Topics

- **More features of Java :**

☑ **Exception Handling:**

  - **Checked Exceptions**
  - **Unchecked Exceptions**
  - ***try*** Block and ***catch Clause***

Exception Handling

- An ***exception*** *is* an **abnormal condition** that occur in a code sequence at ***run time*.**
  - Exception is a RUN TIME ERROR
- A Java exception is an **object** that describes an exceptional (that is, error) condition that occurred in a piece of code.
- When an exceptional condition arises,
  - an object representing that exception is created and
  - *It is thrown in the method that caused the error.*
    - That method may choose to handle the exception itself, or pass it on.
    - The exception is then *caught and processed*

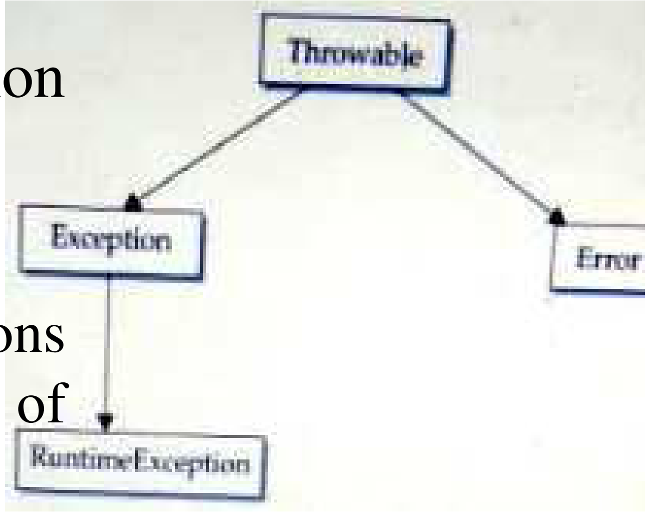
Exception Handling(contd.)

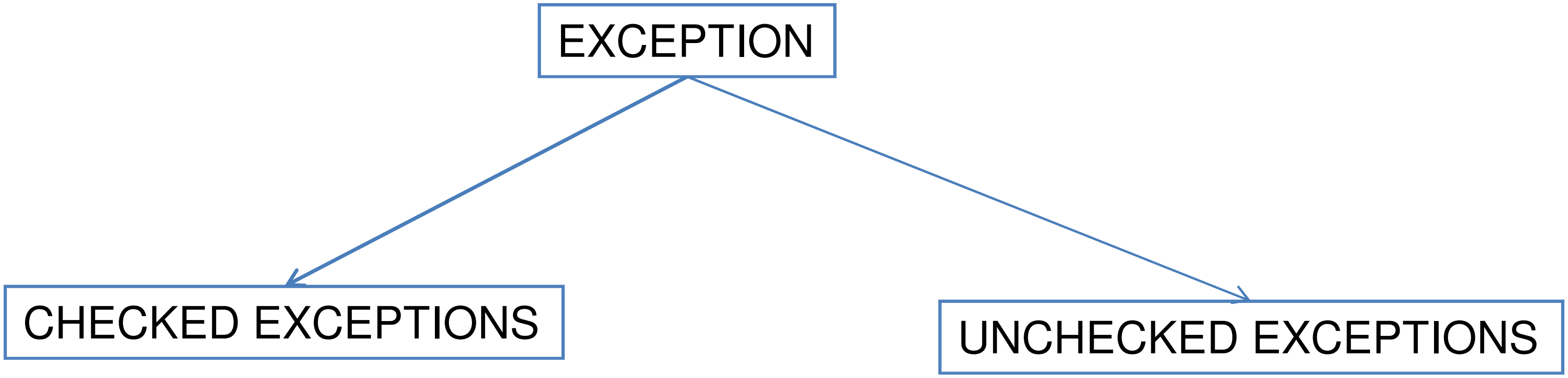
- Exceptions can be **generated by**
  - the Java run-time system, or
  - they can be manually generated by your code.
- Exceptions thrown by Java are related to
  - **Fundamental errors** that **violate the rules** of
    - the Java language or
    - the constraints of the Java execution environment.

Exception Types

- All exception types are subclasses of the built-in class **Throwable.**
- **Throwable** has two subclasses that partition exceptions into two distinct branches.
  - ❑ One branch is headed by **Exception.**
    - This class is used for exceptional conditions that ***user programs should catch***. Subclass of this helps to create custom exception types.
    - **RuntimeException** is a subclass of **Exception.**
  - ❑ The other branch is headed by **Error**
    - This defines exceptions that are ***not expected to be caught*** under normal circumstances by our program.(***unchecked***)
    - Exceptions of type Error are used by the Java run-time system to <u>indicate errors.</u>

