception occurred ");  
        }  
    }  
}
```

COMPILE ERROR- the second catch statement is unreachable because the exception has already been caught by Exception

A subclass must come before its superclass in a series of catch statements.



```
class SuperSubCatch {  
    public static void main(String args[])  
    {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println(" Arithmetic Exception occurred ");  
        }  
        catch(Exception e)  
        { System.out.println("Generic Exception catch.");  
        }  
    }  
}
```

This is the correct usage of catch. The catch with subclass exception(ArithmeticException) should appear before catch with super class exception(Exception)

Nested *try* Statements



- The **try** statement can be nested.
 - A **try** statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
 - If an **inner try** statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
 - This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
 - If **no catch statement matches**, then the Java run-time system will handle the exception.



```
class NestTry {
public static void main(String args[]) {
try {
    int a = args.length;
    int b = 42 / a;
    System.out.println("a = " + a);
    try {
        if(a==1) a = a/(a-a);           // division by zero
        if(a==2)
        { int c[] = { 1 };
          c[42] = 99;                   // generate an out-of-bounds exception
        }
    } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
    }
}
catch(ArithmeticException e) {
    System.out.println("Divide by 0: " + e);
}
}
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block.

Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block.

Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled.

If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

Nested try(contd.)



- We can enclose a call to a method within a **try** block.
 - Inside that method we can have another try statement.
- In this case, the try within the method is still nested inside the outer try block, which calls that method.



```
class MethNestTry {
static void show(int a) {
try {           // nested try block

if(a==1) a = a/(a-a);    // division by zero
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}

public static void main(String args[]) {
try {
    int a = args.length;
    int b = 42 / a;
    System.out.println("a = " + a);
    show(a); // show contains a try – catch . So nested try.
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

Here try in main function act as outer try block. Inside that try show() function is called . So try catch inside show() function is **inner** to the try in main function.

When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block and is caught by outer catch clause(matching is there).

Execution of the program with one command-line argument generates a divide-by-zero exception from within the try block in show().

Since the inner catch(no matching) block does not catch this exception, it is passed on to the outer try block in main function(matching is there) , and it is handled.

If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block and is caught by innercatch inside show

Reference



- **Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.**