      <u>E.g. </u>Stack overflow,Out of Memory error





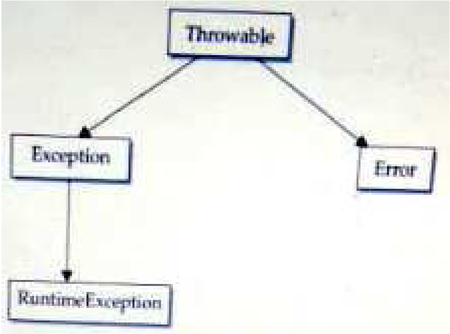
EXCEPTION

CHECKED EXCEPTIONS

UNCHECKED EXCEPTIONS

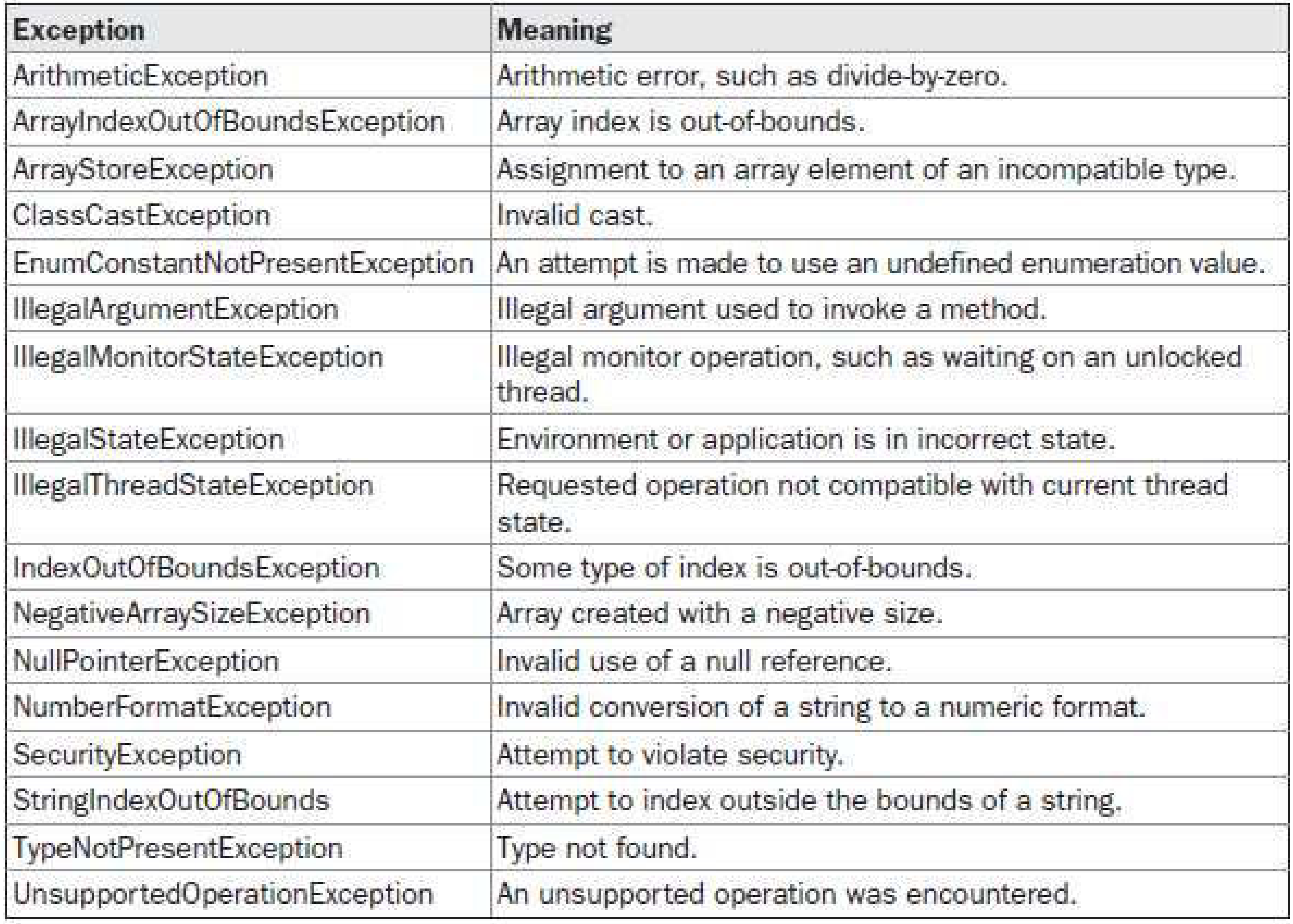
Unchecked exception

- Unchecked exception classes are defined inside **java.lang** package.
  - The **unchecked exceptions** are subclasses of the standard type **RuntimeException.**
  - In the Java language, these are called ***unchecked exceptions*** *because the* ***compiler does not check*** *to see whether there is a method that handles or* throws these exceptions.
  - If the program has unchecked exception then it will *compile without error* but exception occurs when program runs.



•E.g Exceptions under Error , ArrayIndexOutOfBoundException

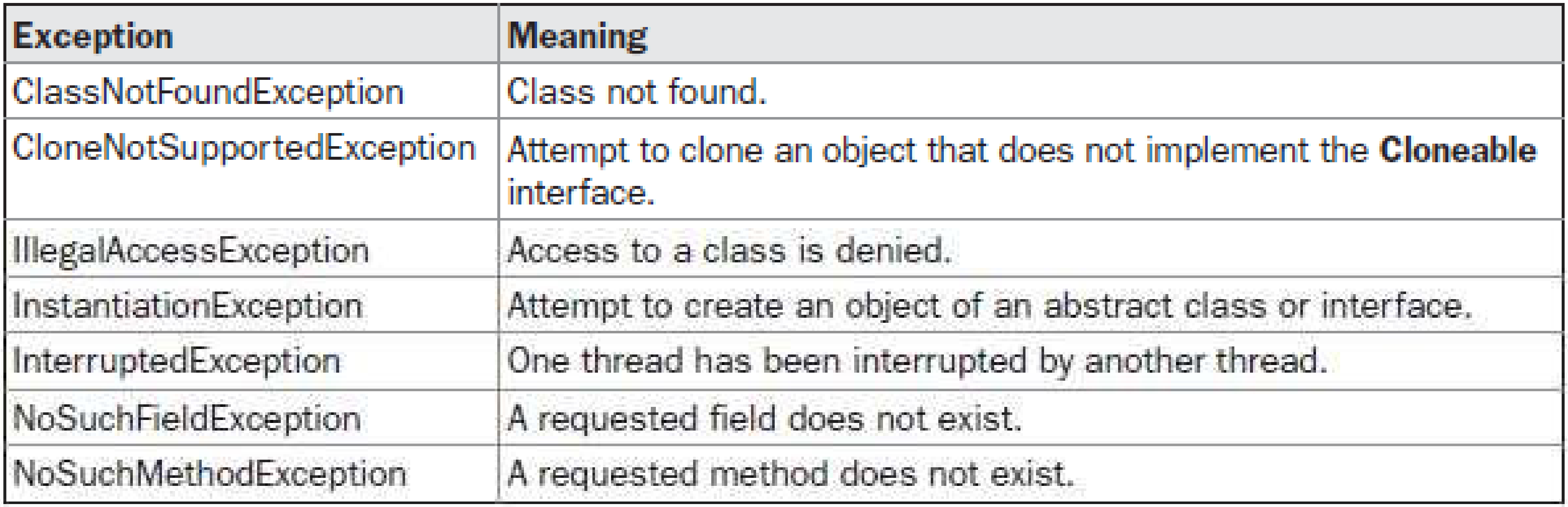
Unchecked exception(contd.)



Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Checked exception

- There are some exceptions that are defined by **java.lang** that **must be included in a method's throws list,** if a method generates such exceptions and that ***method does not handle it itself***. These are called **checked exceptions**



Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

- **IOException**
- **FileNotFoundException**
- **SQLException**

Checked exception(contd.)

- Checked exceptions are the exceptions (in java.lang) that are checked at compile time.
  - If some statement in a method **throws a checked exception**, then that method must
    - either handle the exception or
    - it must specify the exception using *throws* keyword.

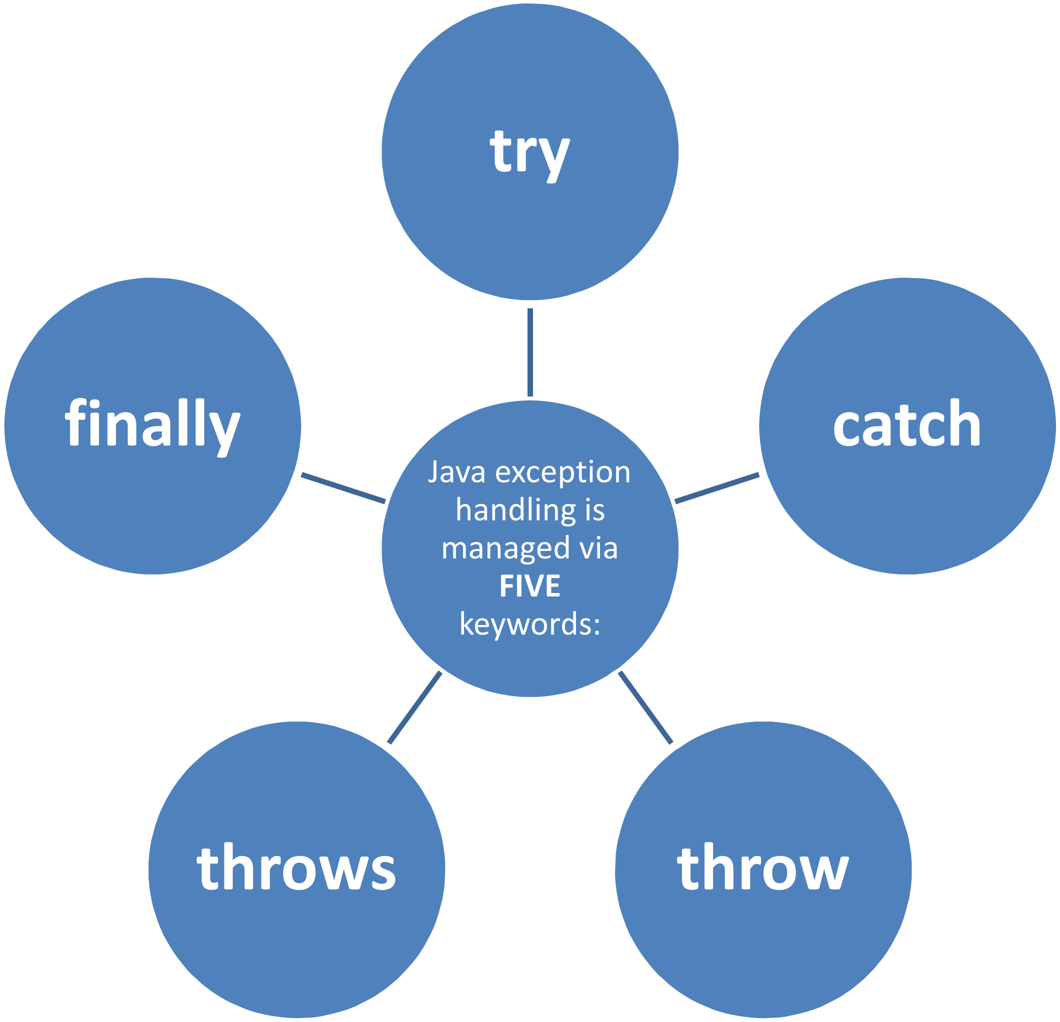
Checked exceptions

- Checked at compile time.(COMPILE TIME EXCEPTIONS)
- Not sub class of RunTimeException
- The method must either handle the exception or it must specify the exception using *throws* keyword.
- Shows compile error if checked exception is not handled.
- E.g. *ClassNotFoundException, IOException*

Unchecked exceptions

- NOT checked at compile time.(RUN TIME EXCEPTINS)
- Sub class of RunTimeException
- It is NOT needed to handle or catch these exceptions
- DO NOT Show compile error if exception is not handled. But shows run-time error.
- Eg. *ArithmeticException, ArrayIndexOutOfBoundsException*

Exception handling fundamentals



Exception handling fundamentals(contd.)

- Program statements that we want to check for exceptions are written within a **try block.**
  - If an exception occurs within the try block, it is **thrown**.
  - The code inside **catch** can catch this exception and handle it in some manner.
- ***System-generated exceptions*** are automatically thrown by the Java run-time system.
- To **manually throw** an exception, use the keyword **throw.**
- Any exception that is thrown out of a **method** must be specified as such by a **throws** clause**.**
- Any code that absolutely must be executed after a **try block completes** is put in a **finally block.**

```
try {
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
// ...
finally
{
    // block of code to be executed after try block ends
}
```

Here, ExceptionType is the type of exception that has occurred.

Uncaught Exceptions

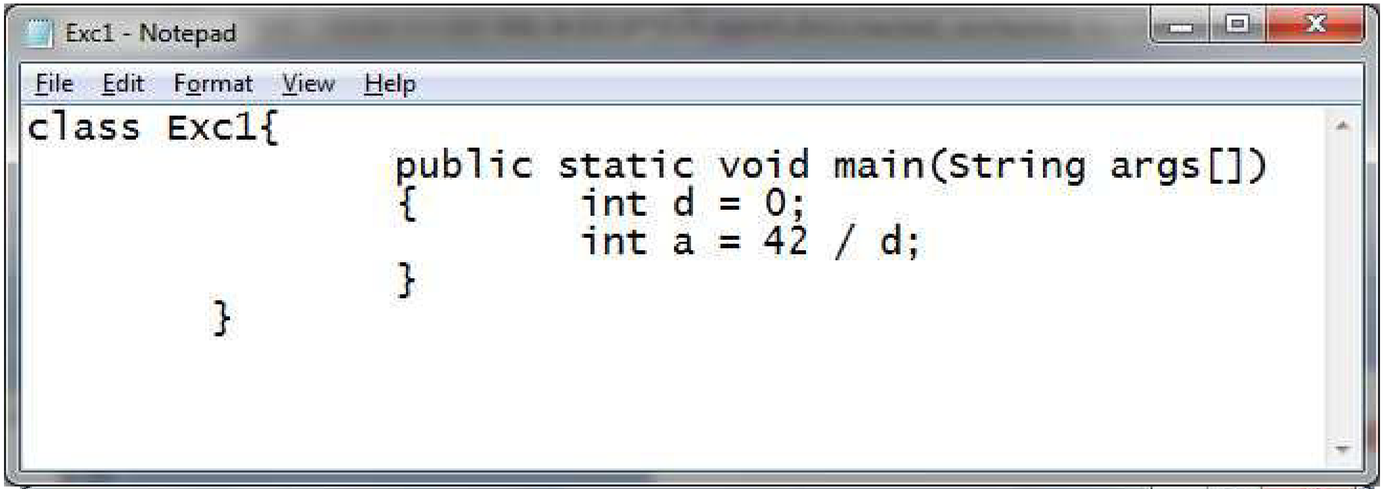
- Consider the program

```
Lineno.1            class Ex{
Lineno.2            public static void main(String args[])
Lineno.3                        {       int d = 0;
Lineno.4                                int a = 42 / d;
Lineno.5                        }
Lineno.6                    }
```

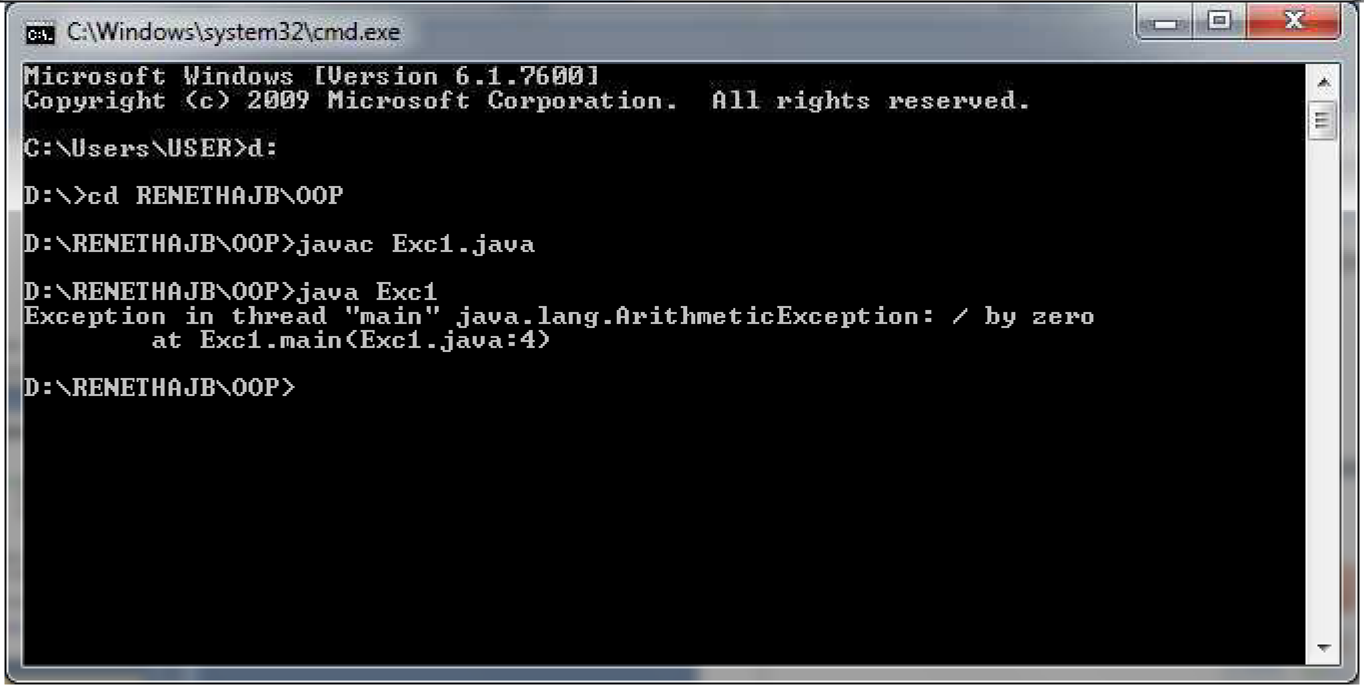
- This small program causes a ***divide-by-zero error***((42/0)
- Java run time system constructs a new exception object and then *throws this exception.*
- *Th erogram stops by showing the following exception(run time erroe)*
- java.lang.ArithmeticException: / by zero at **Ex**.**main**(**Ex**.java:**4**)

- java.lang.**ArithmeticException**: / by zero at **Ex**.**main**(**Ex**.java:**4**)

- Here **Ex** is the class name , **main** is the method name,**;** **Ex.**java is the file name**;** and the exception is inline number **4.**
- These details are all included in the simple stack trace.
- The type of exception thrown is a subclass of Exception called **ArithmeticException** (describes what type of error happened.)



```
class Exc1{
                public static void main(String args[])
                {       int d = 0;
                        int a = 42 / d;
                }
        }
```



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\USER>d:

D:\>cd RENETHAJB\OOP

D:\RENETHAJB\OOP>javac Exc1.java

D:\RENETHAJB\OOP>java Exc1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exc1.main(Exc1.java:4)

D:\RENETHAJB\OOP>
```

```
Lineno.1          class Exc1 {
Lineno.2          static void subroutine()
Lineno.3           { int d = 0;
Lineno.4             int a = 10 / d;
Lineno.5            }
Lineno.6          public static void main(String args[])
Lineno.7           { Exc1.subroutine();
Lineno.8            }
Lineno.9          }
```

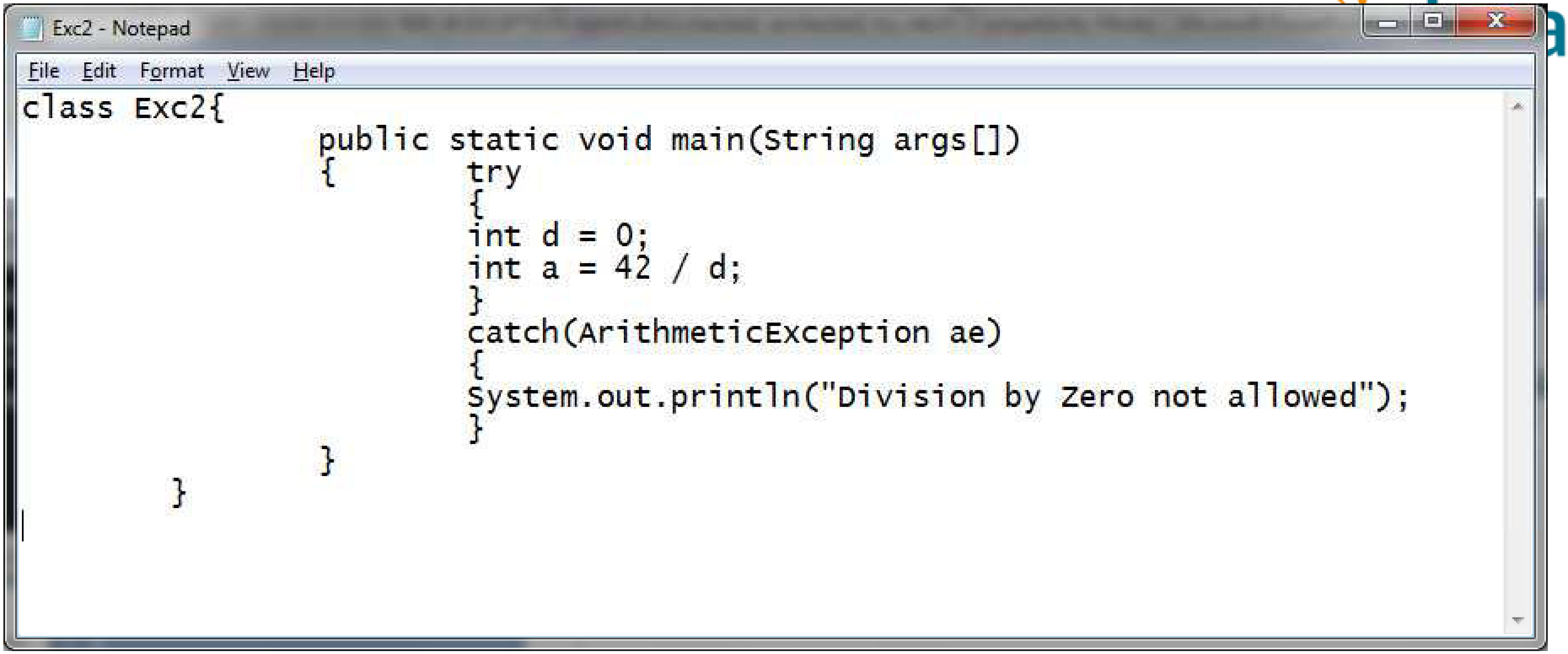
```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

try Block and *catch Clause*

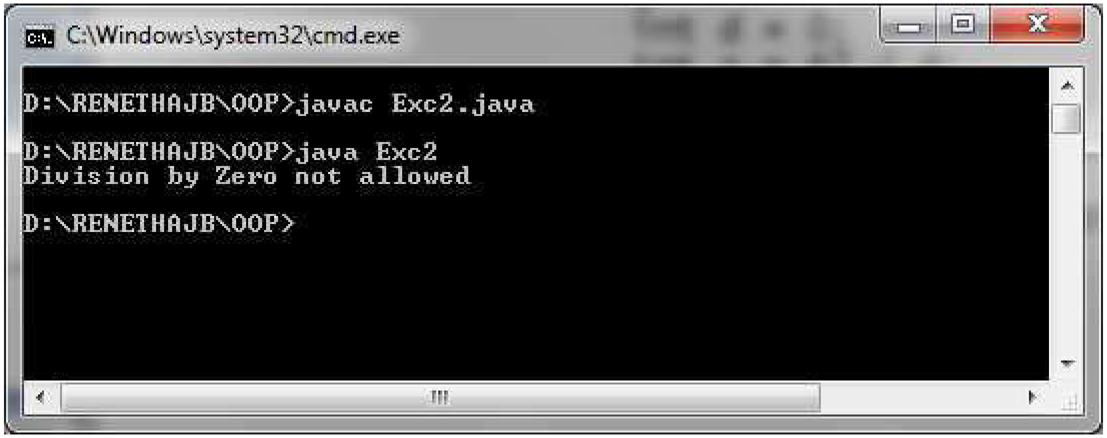
- Benefits of exception handling
  - First, it allows us to **fix the error.**
  - Second, it **prevents** the program from **automatically terminating.**
- To **guard against** and **handle a run-time error**, simply enclose the code that we want to monitor inside a **try block.**
- Immediately ***after the try block***, there is a **catch clause** that **specifies the exception type** that we wish to catch . The catch block can process that exception..

```
class Exc2{
        public static void main(String args[])
        {       try
                {
                int d = 0;
                int a = 42 / d;
                }
                catch(ArithmeticException ae)
                {
                System.out.println("Division by Zero not allowed");
                }
        }
   }
```

```
class Exc2{
        public static void main(String args[])
        {       try
                {
                int d = 0;
                int a = 42 / d;
                }
                catch(ArithmeticException ae)
                {
                System.out.println("Division by Zero not allowed");
                }
        }
   }
```



```
class Exc2{
                public static void main(String args[])
                {       try
                        {
                        int d = 0;
                        int a = 42 / d;
                        }
                        catch(ArithmeticException ae)
                        {
                        System.out.println("Division by Zero not allowed");
                        }
                }
        }
```



```
D:\RENETHAJB\OOP>javac Exc2.java

D:\RENETHAJB\OOP>java Exc2
Division by Zero not allowed

D:\RENETHAJB\OOP>
```

try-catch block to handle division by zero exception

```
class Ex {
public static void main(String args[]) {
int d, a;
try {                                        // monitor a block of code.
    d = 0;
    a = 42 / d;
    System.out.println("This will not be printed.");
   }
   catch (ArithmeticException e)             // catch divide-by-zero error
   {
   System.out.println("Division by zero.");
   }
System.out.println("After catch statement.");
}
}
```

OUTPUT

```
Division by zero.
After catch statement.
```

Working of the program

- In this program the *System.out.println("This will not be printed.");* inside the try block is never executed because a = 42 / d;

- Once an exception is thrown, program control transfers out of the try block into the catch block.
  - i.e. catch is not "called" but controls goes out to catch when exception occurs, so execution never "returns" to the try block from a catch.
  - Thus, the line "This will not be printed." is not displayed.

try-catch (contd.)

- A try and its catch statement form a unit.
- The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.
  - Each catch block can catch exceptions in statements inside immediately preceding try block.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.)
- We cannot use try on a single statement

try-catch Example

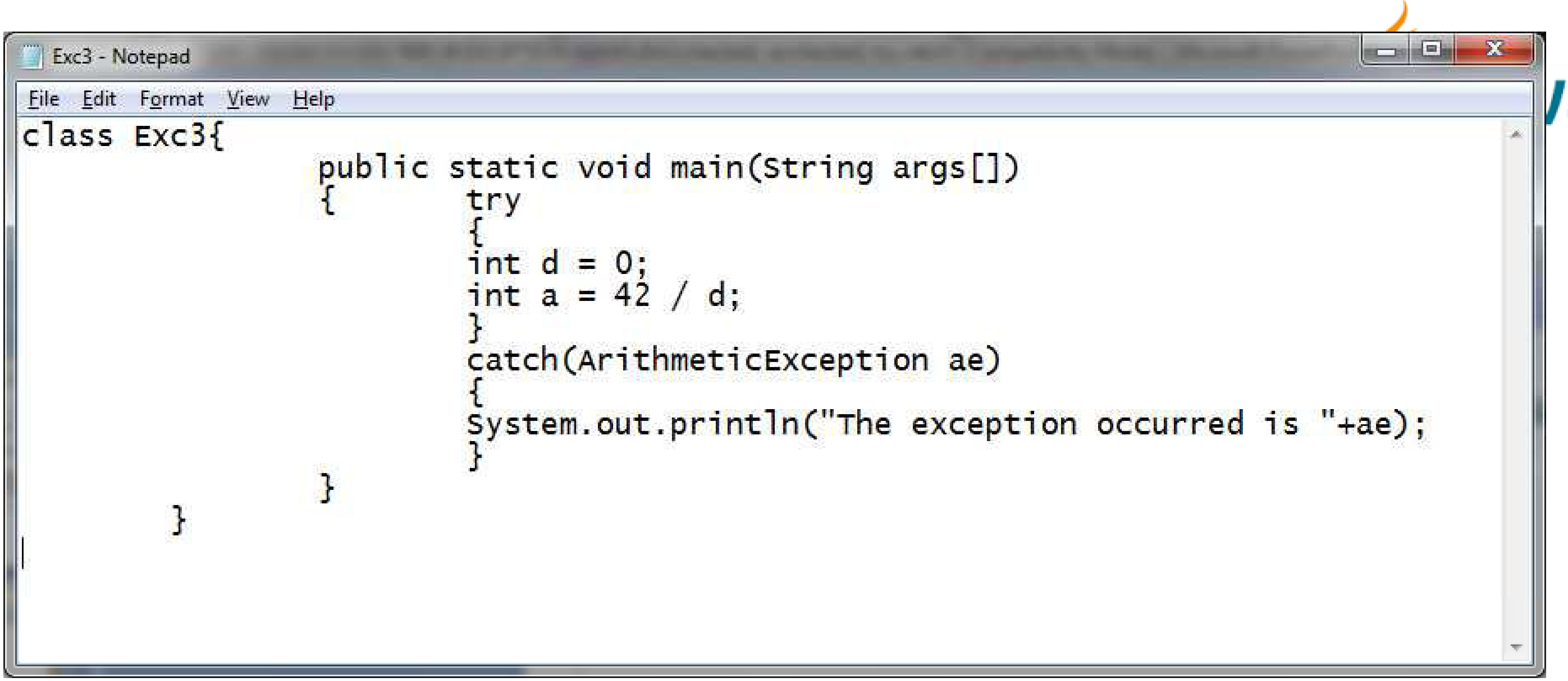
```
import java.util.Random;
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try {
      b = r.nextInt();
      c = r.nextInt();
      a = 12345 / (b/c);
     }
catch(ArithmeticException e)
{
      System.out.println("Division by zero.");
      a = 0;                    // set a to zero and continue
}
System.out.println("a: " + a);
} } }
```

Here **b** and **c** are random numbers .
If the value of b or c becomes zero then
a=12345./ (b/c) becomes
a=12345/0;
(Division by zero(ArithmeticException) will occur)
This statement is inside try block
So exception will be caught by catch and prints message ***Division by zero.*** *and set the value of a to 0* and proceeds
NO RUNTIME ERROR!!

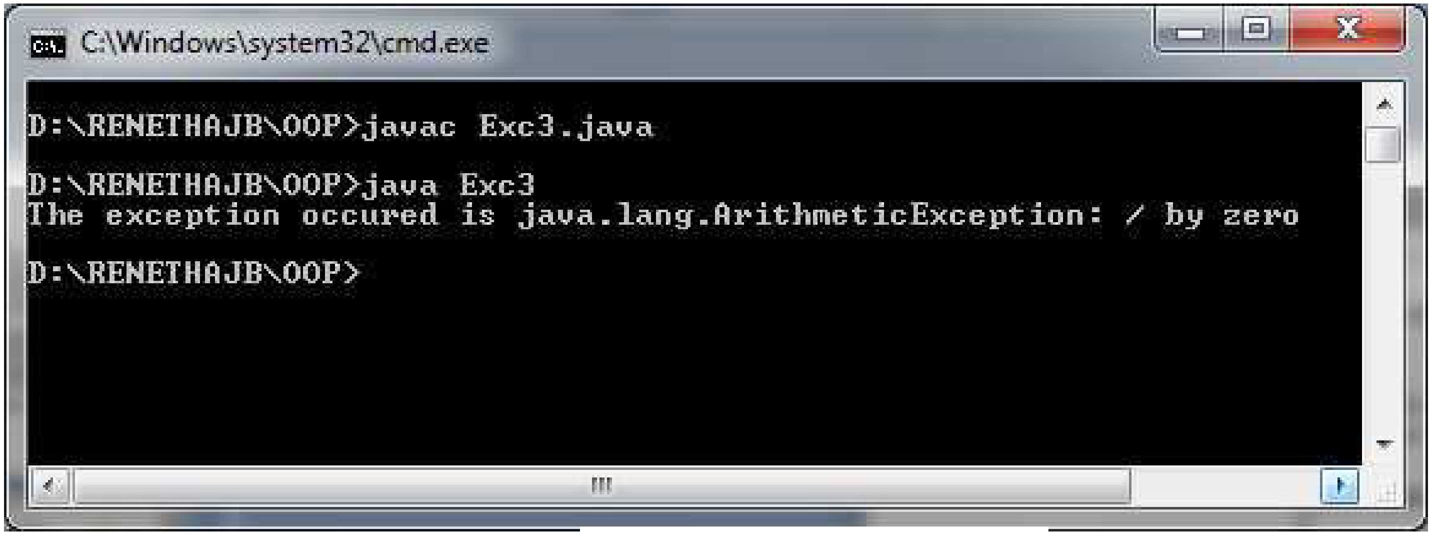
Displaying a Description of an Exception

- We can display this description in a **println( )** statement by simply passing the exception as an argument.

```
class Exc3{
        public static void main(String args[])
        {       try
                {
                int d = 0;
                int a = 42 / d;
                }
                catch(ArithmeticException ae)
                {
                System.out.println("The exception occurred is "+ae);
                }
        }
   }
```



```
class Exc3{
                public static void main(String args[])
                {       try
                        {
                        int d = 0;
                        int a = 42 / d;
                        }
                        catch(ArithmeticException ae)
                        {
                        System.out.println("The exception occurred is "+ae);
                        }
                }
        }
```



```
D:\RENETHAJB\OOP>javac Exc3.java

D:\RENETHAJB\OOP>java Exc3
The exception occured is java.lang.ArithmeticException: / by zero

D:\RENETHAJB\OOP>
```

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 ***<u>different type of exception</u>***.
- When an exception is thrown,
  - each catch statement is inspected <u>in order,</u> 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.

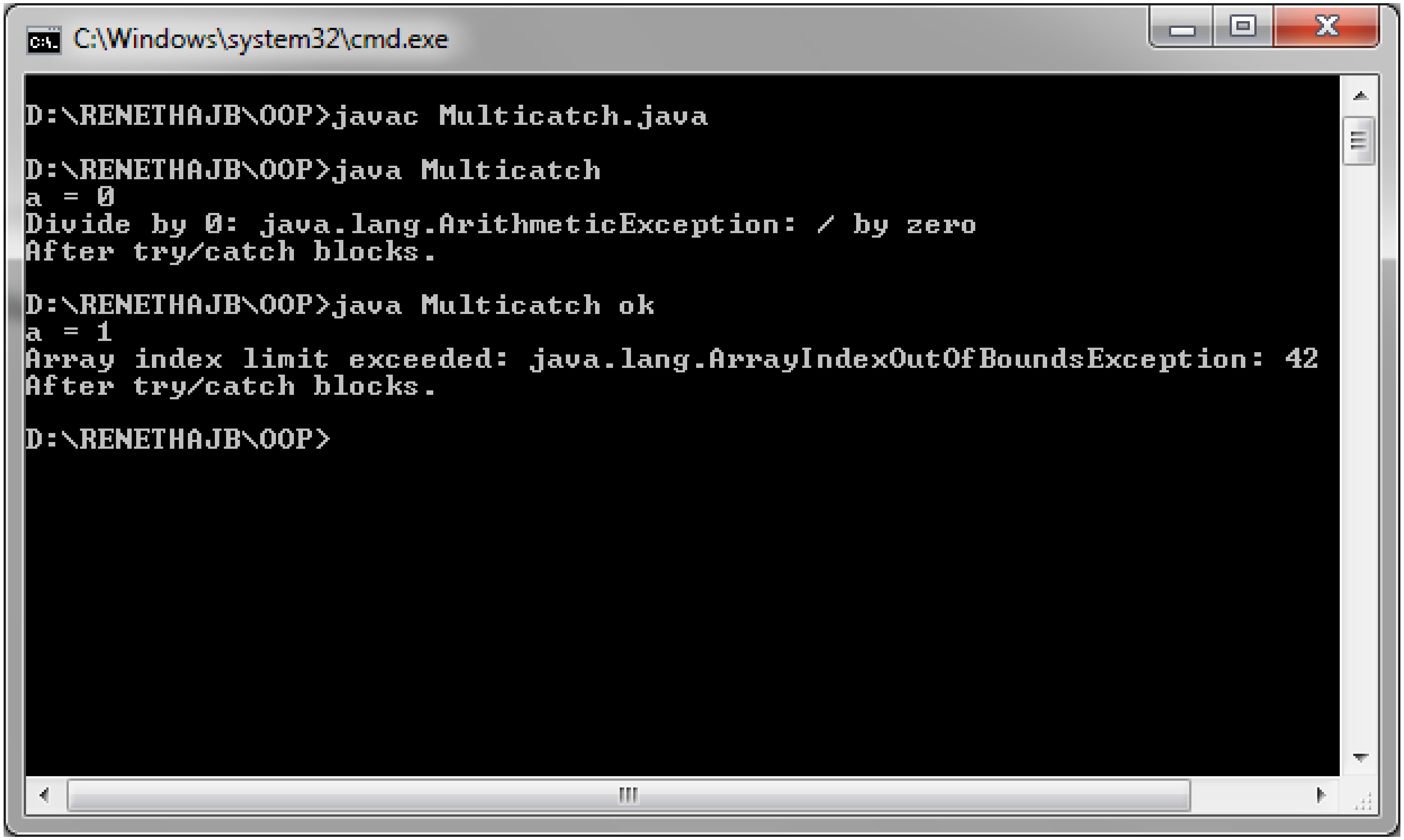
Multi catch-Example

```
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 is1 (only one element).
So c[42] = 99; will cause ArrayIndexOutofBoundsException occurs(because position 42 is not there in this array)

- Output



```
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 ArithmeticExcepton 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(AritnmeticException) 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

Topics

- **More features of Java :**

☑ **Exception Handling:**

- ***throw***
- ***throws***
- ***finally***

throw statement

- Our program can throw an exception explicitly, using the **throw** statement.

- The general form of **throw is shown here:**

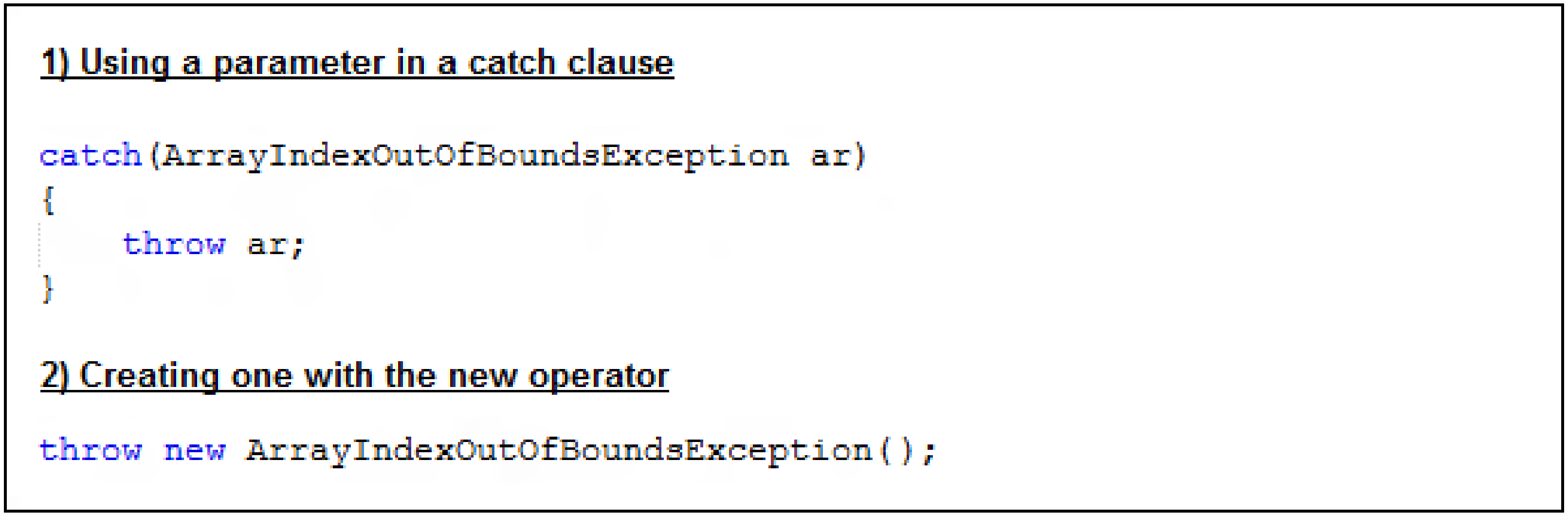
```
throw ThrowableInstance;
```

  - *ThrowableInstance* must be an **object** of type ***Throwable*** or a ***subclass of Throwable.***
  - Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

throw(contd.)

- Two ways to obtain a **Throwable object:**

1. using a parameter in a **catch** clause, or
2. creating one with the **new** operator



1) Using a parameter in a catch clause

```
catch(ArrayIndexOutOfBoundsException ar)
{
    throw ar;
}
```

2) Creating one with the new operator

```
throw new ArrayIndexOutOfBoundsException();
```

throw statement(contd..)

- The *flow of execution* **stops** immediately after the **throw** statement.
  - Any statements after throw statement will not be executed.
- When exception is thrown using **throw** statement :-
  - the nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception thrown.
    - If that catch statement has a **matching exception type as the thrown exception**, control is transferred to that statement.
    - If **not matching**, then the *next enclosing try statement is inspected, and so on.*
    - If **no matching catch is found**, then the *default exception handler halts the program and prints the stack trace.*

throw Example 1

```
class ThrowDemo
{
static void show()
{
    try
    {throw new NullPointerException("demoexception");
    }
    catch(NullPointerException e)
    {
     System.out.println("Caught inside show");
     throw e;      // rethrow the exception
     }
}
public static void main(String args[])
{
     try {
     show();
         }
     catch(NullPointerException e)
     { System.out.println("Recaught in main: " + e);
     }
}  }
```

Here first **throw** in show is caught by matching catch is in show function.

Next throw has no immediate catch So since the exception matches with catch in the **main** function(that calls show), the exception is caught by that matching catch in main.

```
java ThrowDemo
Caught inside show
Recaught in main: java.lang.NullPointerException: demoexception
```

throw with matching catch in calling function

```
class ThrowDemo2
{
   static void show()
   {
          throw new NullPointerException("demoxception");
   }
    public static void main(String args[])
    {
          try
          {
          show();
          } catch(NullPointerException e)
          { System.out.println("Caught in main: " + e);
          }
      }
}
```

Here no matching catch for throw is in show function
So since the exception matches with **catch** in the
main function(that calls show), the exception
is caught by that matching catch

OUTPUT

```
Caught in main: java.lang.NullPointerException: demoexception
```

throw with NO matching catch

```
class ThrowDemo2
{
   static void show()
   {
          throw new NullPointerException("demoxception");
   }
    public static void main(String args[])
    {
          try
          {
          show();
          }
           catch(ArithmeticException e)
          { System.out.println("Caught in main: " + e);
          }
      }
}
```

Here no matching catch is in show function.
So since the exception does not matches
with catch in the main function(that calls show)
also, the exception is not caught in the program
the ***default exception handler halts the program and prints the stack trace***

OUTPUT

```
Exception in thread "main" java.lang.ArithmeticException: demoxception
        at ThrowDemo2.show(ThrowDemo2.java:3)
        at ThrowDemo2.main(ThrowDemo2.java:9)
```

throw(contd.)

- Many of Java's built-in run-time exceptions have at least two constructors:
  - one with no parameter and
  - one that takes a string parameter.
- When constructor with string parameter is used, the argument specifies a **string <u>that describes the exception</u>**.
  - This string is displayed when the object is printed using **print( ) or println( ).**
  - It can also be obtained by a call to getMessage( ), which is defined by Throwable.

throw new ***NullPointerException***("**demoxception**");

- Here the string **demoxception** inside the constructor of ***NullPointerException*** is the name of the exception.

throws

- A **throws** clause lists the types of exceptions that a **method(function) might throw.**
- **throws** keyword is used with the method signature(header)
- If a method has an exception and it does not handle that exception, it must specify this using **throws** , so that callers of the method can guard themselves against that exception.
- **throws** is necessary for all exceptions, except those of type **Error** or **RuntimeException** or any of their subclasses

throws (contd.)

- All other **exceptions** that a **method can throw** must be declared in the **throws** clause**.**
  - If they are not, a compile-time error will result.
- General form of a method declaration that includes a **throws clause:**

type method-name(parameter-list) ***throws*** *exception-list*

```
{
    // body of method
}
```

throw statement but no throws in method-ERROR

```
public class ThrowsEg {
  static void vote(int age) {
    if (age < 18) {
      throw new IllegalAccessException("You must be at least 18 years old.");
                          } else {
                            System.out.println(“ You can vote!");
                                        }
  }
  public static void main(String[] args)
  {
               vote(15);
  }
}
```

```
D:\RENETHAJB\OOP>javac ThrowsEg.java
ThrowsEg.java:4: unreported exception java.lang.IllegalAccessException; must be
caught or declared to be thrown
    throw new IllegalAccessException("You must be at least 18 years old.");
    ^
1 error
```

COMPILE ERROR

This program tries to throw an exception that it does not catch.

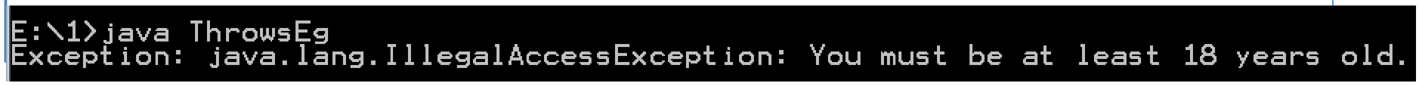
Because the program does not specify a **throws clause to declare this exception to be thrown,** the program will not compile.

Include **throws** in method and **try catch** in calling function.

Using throws

```
public class ThrowsEg {
  static void vote(int age) throws IllegalAccessException{
    if (age < 18) {
      throw new IllegalAccessException("You must be at least 18 years old.");
                          } else {
                                    System.out.println(“ You can vote!");
                                    }
 }
 public static void main(String[] args)
 {
  try{
            vote(15);
      }
      catch(Exception e)
      {
     System.out.println("Exception: "+e);
      }
 }
}
```

OUTPUT



```
E:\1>java ThrowsEg
Exception: java.lang.IllegalAccessException: You must be at least 18 years old.
```

```
import java.io.*;
class Sample{
 void show() throws IOException{
  throw new IOException(“Thrown IO error");
 }
}
public class Testthrows{
   public static void main(String args[]){
    try{
     Sample s=new Sample();
     s.show();
    }
catch(Exception e){System.out.println("Exception handled. "+e);}

    System.out.println("Normal program flow");
  }
}
```

Output

```
Exception handledjava.io.IOException: Thrown IO error
Normal program flow
```

finally

- **finally** creates a block of code that will be executed **after** a try/catch block has completed and **before** the control goes out from the try/catch block.

```
try {
// block of code to monitor for errors
      }
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{   // exception handler for ExceptionType2
}
// ...
finally
{   // block of code to be executed after try block ends
}
```

Why finally is needed?

- When exceptions are thrown, execution in a method takes a nonlinear path and changes the normal flow through the method.
  - Sometime exception causes the method to return prematurely.
  - This may cause problems in some cases.
  - E.g a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism.
    - In such situations the code for closing that file and other codes that should not be bypassed should be written inside **finally** block
    - This will ensure that necessary codes are not skipped because of exception handling.

finally(contd.)

- The **finally** block **will execute** whether or not an exception is thrown.
  - If an **exception is thrown**, the **finally** block will execute *even if no catch statement matches the exception*.
  - Any time a method is about to return to the caller from inside a **try/catch block**, (via an uncaught exception or an explicit return statement). the **finally clause is also executed just before the method returns.**
- *If* a **finally** block is associated with a **try**, the finally block will be executed upon conclusion of the try.

finally(contd.)

- The finally clause is optional. However, each try statement requires **at least one catch or a finally clause**

```
try
{
//monitor exception
}
finally
{
}
```

```
try
{
//monitor exception
}
catch(ExceptionType1 ob)
{
}
```

```
try
{
//monitor exception
}
catch(ExceptionType1 ob)
{
}
catch(ExceptionType2 ob)
{
}
//
finally
{
}
```

finally example

```
class FinallyTry
{
public static void main(String[] args)
      {
            try
            {
            int a=5/0;
            }
            catch(ArithmeticException ae)
            {
            System.out.println("Exception is "+ae);
            }
            finally
            {
            System.out.println("Inside finally");
            }
      System.out.println("After try - catch -finally");
      }
}
```

OUTPUT

```
Exception is java.lang.ArithmeticException: / by zero
Inside finally
After try - catch -finally
```

Here int a=5/0; inside try causes ArithmeticException
And it is caught by **catch(ArithmeticException ae)**
And prints the message
Exception is *details about exception*
Then it enters finally block and prints ***Inside finally***
Then it comes out from try catch finally block
and prints the message
After try - catch -finally

finally example

```
class FinallyTry
{
public static void main(String[] args)
      {
            try
            {
            int a=5/2;
            }
            catch(ArithmeticException ae)
            {
            System.out.println("Exception is "+ae);
            }
            finally
            {
            System.out.println("Inside finally");
            }
      System.out.println("After try - catch -finally");
      }
}
```

OUTPUT

```
Inside finally
After try - catch -finally
```

Here int a=5/2; inside try does not cause exception
(So it is not caught by catch(ArithmeticException ae)
)
Then it enters finally block and prints ***Inside finally***
Then it comes out from try catch finally block
and prints the message
After try - catch -finally

finally Example

```
class Sample{
 void show(int n)
 {  int c=10;
    try
    {
    System.out.println("inside try");
    c=10/n;
    }
    catch(Exception e)
    {
    System.out.println("Exception  caught"+e);
    }

    finally
    {
    System.out.println("Finally done");
    }
 }


}
```

```
class Finally1
{
public static void main(String[] args)
      {
            Sample ob=new Sample();
            ob.show(1);
            ob.show(0);

            System.out.println(“Finished");
      }
  }
```

```
inside try
Finally done
inside try
Exception caughtjava.lang.ArithmeticException: / by zero
Finally done
Finished
```

```
class FinallyDemo {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
          }
}
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
          }
}
```

```
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {

System.out.println("procC's finally");
          }
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

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